Contents

Get Started with Xamarin.Android

10/28/2019 • 2 minutes to read • Edit Online

Setup and Installation

Get Xamarin.Android set up and running in Visual Studio. This section covers downloading, installation, emulator configuration, device provisioning, and more.

Hello, Android

In this two-part guide, you'll build your first Xamarin.Android application using Visual Studio, and you'll develop an understanding of the fundamentals of Android application development with Xamarin. Along the way, this guide introduces you to the tools, concepts, and steps required to build and deploy a Xamarin.Android application.

Hello, Android Multiscreen

In this two-part guide, you'll expand the application created in *Hello, Android* so that it implements a second screen. Along the way, you will be introduced to the basic Android *Application Building Blocks* and dive deeper into Android architecture as you develop a better understanding of Android application structure and functionality.

Xamarin for Java Developers

This article provides an introduction to C# programming for Java developers, focusing primarily on the C# language features that Java developers will encounter while learning about Xamarin.Android app development.

Video

Building Your First Android App with Xamarin for Visual Studio

Install and setup Xamarin.Android

11/2/2020 • 2 minutes to read • Edit Online

The topics in this section explain how to install and configure Xamarin.Android to work with Visual Studio on Windows and macOS, how to use the Android SDK Manager to download and install Android SDK tools and components that are required for building and testing your app, how to configure the Android emulator for debugging, and how to connect a physical Android device to your development computer for debugging and final testing your app.

Windows Installation

This guide walks you through the installation steps and configuration details required to install Xamarin.Android on Windows. By the end of this article, you will have a working Xamarin.Android installation integrated into Visual Studio, and you'll be ready to start building your first Xamarin.Android application.

Mac Installation

This article walks you through the installation steps and configuration details required to install Xamarin.Android on a Mac. By the end of this article, you will have a working Xamarin.Android installation integrated into Visual Studio for Mac, and you'll be ready to start building your first Xamarin.Android application.

Android SDK Setup

Visual Studio includes an Android SDK Manager that replaces Google's standalone Android SDK Manager. This article explains how to use the SDK Manager to download Android SDK tools, platforms, and other components that you need for developing Xamarin.Android apps.

Android Emulator Setup

These articles explain how to setup the Android Emulator for testing and debugging Xamarin.Android applications.

Android Device Setup

This article explains how to setup a physical Android device and connect it to a development computer so that the device may be used to run and debug Xamarin.Android applications.

Microsoft Mobile OpenJDK Preview

This guide describes the steps for switching to the preview release of Microsoft's distribution of the OpenJDK. This distribution of the OpenJDK is intended for mobile development.

Windows Installation

7/8/2021 • 5 minutes to read • Edit Online

This guide describes the steps for installing Xamarin.Android for Visual Studio on Windows, and it explains how to configure Xamarin.Android for building your first Xamarin.Android application.

Overview

Because Xamarin is now included with all editions of Visual Studio at no extra cost and does not require a separate license, you can use the Visual Studio installer to download and install Xamarin.Android tools. (The manual installation and licensing steps that were required for earlier versions of Xamarin.Android are no longer necessary.) In this guide, you will learn the following:

- How to configure custom locations for the Java Development Kit, Android SDK, and Android NDK.
- How to launch the Android SDK Manager to download and install additional Android SDK components.
- How to prepare an Android device or emulator for debugging and testing.
- How to create your first Xamarin.Android app project.

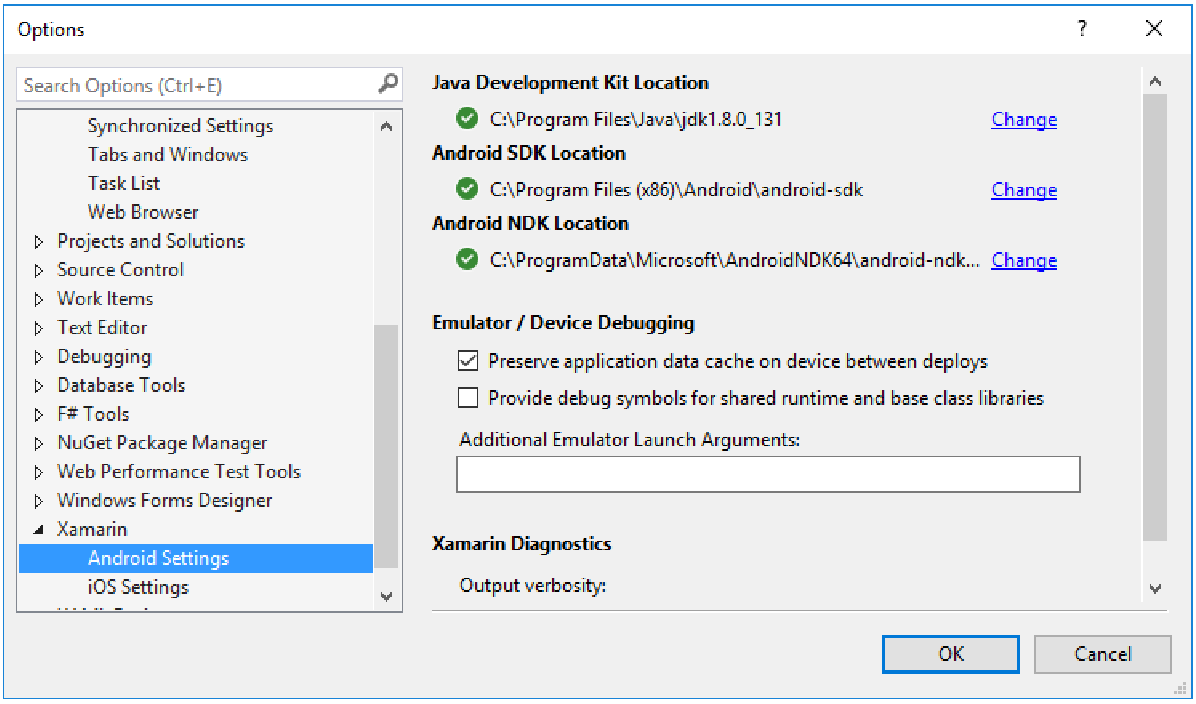
By the end of this guide, you will have a working Xamarin.Android installation integrated into Visual Studio, and you will be ready to start building your first Xamarin.Android application.

Installation

For detailed information on installing Xamarin for use with Visual Studio on Windows, see the Windows Install guide.

Configuration

Xamarin.Android uses the Java Development Kit (JDK) and the Android SDK to build apps. During installation, the Visual Studio installer places these tools in their default locations and configures the development environment with the appropriate path configuration. You can view and change these locations by clicking **Tools > Options > Xamarin > Android Settings**:



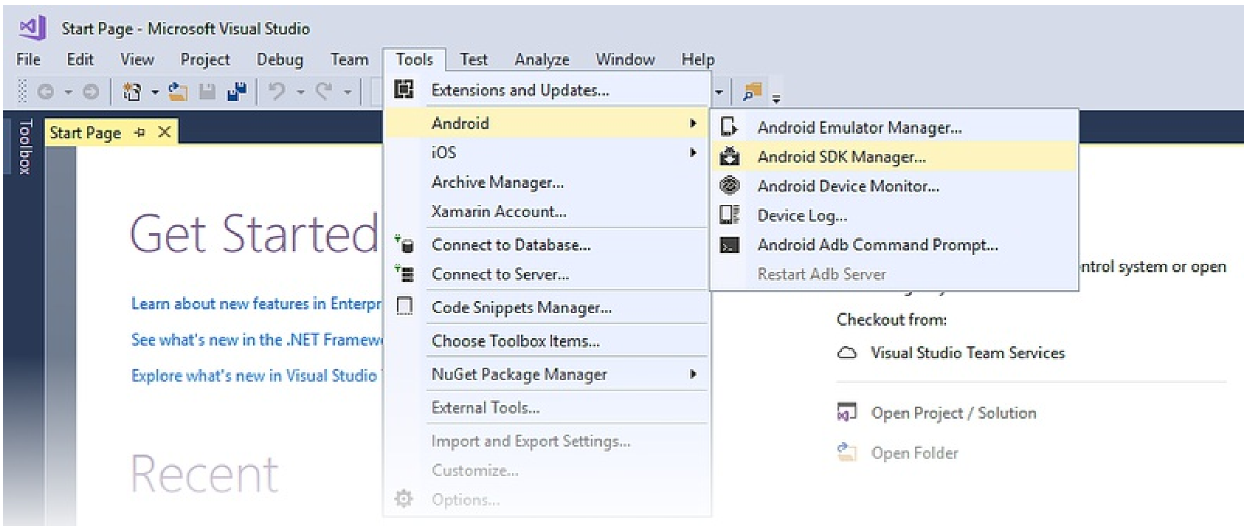
For most users these default locations will work without further changes. However, you may wish to configure Visual Studio with custom locations for these tools (for example, if you have installed the Java JDK, Android SDK, or NDK in a different location). Click **Change** next to a path that you want to change, then navigate to the new location.

Xamarin.Android uses JDK 8, which is required if you are developing for API level 24 or greater (JDK 8 also supports API levels earlier than 24). You can continue to use JDK 7 if you are developing specifically for API level 23 or earlier.

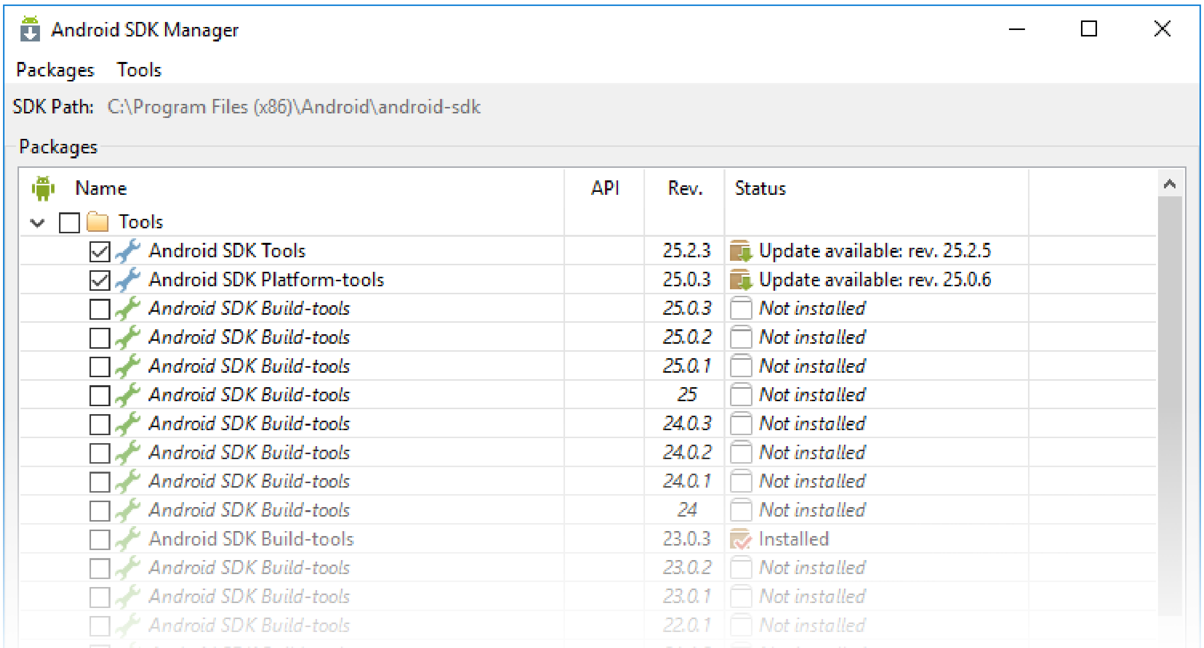
> **IMPORTANT**
>
> Xamarin.Android does not support JDK 9.

Android SDK Manager

Android uses multiple Android API level settings to determine your app's compatibility across the various versions of Android (for more information about Android API levels, see Understanding Android API Levels). Depending on what Android API level(s) you want to target, you may need to download and install additional Android SDK components. In addition, you may need to install optional tools and emulator images provided in the Android SDK. To do this, use the **Android SDK Manager**. You can launch the **Android SDK Manager** by clicking **Tools > Android > Android SDK Manager**:



By default, Visual Studio installs the Google Android SDK Manager:



You can use the Google Android SDK Manager to install versions of the Android SDK Tools package up to version 25.2.3. However, if you need to use a later version of the Android SDK Tools package, you must install the Xamarin Android SDK Manager plugin for Visual Studio (available from the Visual Studio Marketplace). This is necessary because Google's standalone SDK Manager was deprecated in version 25.2.3 of the Android SDK Tools package.

For more information about using the Xamarin Android SDK Manager, see Android SDK Setup.

Android Emulator

The Android Emulator can be helpful tool to develop and test a Xamarin.Android app. For example, a physical device such as a tablet may not be readily available during development, or a developer may want to run some integration tests on their computer before committing code.

Emulating an Android device on a computer involves the following components:

- **Google Android Emulator** – This is an emulator based on QEMU that creates a virtualized device running on the developer's workstation.
- **An Emulator Image** – An *emulator image* is a template or a specification of the hardware and operating system that is meant to be virtualized. For example, one emulator image would identify the hardware requirements for a Nexus 5X running Android 7.0 with Google Play Services installed. Another emulator image might specific a 10" table running Android 6.0.
- **Android Virtual Device (AVD)** – An *Android Virtual Device* is an emulated Android device created from an emulator image. When running and testing Android apps, Xamarin.Android will start the Android Emulator, starting a specific AVD, install the APK, and then run the app.

A significant improvement in performance when developing on x86 based computers can be achieved by using special emulator images that are optimized for x86 architecture and one of two virtualization technologies:

1. Microsoft's Hyper-V – Available on computers running the Windows 10 April 2018 Update or later.
2. Intel's Hardware Accelerated Execution Manager (HAXM) – Available on x86 computers running OS X, macOS, or older versions of Windows.

For more information about the Android Emulator, Hyper-V, and HAXM, please see Hardware Acceleration for Emulator Performance guide.

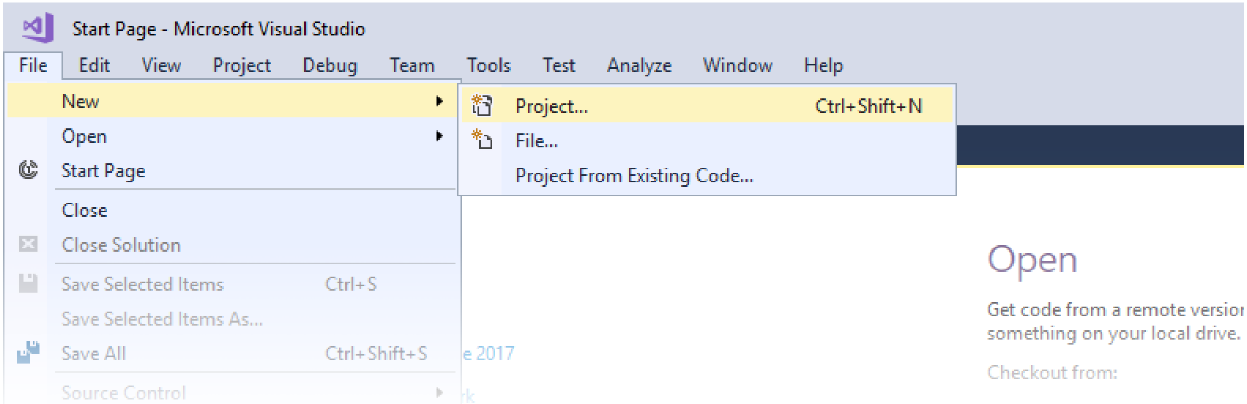
> **NOTE**
>
> On versions of Windows prior to Windows 10 April 2018 Update, HAXM is not compatible with Hyper-V. In this scenario it is necessary to either disable Hyper-V or to use the slower emulator images that do not have the x86 optimizations.

Android Device

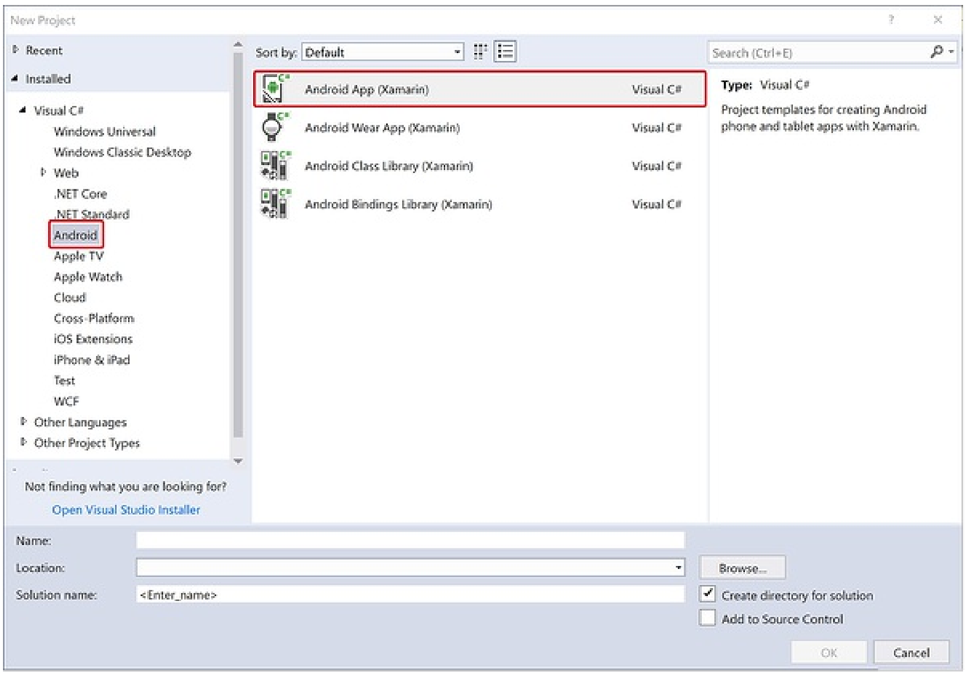
If you have a physical Android device to use for testing, this is a good time to set it up for development use. See Set Up Device for Development to configure your Android device for development, then connect it to your computer for running and debugging Xamarin.Android applications.

Create an Application

Now that you have installed Xamarin.Android, you can launch Visual Studio create a new project. Click **File** > **New** > **Project** to begin creating your app:



In the **New Project** dialog, select **Android** under **Templates** and click **Android App** in the right pane. Enter a name for your app (in the screenshot below, the app is called **MyApp**), then click **OK**:



That's it! Now you are ready to use Xamarin.Android to create Android applications!

Summary

In this article, you learned how to set up and install the Xamarin.Android platform on Windows, how to (optionally) configure Visual Studio with custom Java JDK and Android SDK installation locations, how to launch the SDK Manager to install additional Android SDK components, how to setup an Android device or emulator, and how to start building your first application.

The next step is to have a look at the Hello, Android tutorials to learn how to create a working Xamarin.Android app.

Related Links

- Download Visual Studio
- Installing Visual Studio Tools for Xamarin
- System Requirements
- Android SDK Setup
- Android Emulator Setup
- Set Up Device For Development
- Run Apps on the Android Emulator

Setting up the Android SDK for Xamarin.Android

7/8/2021 • 9 minutes to read • Edit Online

Visual Studio includes an Android SDK Manager that you use to download Android SDK tools, platforms, and other components that you need for developing Xamarin.Android apps.

Overview

This guide explains how to use the Xamarin Android SDK Manager in Visual Studio and Visual Studio for Mac.

> **NOTE**
>
> This guide applies to Visual Studio 2019, Visual Studio 2017, and Visual Studio for Mac.

The Xamarin Android SDK Manager (installed as part of the **Mobile development with .NET** workload) helps you download the latest Android components that you need for developing your Xamarin.Android app. It replaces Google's standalone SDK Manager, which has been deprecated.

- Visual Studio
- Visual Studio for Mac

Requirements

To use the Xamarin Android SDK Manager, you will need the following:

- Visual Studio 2019 Community, Professional, or Enterprise.
- OR Visual Studio 2017 (Community, Professional, or Enterprise edition). Visual Studio 2017 version 15.7 or later is required.
- Visual Studio Tools for Xamarin version 4.10.0 or later (installed as part of the **Mobile development with .NET** workload).

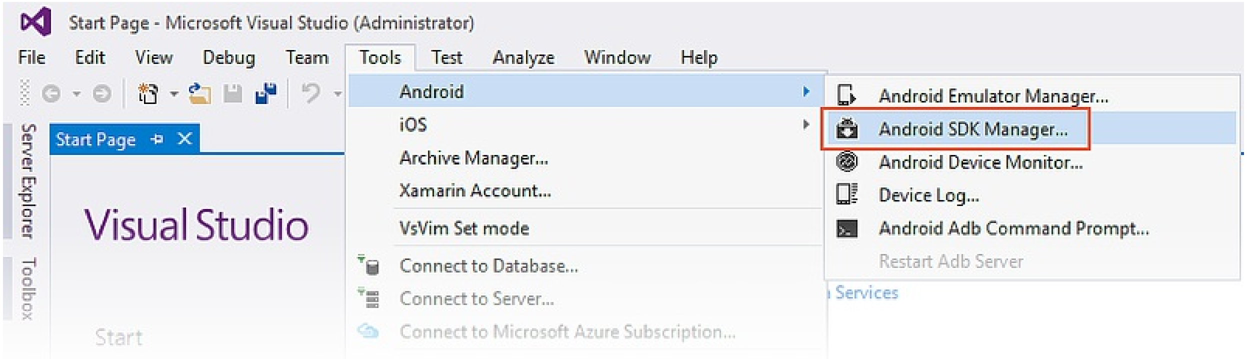
The Xamarin Android SDK Manager also requires the Java Development Kit (which is automatically installed with Xamarin.Android). There are several JDK alternatives to choose from:

- By default, Xamarin.Android uses JDK 8, which is required if you are developing for API level 24 or greater (JDK 8 also supports API levels earlier than 24).
- You can continue to use JDK 7 if you are developing specifically for API level 23 or earlier.
- If you are using Visual Studio 15.8 Preview 5 or later, you can try using Microsoft's Mobile OpenJDK Distribution rather than JDK 8.

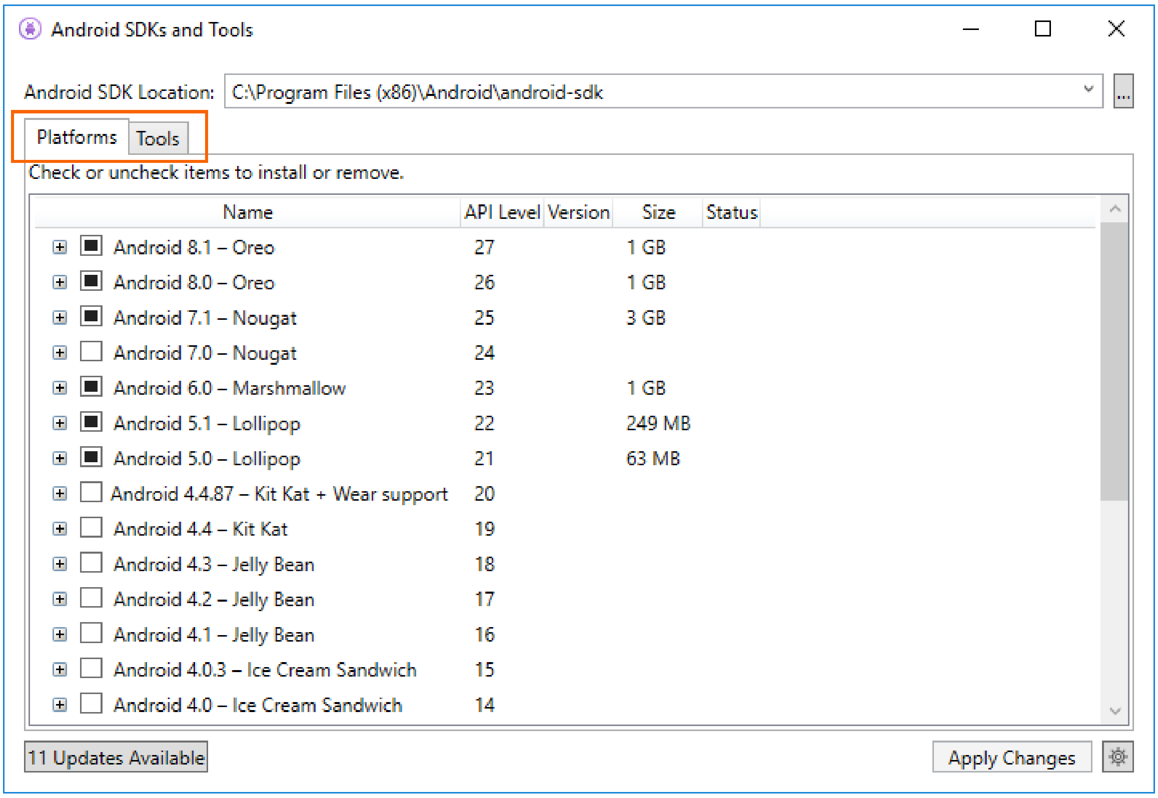
> **IMPORTANT**
>
> Xamarin.Android does not support JDK 9.

SDK Manager

To start the SDK Manager in Visual Studio, click **Tools > Android > Android SDK Manager**:



The Android SDK Manager opens in the **Android SDKs and Tools** screen. This screen has two tabs – **Platforms** and **Tools**:



The **Android SDKs and Tools** screen is described in more detail in the following sections.

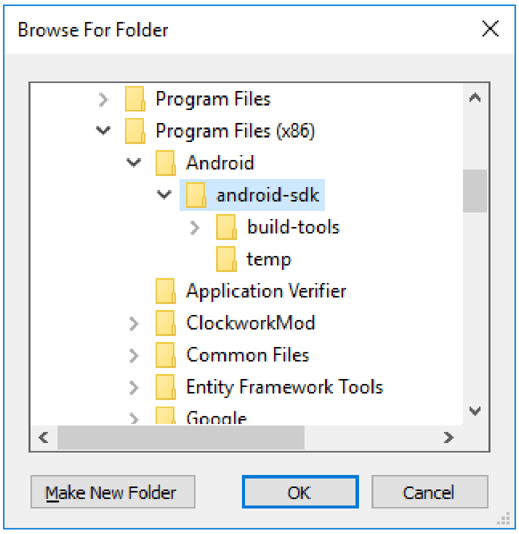
Android SDK location

The Android SDK location is configured at the top of the **Android SDKs and Tools** screen, as seen in the previous screenshot. This location must be configured correctly before the **Platforms** and **Tools** tabs will function properly. You may need to set the location of the Android SDK for one or more of the following reasons:

1. The Android SDK Manager was unable to locate the Android SDK.

2. You have installed the Android SDK in a alternate (non-default) location.

To set the location of the Android SDK, click the ellipsis (...) button to the far right of **Android SDK Location**. This opens the **Browse For Folder** dialog to use for navigating to the location of the Android SDK. In the following screenshot, the Android SDK under **Program Files (x86)\Android** is being selected:

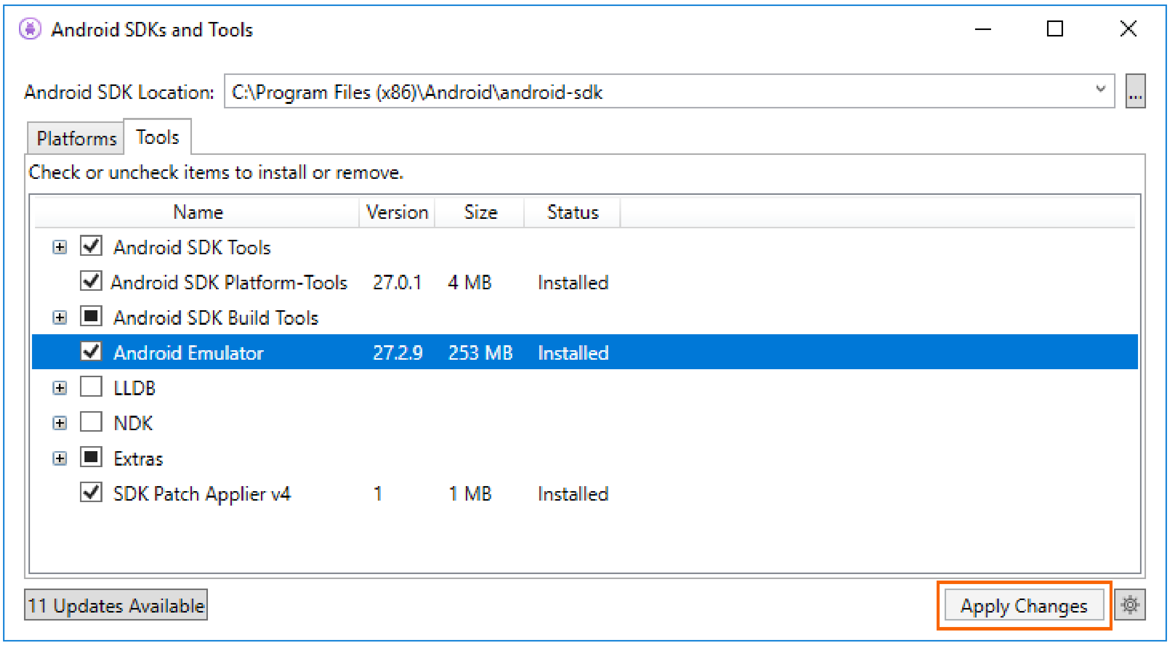


When you click **OK**, the SDK Manager will manage the Android SDK that is installed at the selected location.

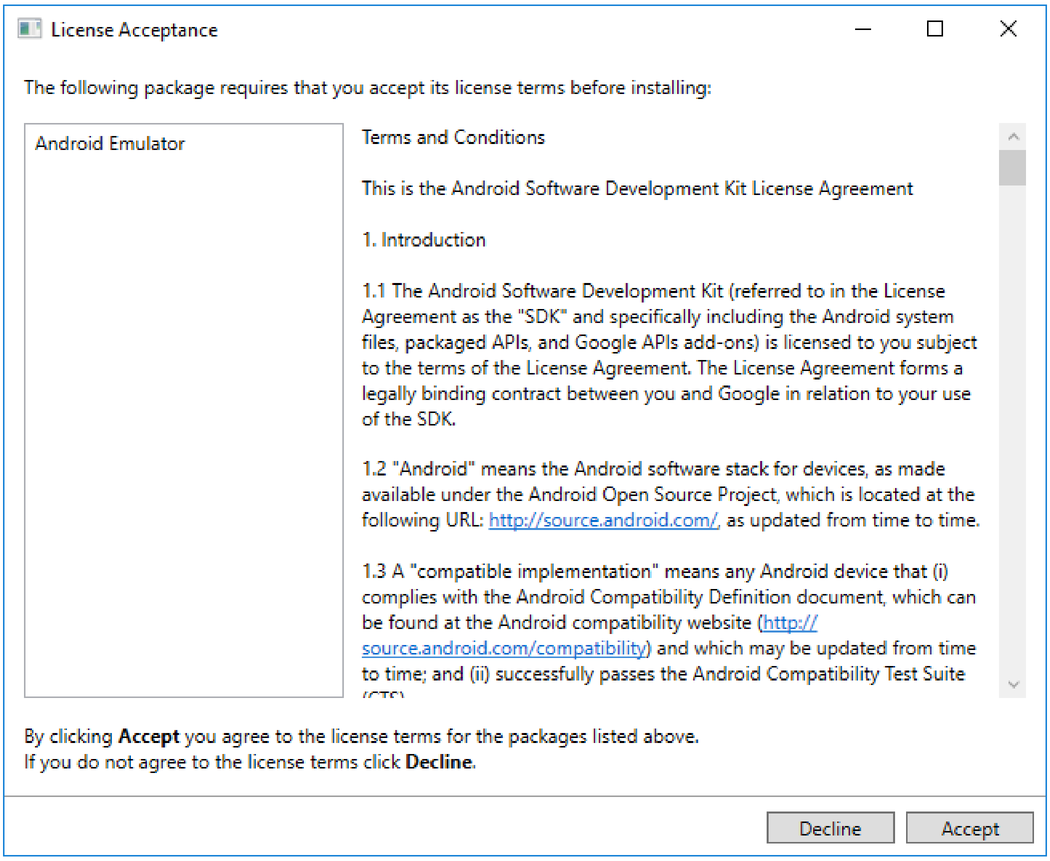
Tools tab

The **Tools** tab displays a list of *tools* and *extras*. Use this tab to install the Android SDK tools, platform tools, and build tools. Also, you can install the Android Emulator, the low-level debugger (LLDB), the NDK, HAXM acceleration, and Google Play libraries.

For example, to download the Google Android Emulator package, click the check mark next to **Android Emulator** and click the **Apply Changes** button:



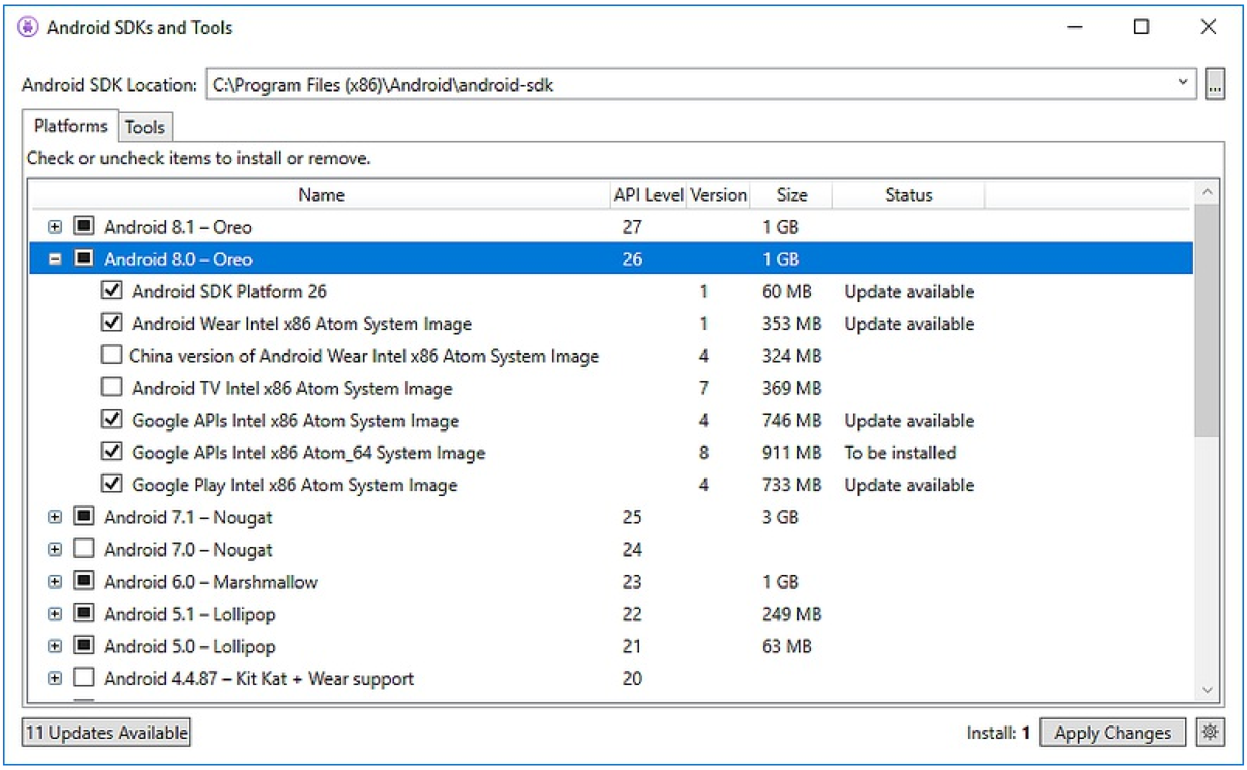
A dialog may be shown with the message, *The following package requires that you accept its license terms before installing*:



Click **Accept** if you accept the Terms and Conditions. At the bottom of the window, a progress bar indicates download and installation progress. After the installation completes, the **Tools** tab will show that the selected tools and extras were installed.

Platforms tab

The **Platforms** tab displays a list of platform SDK versions along with other resources (like system images) for each platform:

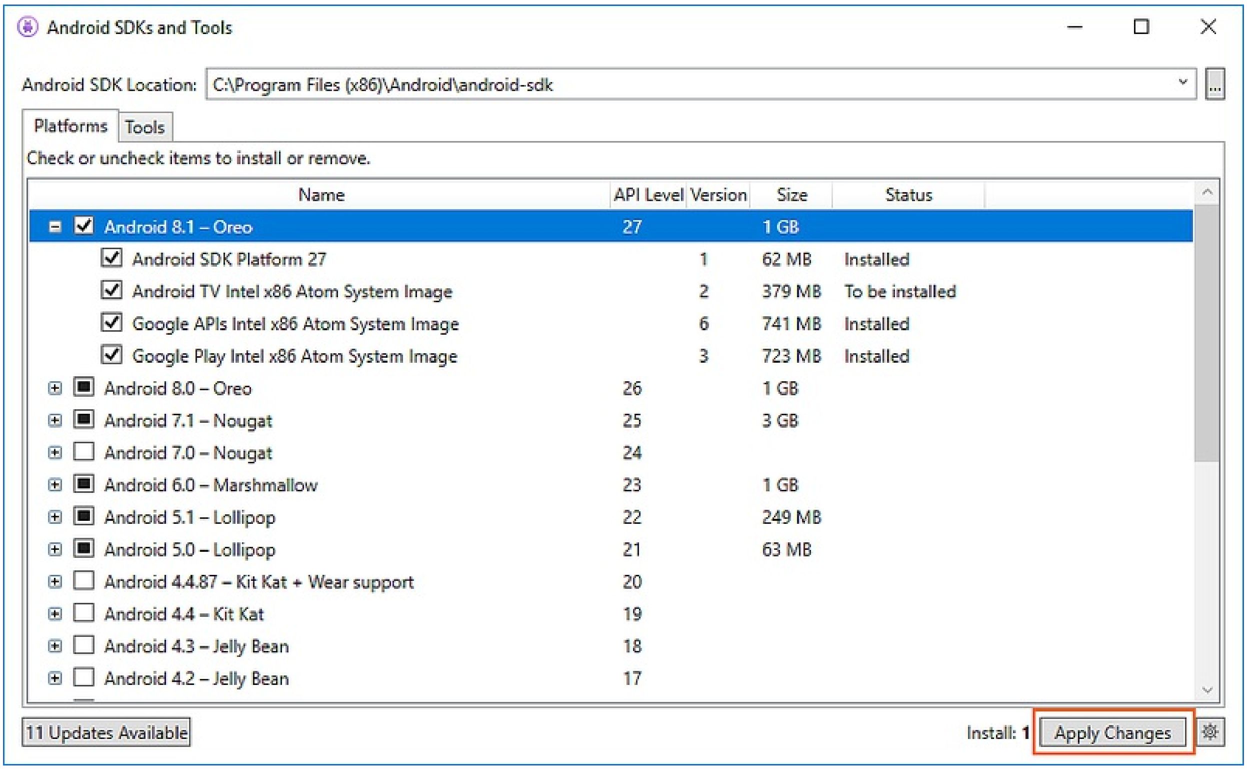


This screen lists the Android version (such as **Android 8.0**), the code name (**Oreo**), the API level (such as **26**), and the sizes of the components for that platform (such as **1 GB**). You use the **Platforms** tab to install components for the Android API level that you want to target. For more information about Android versions and

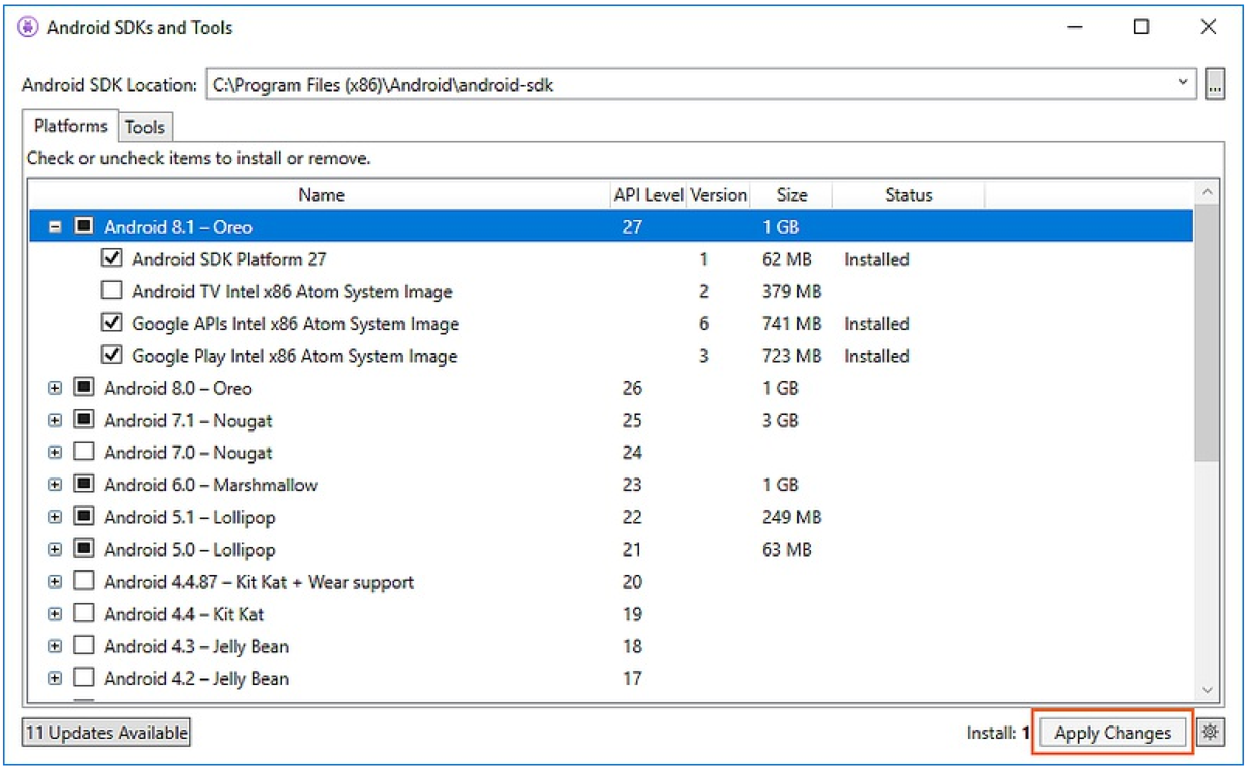
API levels, see Understanding Android API Levels.

When all components of a platform are installed, a checkmark appears next to the platform name. If not all components of a platform are installed, the box for that platform is filled. You can expand a platform to see its components (and which components are installed) by clicking the + box to the left of the platform. Click - to unexpand the component listing for a platform.

To add another platform to the SDK, click the box next to the platform until the checkmark appears to install all of its components, then click **Apply Changes**:



To install only specific components, click the box next to the platform once. You can then select any individual components that you need:

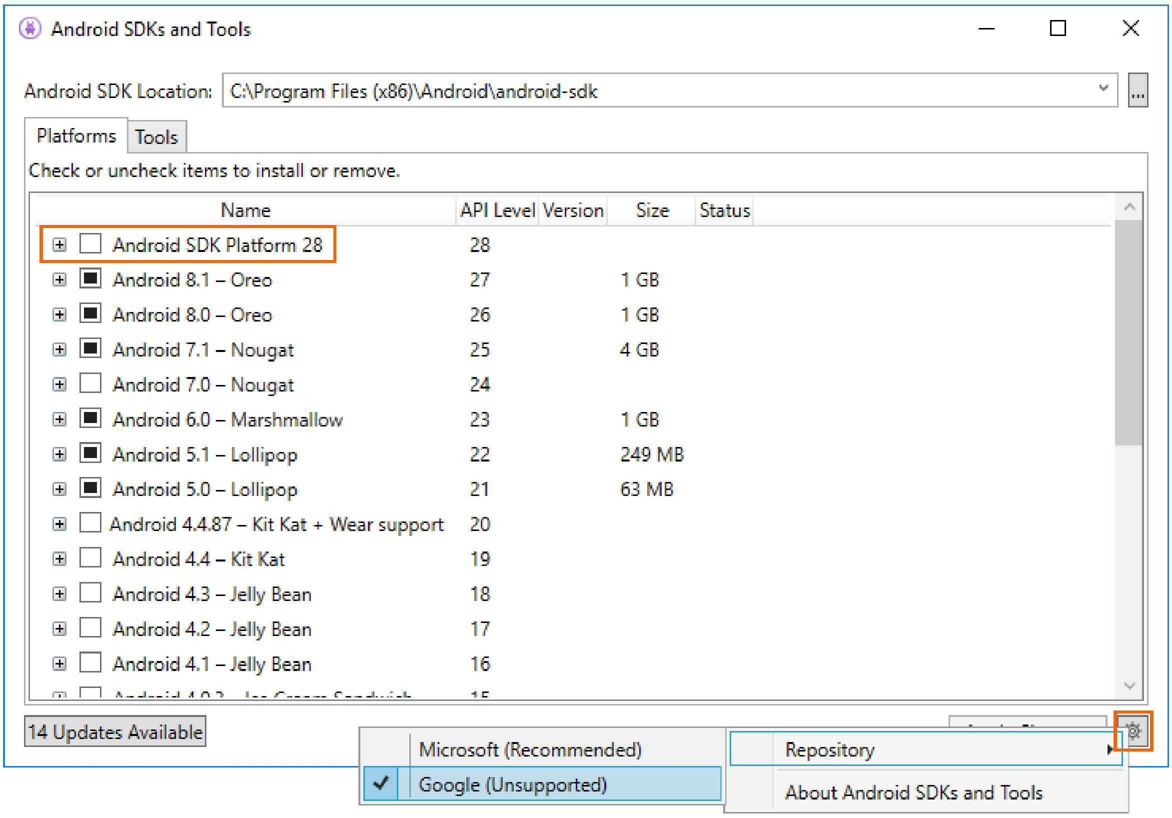


Notice that the number of components to install appears next to the **Apply Changes** button. After you click the

Apply Changes button, you will see the **License Acceptance** screen as shown earlier. Click **Accept** if you accept the Terms and Conditions. You may see this dialog more than one time when there are multiple components to install. At the bottom of the window, a progress bar will indicate download and installation progress. When the download and installation process completes (this can take many minutes, depending on how many components need to be downloaded), the added components are marked with a checkmark and listed as **Installed**.

Repository selection

By default, the Android SDK Manager downloads platform components and tools from a Microsoft-managed repository. If you need access to experimental alpha/beta platforms and tools that are not yet available in the Microsoft repository, you can switch the SDK Manager to use Google's repository. To make this switch, click the gear icon in the lower right-hand corner and select **Repository > Google (Unsupported)**:



When the Google repository is selected, additional packages may appear in the **Platforms** tab that were not available previously. (In the above screenshot, **Android SDK Platform 28** was added by switching to the Google repository.) Keep in mind that use of the Google repository is unsupported and is therefore not recommended for everyday development.

To switch back to the supported repository of platforms and tools, click **Microsoft (Recommended)**. This restores the list of packages and tools to the default selection.

Summary

This guide explained how to install and use the Xamarin Android SDK Manager tool in Visual Studio and Visual Studio for Mac.

Related Links

- [Understanding Android API levels]
- [Changes to the Android SDK Tooling]

Android Emulator Setup

4/7/2021 • 2 minutes to read • Edit Online

This guide explains how to prepare the Android Emulator for testing your app.

Overview

The Android Emulator can be run in a variety of configurations to simulate different devices. Each configuration is called a *virtual device*. When you deploy and test your app on the emulator, you select a pre-configured or custom virtual device that simulates a physical Android device such as a Nexus or Pixel phone.

The sections listed below describe how to accelerate the Android emulator for maximum performance, how to use the Android Device Manager to create and customize virtual devices, and how to customize the profile properties of a virtual device. In addition, a troubleshooting section explains common emulator problems and workarounds.

Sections

Hardware Acceleration for Emulator Performance

How to prepare your computer for maximum Android Emulator performance by using either Hyper-V or HAXM virtualization technology. Because the Android Emulator can be prohibitively slow without hardware acceleration, we recommend that you enable hardware acceleration on your computer before you use the emulator.

Managing Virtual Devices with the Android Device Manager

How to use the Android Device Manager to create and customize virtual devices.

Editing Android Virtual Device Properties

How to use the Android Device Manager to edit the profile properties of a virtual device.

Android Emulator Troubleshooting

In this article, the most common warning messages and issues that occur while running the Android Emulator are described, along with workarounds and tips.

> **NOTE**
>
> If you are using a Mac with an Apple chip, such as the M1, you will need to install the Android Emulator for M1 preview from GitHub.

After you have configured the Android Emulator, see Debugging on the Android Emulator for information about how to launch the emulator and use it for testing and debugging your app.

> **NOTE**
>
> As of Android SDK Tools version `26.0.1` and later, Google has removed support for existing AVD/SDK managers in favor of their new CLI (Command Line Interface) tools. Because of this deprecation change, Xamarin SDK/Device Managers are now used instead of Google SDK/Device Managers for Android Tools 26.0.1 and later. For more information about the Xamarin SDK Manager, see Setting up the Android SDK for Xamarin.Android.

Hardware acceleration for emulator performance (Hyper-V & HAXM)

7/8/2021 • 7 minutes to read • Edit Online

This article explains how to use your computer's hardware acceleration features to maximize Android Emulator performance.

Visual Studio makes it easier for developers to test and debug their Xamarin.Android applications by using the Android emulator in situations where an Android device is unavailable or impractical. However, the Android emulator runs too slowly if hardware acceleration is not available on the computer that runs it. You can drastically improve the performance of the Android emulator by using special x86 virtual device images in conjunction with the virtualization features of your computer.

SCENARIO	HAXM	WHPX	HYPERVISOR.FRAMEWORK
You have an Intel Processor	X	X	X
You have an AMD Processor		X	
You want to support Hyper-V		X	
You want to support nested Virtualization		Limited	
You want to use technologies like Docker	(with WSL2)	X	X

Accelerating Android emulators on Windows

The following virtualization technologies are available for accelerating the Android emulator:

1. **Microsoft's Hyper-V and the Windows Hypervisor Platform (WHPX)**. Hyper-V is a virtualization feature of Windows that makes it possible to run virtualized computer systems on a physical host computer.
2. **Intel's Hardware Accelerated Execution Manager (HAXM)**. HAXM is a virtualization engine for computers running Intel CPUs.

For the best experience on Windows, it is recommended that you use WHPX to accelerate the Android emulator. If WHPX is not available on your computer, then HAXM can be used. The Android emulator will automatically make use of hardware acceleration if the following criteria are met:

- Hardware acceleration is available and enabled on your development computer.
- The emulator is running a system image created for an **x86**-based virtual device.

> **IMPORTANT**
>
> You can't run a VM-accelerated emulator inside another VM, such as a VM hosted by VirtualBox, VMware, or Docker (unless using WSL2). You must run the Android emulator directly on your system hardware.

For information about launching and debugging with the Android emulator, see Debugging on the Android Emulator.

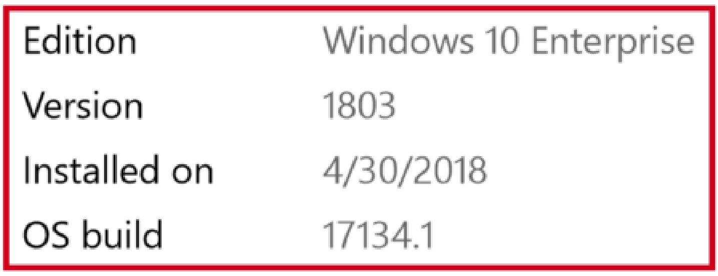
Accelerating with Hyper-V

Before enabling Hyper-V, read the following section to verify that your computer supports Hyper-V.

Verifying support for Hyper-V

Hyper-V runs on the Windows Hypervisor Platform. To use the Android emulator with Hyper-V, your computer must meet the following criteria to support the Windows Hypervisor Platform:

- Your computer hardware must meet the following requirements:
  - A 64-bit Intel or AMD Ryzen CPU with Second Level Address Translation (SLAT).
  - CPU support for VM Monitor Mode Extension (VT-c on Intel CPUs).
  - Minimum of 4-GB memory.
- In your computer's BIOS, the following items must be enabled:
  - Virtualization Technology (may have a different label depending on motherboard manufacturer).
  - Hardware Enforced Data Execution Prevention.
- Your computer must be updated to Windows 10 April 2018 update (build 1803) or later. You can verify that your Windows version is up-to-date by using the following steps:
  1. Enter **About** in the Windows search box.
  2. Select **About your PC** in the search results.
  3. Scroll down in the **About** dialog to the **Windows specifications** section.
  4. Verify that the **Version** is at least 1803:



Edition	Windows 10 Enterprise
Version	1803
Installed on	4/30/2018
OS build	17134.1

To verify that your computer hardware and software is compatible with Hyper-V, open a command prompt and type the following command:

```
systeminfo
```

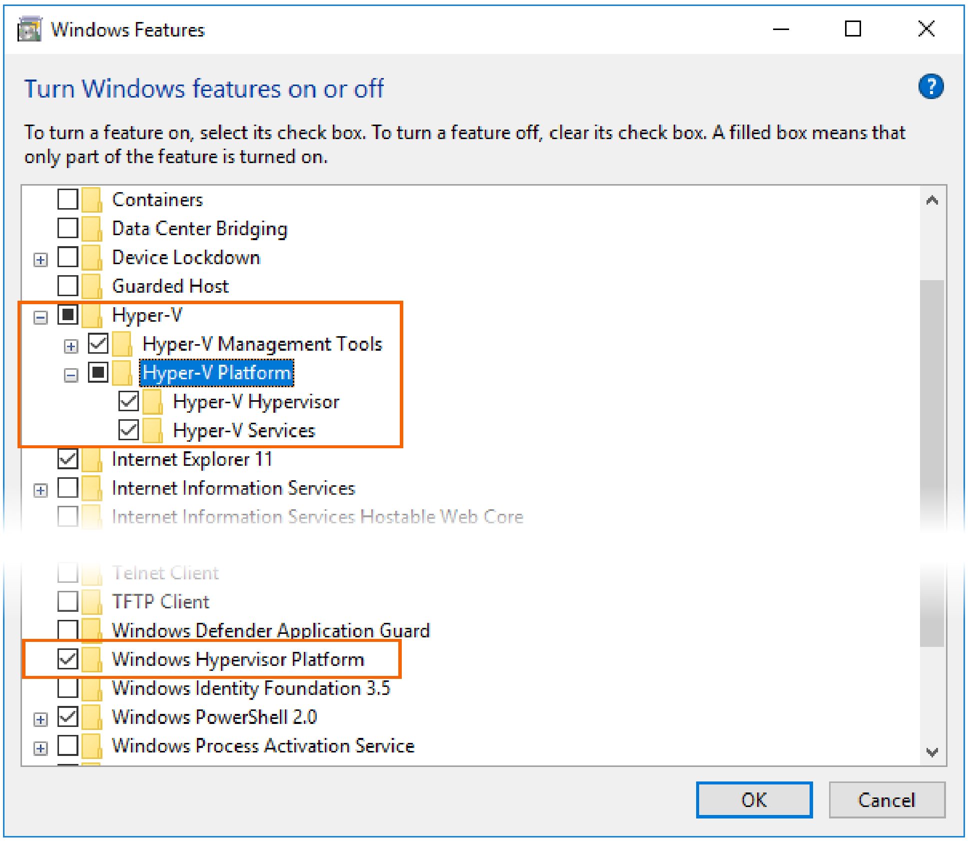
If all listed Hyper-V requirements have a value of **Yes**, then your computer can support Hyper-V. For example:

```
Hyper-V Requirements:     VM Monitor Mode Extensions: Yes
                          Virtualization Enabled In Firmware: Yes
                          Second Level Address Translation: Yes
                          Data Execution Prevention Available: Yes
```

Enabling Hyper-V acceleration

If your computer meets the above criteria, use the following steps to accelerate the Android emulator with Hyper-V:

1. Enter **windows features** in the Windows search box and select **Turn Windows features on or off** in the search results. In the **Windows Features** dialog, enable both **Hyper-V** and **Windows Hypervisor Platform**:

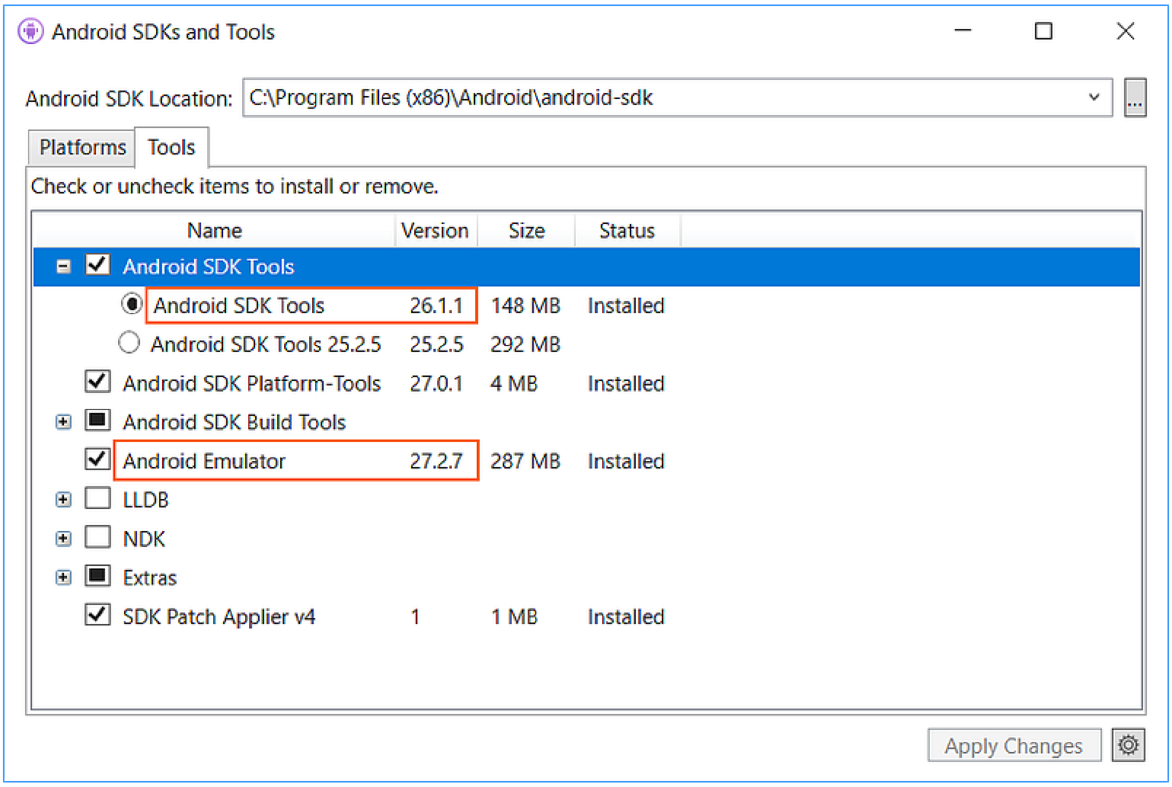


After making these changes, reboot your computer.

> **IMPORTANT**
>
> On Windows 10 October 2018 Update (RS5) and higher, you only need to enable Hyper-V, as it will use Windows Hypervisor Platform (WHPX) automatically.

2. **Install Visual Studio 15.8 or later** (this version of Visual Studio provides IDE support for running the Android emulator with Hyper-V).

3. **Install the Android Emulator package 27.2.7 or later**. To install this package, navigate to **Tools > Android > Android SDK Manager** in Visual Studio. Select the **Tools** tab and ensure that the Android emulator version is at least 27.2.7. Also ensure that the Android SDK Tools version is 26.1.1 or later:



When you create a virtual device (see Managing Virtual Devices with the Android Device Manager), be sure to select an **x86**-based system image. If you use an ARM-based system image, the virtual device will not be accelerated and will run slowly.

Hyper-V should now be enabled and you can run your accelerated Android emulator.

Accelerating with HAXM

If your computer does not support Hyper-V, you may use HAXM to accelerate the Android emulator. You must disable Device Guard if you want to use HAXM.

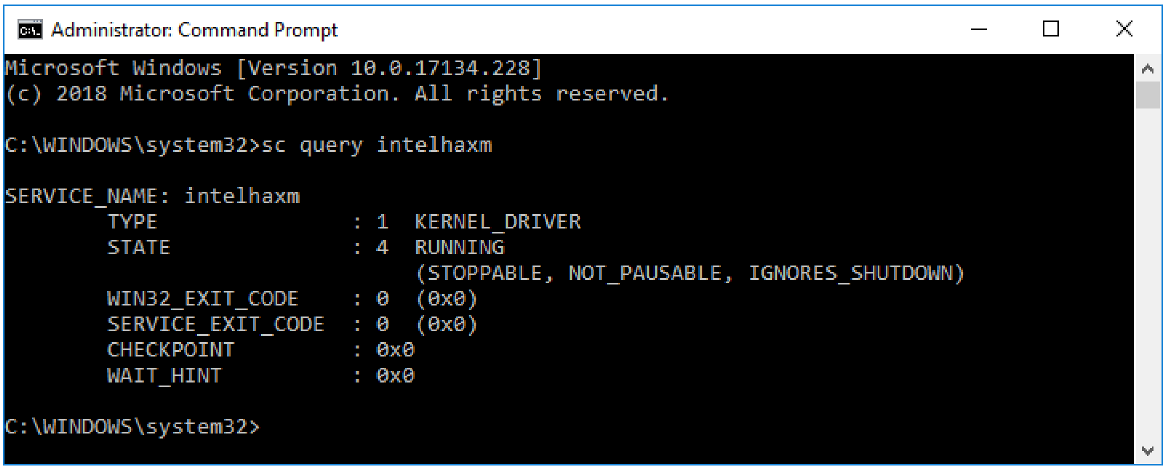
Verifying HAXM support

To determine if your hardware supports HAXM, follow the steps in Does My Processor Support Intel Virtualization Technology?. If your hardware supports HAXM, you can check to see if HAXM is already installed by using the following steps:

1. Open a command prompt window and enter the following command:

```
sc query intelhaxm
```

2. Examine the output to see if the HAXM process is running. if it is, you should see output listing the `intelhaxm` state as `RUNNING`. For example:



If `STATE` is not set to `RUNNING`, then HAXM is not installed.

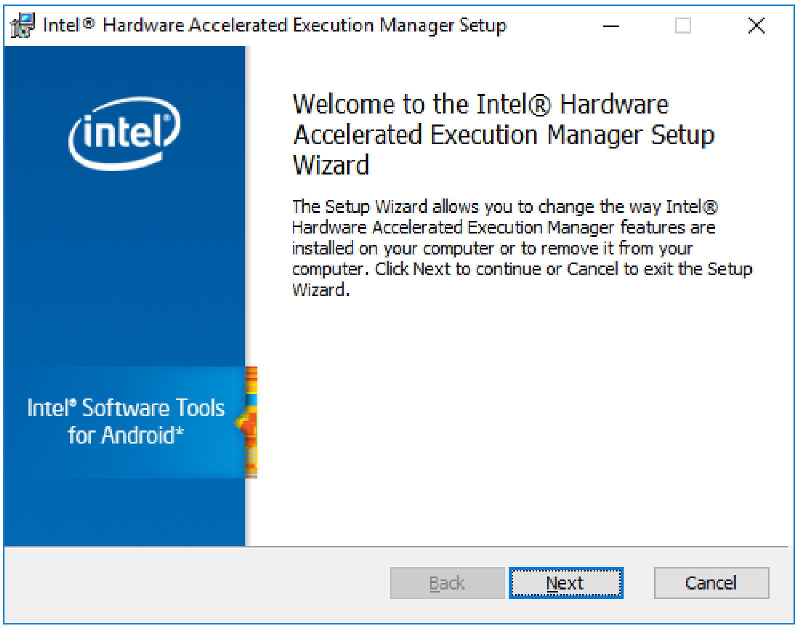
If your computer can support HAXM but HAXM is not installed, use the steps in the next section to install HAXM.

Installing HAXM

HAXM install packages for Windows are available from the Intel Hardware Accelerated Execution Manager GitHub releases page. Use the following steps to download and install HAXM:

1. From the Intel website, download the latest HAXM virtualization engine installer for Windows. The advantage of downloading the HAXM installer directly from the Intel website is that you can be assured of using the latest version.

2. Run **intelhaxm-android.exe** to start the HAXM installer. Accept the default values in the installer dialogs:



When you create a virtual device (see Managing Virtual Devices with the Android Device Manager), be sure to select an **x86**-based system image. If you use an ARM-based system image, the virtual device will not be accelerated and will run slowly.

Troubleshooting

For help with troubleshooting hardware acceleration issues, see the Android emulator Troubleshooting guide.

Accelerating Android emulators on macOS

The following virtualization technologies are available for accelerating the Android emulator:

1. **Apple's Hypervisor Framework**. Hypervisor is a feature of macOS 10.10 and later that makes it possible to run virtual machines on a Mac.

2. **Intel's Hardware Accelerated Execution Manager (HAXM)**. HAXM is a virtualization engine for computers running Intel CPUs.

It is recommended that you use the Hypervisor Framework to accelerate the Android emulator. If the Hypervisor Framework is not available on your Mac, then HAXM can be used. The Android emulator will automatically make use of hardware acceleration if the following criteria are met:

- Hardware acceleration is available and enabled on the development computer.

- The emulator is running a system image created for an **x86**-based virtual device.

> **IMPORTANT**
>
> You can't run a VM-accelerated emulator inside another VM, such as a VM hosted by VirtualBox, VMware, or Docker. You must run the Android emulator directly on your system hardware.

For information about launching and debugging with the Android emulator, see Debugging on the Android Emulator.

Accelerating with the Hypervisor Framework

To use the Android emulator with the Hypervisor Framework, your Mac must meet the following criteria:

- Your Mac must be running macOS 10.10 or later.

- Your Mac's CPU must be able to support the Hypervisor Framework.

If your Mac meets these criteria, the Android emulator will automatically use the Hypervisor Framework for acceleration. If you are not sure if Hypervisor Framework is supported on your Mac, see the Troubleshooting guide for ways to verify that your Mac supports Hypervisor.

If the Hypervisor Framework is not supported by your Mac, you can use HAXM to accelerate the Android emulator (described next).

Accelerating with HAXM

If your Mac does not support the Hypervisor framework (or you are using a version of macOS earlier than 10.10), you can use **Intel's Hardware Accelerated Execution Manager** (HAXM) to speed up the Android emulator.

Before using the Android emulator with HAXM for the first time, it's a good idea to verify that HAXM is installed and available for the Android emulator to use.

Verifying HAXM support

You can check to see if HAXM is already installed by using the following steps:

1. Open a Terminal and enter the following command:

   ```
   ~/Library/Developer/Xamarin/android-sdk-macosx/tools/emulator -accel-check
   ```

   This command assumes that the Android SDK is installed at the default location of

~/Library/Developer/Xamarin/android-sdk-macosx; if not, modify the above path for the location of the Android SDK on your Mac.

2. If HAXM is installed, the above command will return a message similar to the following result:

```
HAXM version 7.2.0 (3) is installed and usable.
```

If HAXM is *not* installed, a message similar to the following output is returned:

```
HAXM is not installed on this machine (/dev/HAX is missing).
```

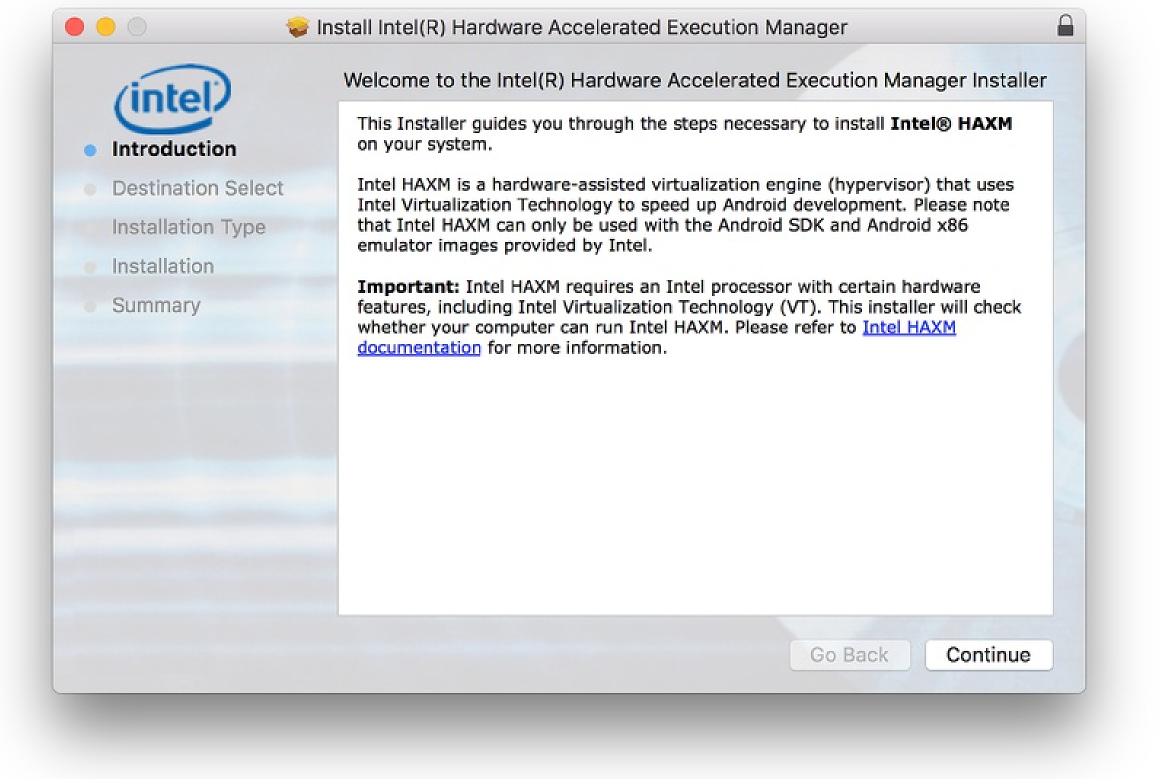
If HAXM is not installed, use the steps in the next section to install HAXM.

Installing HAXM

HAXM installation packages for macOS are available from the Intel Hardware Accelerated Execution Manager page. Use the following steps to download and install HAXM:

1. From the Intel website, download the latest HAXM virtualization engine installer for macOS.

2. Run the HAXM installer. Accept the default values in the installer dialogs:



Troubleshooting

For help with troubleshooting hardware acceleration issues, see the Android emulator Troubleshooting guide.

Related Links

- Run Apps on the Android Emulator

Managing Virtual Devices with the Android Device Manager

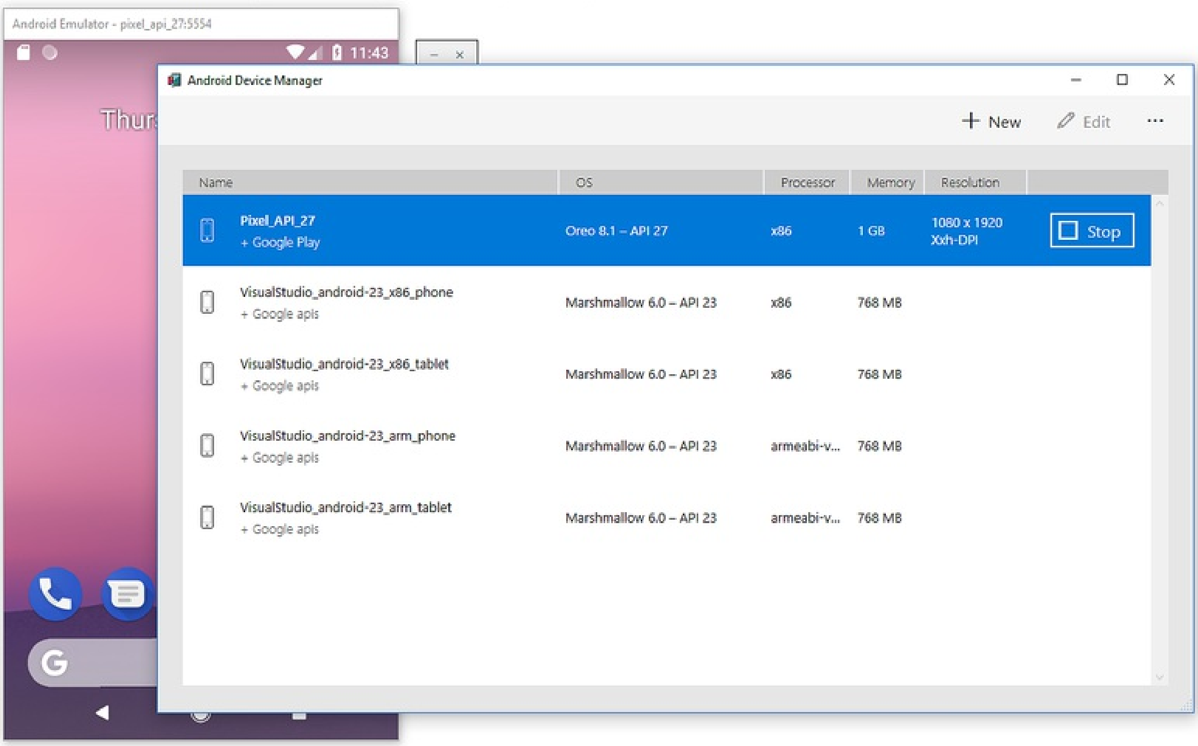
7/8/2021 • 14 minutes to read • Edit Online

This article explains how to use the Android Device Manager to create and configure Android Virtual Devices (AVDs) that emulate physical Android devices. You can use these virtual devices to run and test your app without having to rely on a physical device.

After you have verified that hardware acceleration is enabled (as described in Hardware Acceleration for Emulator Performance), the next step is to use the *Android Device Manager* (also referred to as the *Xamarin Android Device Manager*) to create virtual devices that you can use to test and debug your app.

Android Device Manager on Windows

This article explains how to use the Android Device Manager to create, duplicate, customize, and launch Android virtual devices.



You use the Android Device Manager to create and configure *Android Virtual Devices* (AVDs) that run in the Android Emulator. Each AVD is an emulator configuration that simulates a physical Android device. This makes it possible to run and test your app in a variety of configurations that simulate different physical Android devices.

Requirements

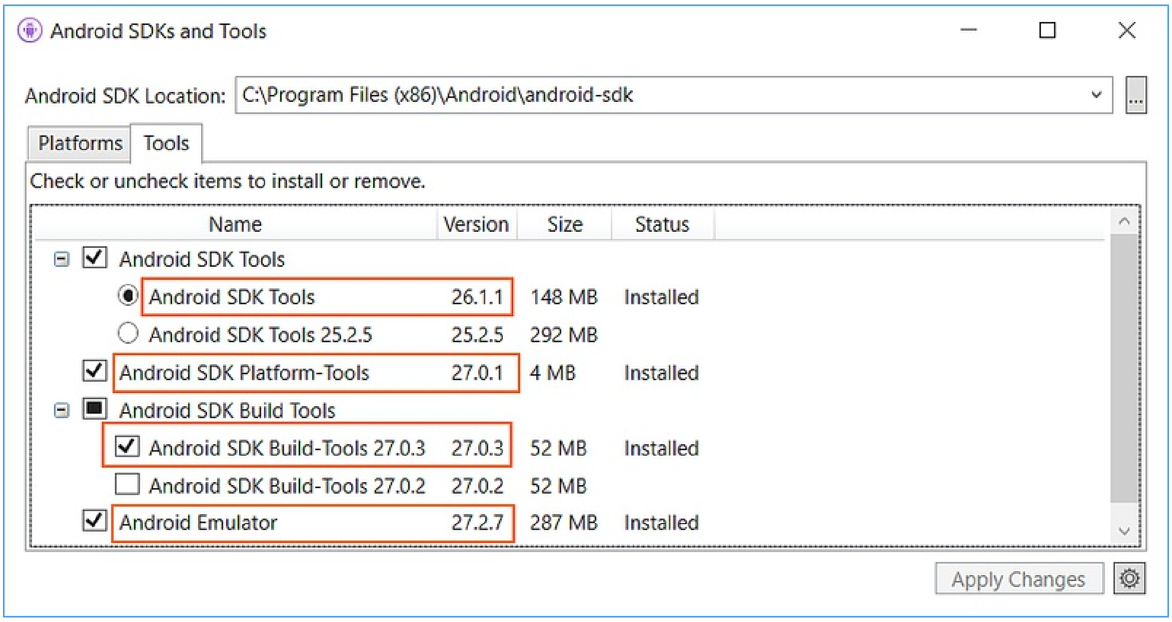
To use the Android Device Manager, you will need the following items:

- Visual Studio 2019 Community, Professional, or Enterprise.
- OR Visual Studio 2017 version 15.8 or later is required. Visual Studio Community, Professional, and Enterprise editions are supported.
- Visual Studio Tools for Xamarin version 4.9 or later.

- The Android SDK must be installed (see Setting up the Android SDK for Xamarin.Android). Be sure to install the Android SDK at its default location if it is not already installed: **C:\Program Files (x86)\Android\android-sdk**.

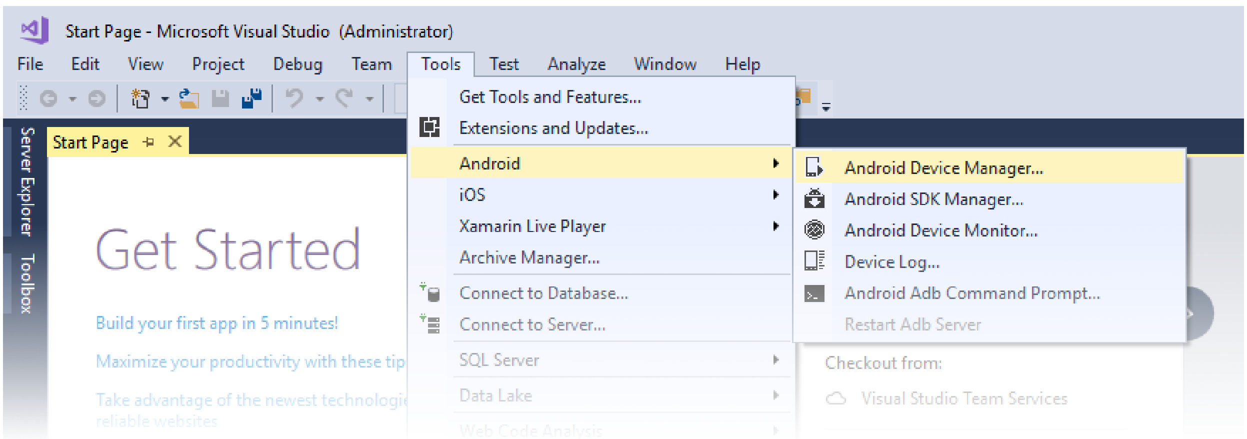
- The following packages must be installed (via the Android SDK Manager):
  - **Android SDK Tools version 26.1.1** or later
  - **Android SDK Platform-Tools 27.0.1** or later
  - **Android SDK Build-Tools 27.0.3** or later
  - **Android Emulator 27.2.7** or later.

  These packages should be displayed with **Installed** status as seen in the following screenshot:

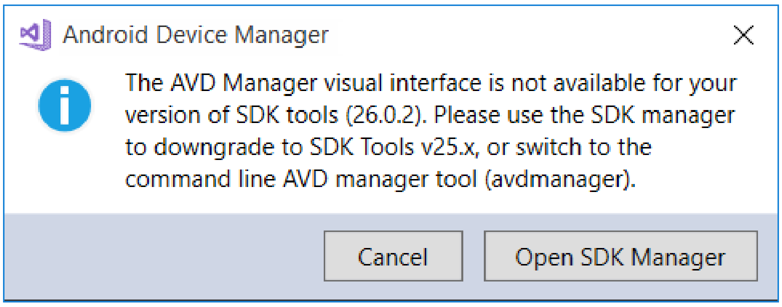


Launching the Device Manager

Launch the Android Device Manager from the **Tools** menu by clicking **Tools > Android > Android Device Manager**:

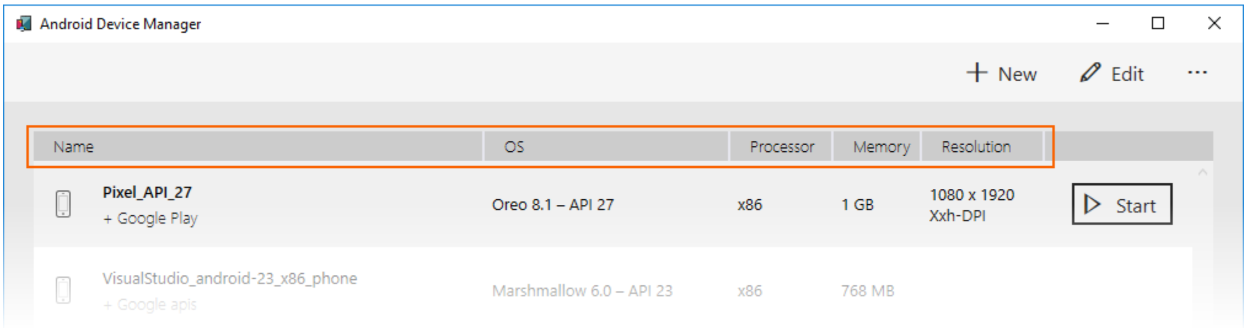


If the following error dialog is presented on launch, see the Troubleshooting section for workaround instructions:

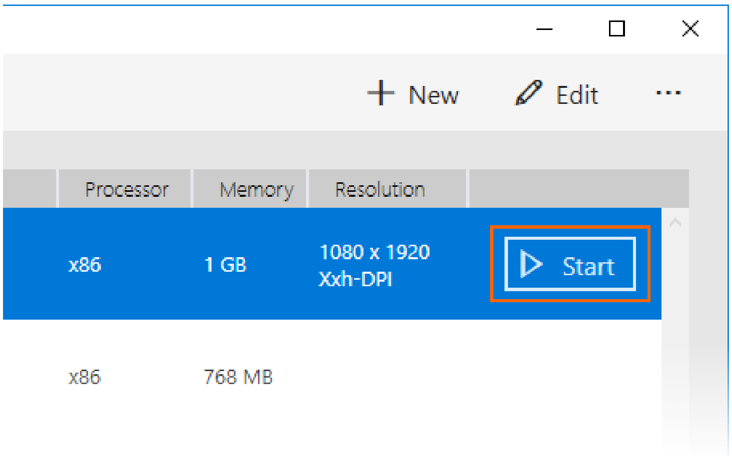


Main Screen

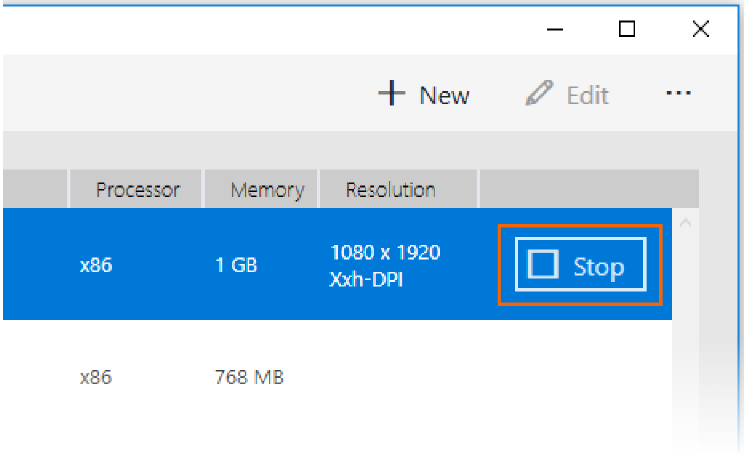
When you first launch the Android Device Manager, it presents a screen that displays all currently-configured virtual devices. For each virtual device, the **Name**, **OS** (Android Version), **Processor**, **Memory** size, and screen **Resolution** are displayed:



When you select a device in the list, the **Start** button appears on the right. You can click the **Start** button to launch the emulator with this virtual device:

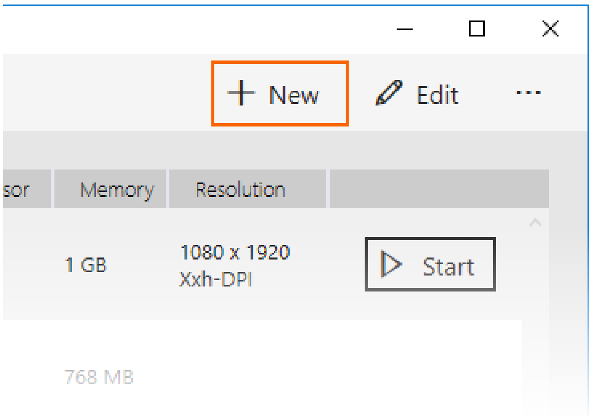


After the emulator starts with the selected virtual device, the **Start** button changes to a **Stop** button that you can use to halt the emulator:

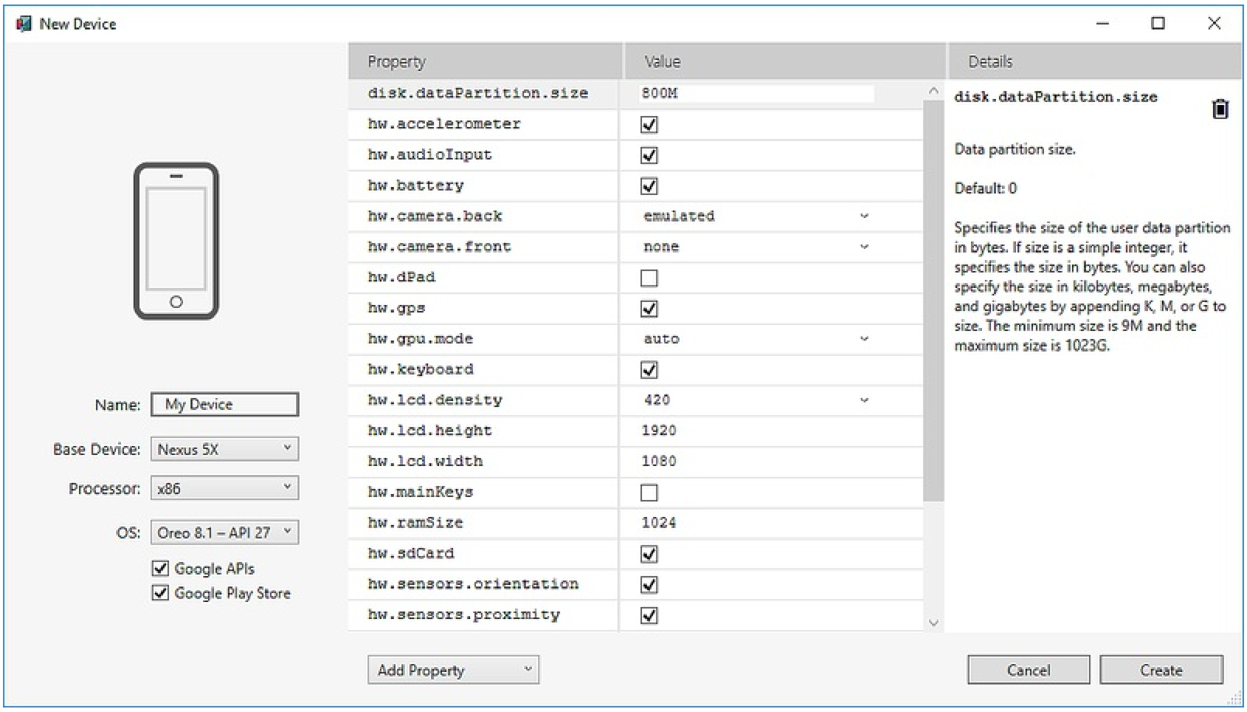


New Device

To create a new device, click the **New** button (located in the upper right-hand area of the screen):

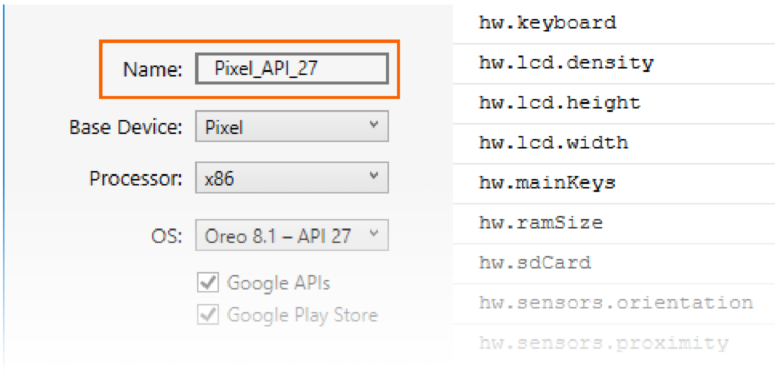


Clicking **New** launches the **New Device** screen:

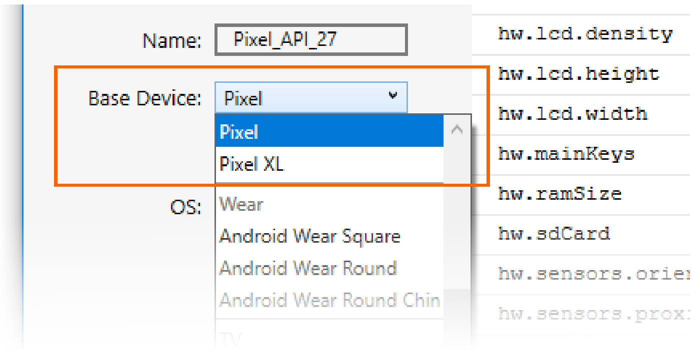


To configure a new device in the **New Device** screen, use the following steps:

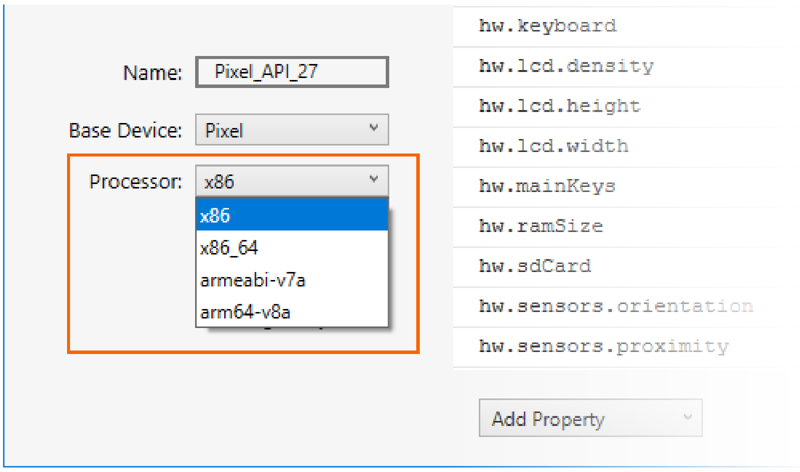
1. Give the device a new name. In the following example, the new device is named **Pixel_API_27**:



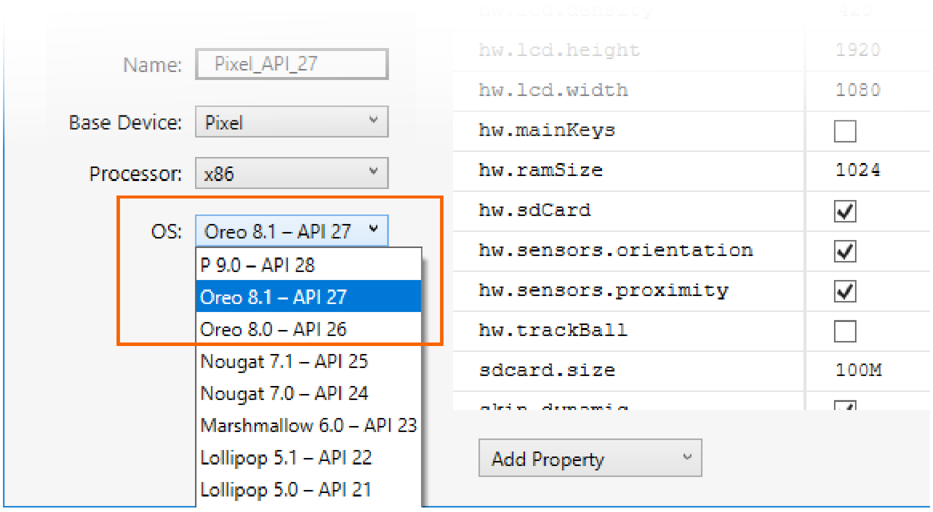
2. Select a physical device to emulate by clicking the **Base Device** pull-down menu:



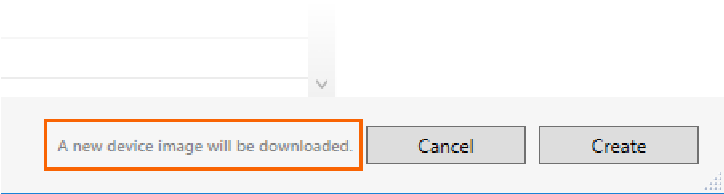
3. Select a processor type for this virtual device by clicking the **Processor** pull-down menu. Selecting **x86** will provide the best performance because it enables the emulator to take advantage of [hardware acceleration](). The **x86_64** option will also make use of hardware acceleration, but it runs slightly slower than **x86** (**x86_64** is normally used for testing 64-bit apps):



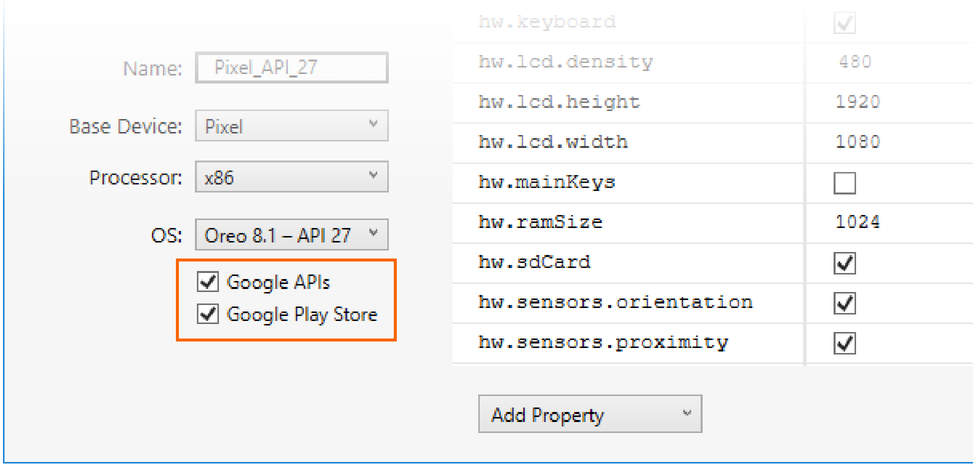
4. Select the Android version (API level) by clicking the **OS** pull-down menu. For example, select **Oreo 8.1 - API 27** to create a virtual device for API level 27:



If you select an Android API level that has not yet been installed, the Device Manager will display **A new device will be downloaded** message at the bottom of the screen – it will download and install the necessary files as it creates the new virtual device:



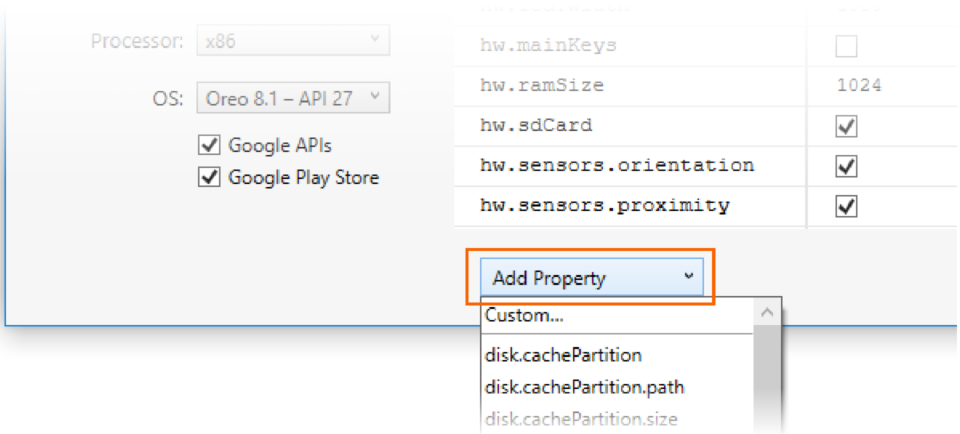
5. If you want to include Google Play Services APIs in your virtual device, enable the **Google APIs** option. To include the Google Play Store app, enable the **Google Play Store** option:



Note that Google Play Store images are available only for some base device types such as Pixel, Pixel 2, Nexus 5, and Nexus 5X.

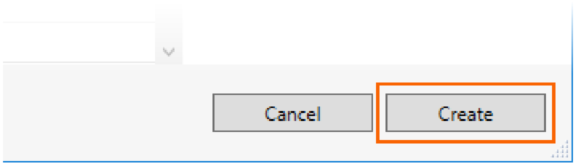
6. Edit any properties that you need to modify. To make changes to properties, see Editing Android Virtual Device Properties.

7. Add any additional properties that you need to explicitly set. The **New Device** screen lists only the most commonly-modified properties, but you can click the **Add Property** pull-down menu (at the bottom) to add additional properties:

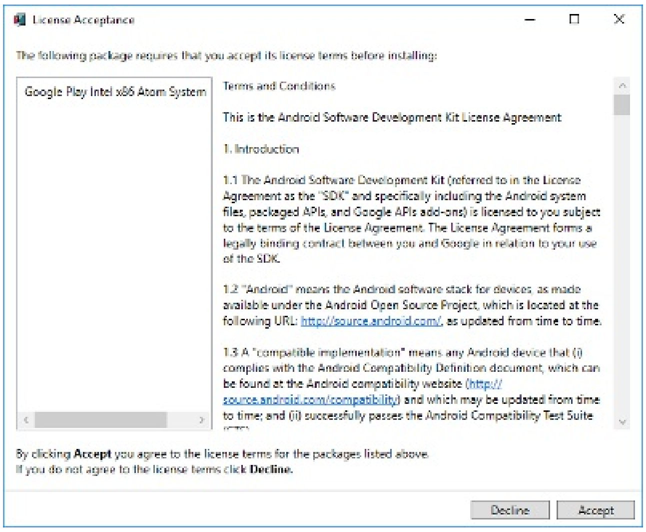


You can also define a custom property by selecting **Custom...** at the top of the property list.

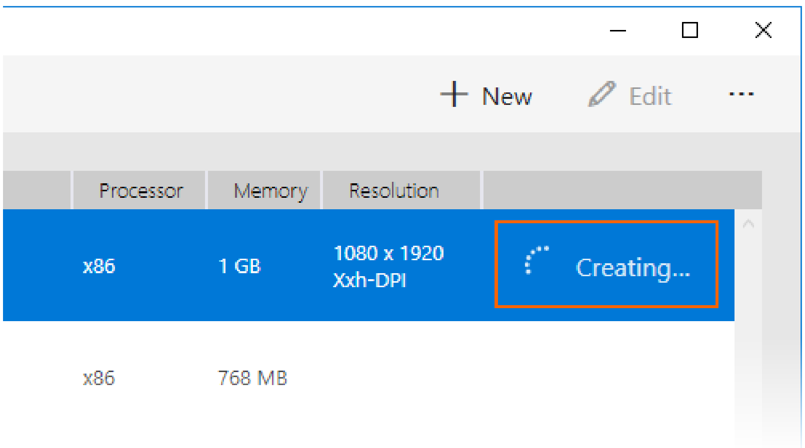
8. Click the **Create** button (lower right-hand corner) to create the new device:



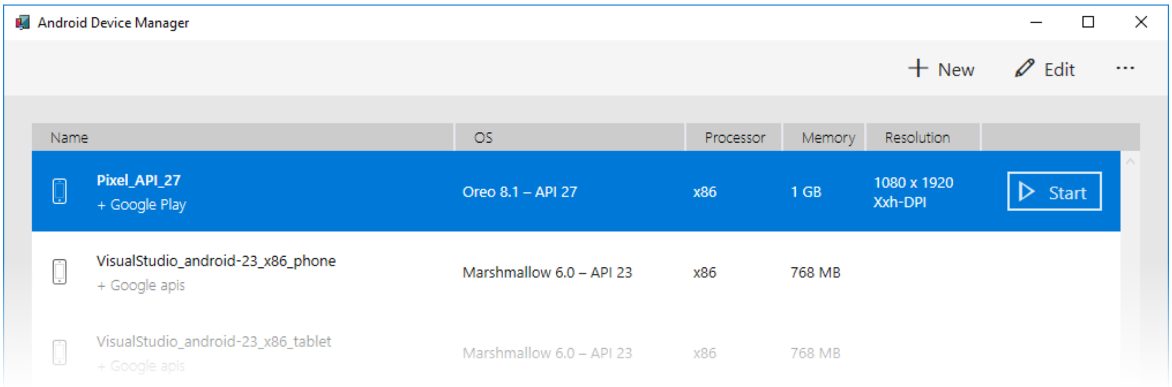
9. You might get a **License Acceptance** screen. Click **Accept** if you agree to the license terms:



10. The Android Device Manager adds the new device to the list of installed virtual devices while displaying a **Creating** progress indicator during device creation:

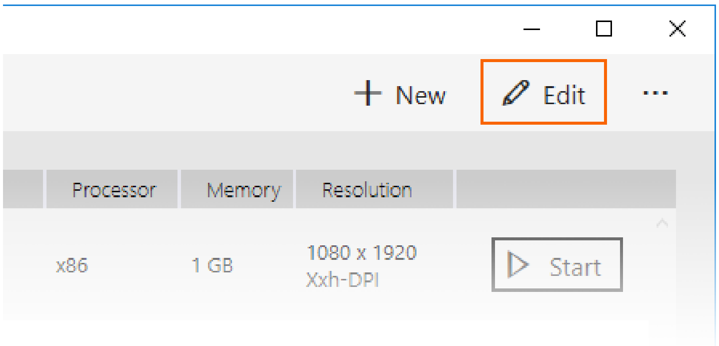


11. When the creation process is complete, the new device is shown in the list of installed virtual devices with a **Start** button, ready to launch:

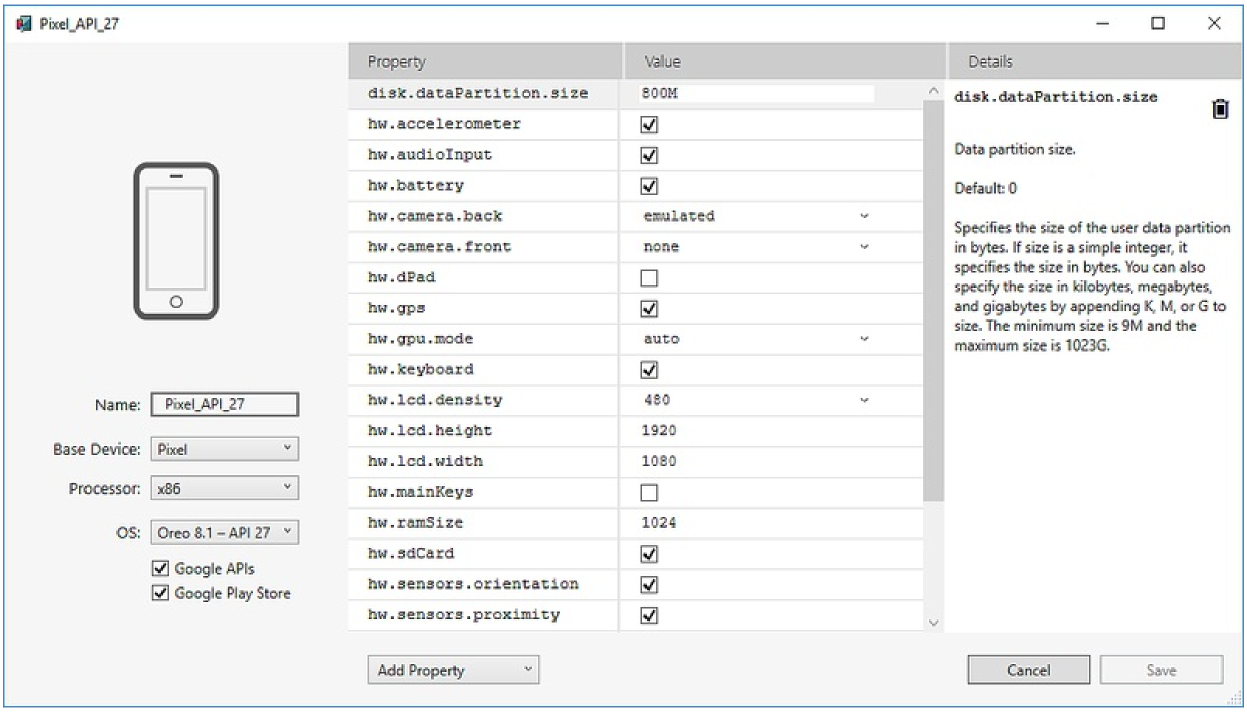


Edit Device

To edit an existing virtual device, select the device and click the **Edit** button (located in the upper right-hand corner of the screen):

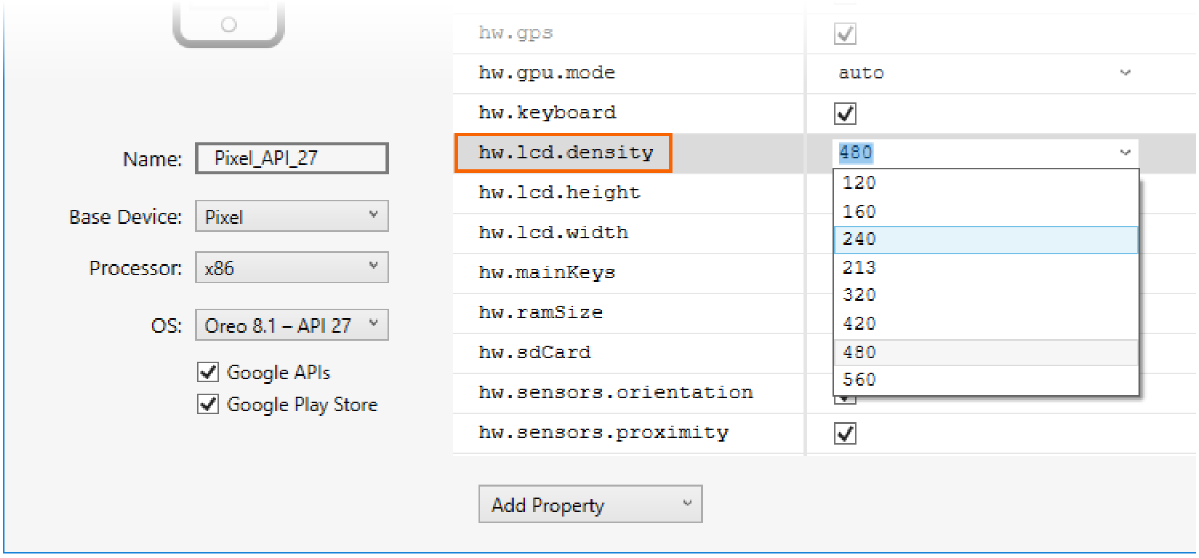


Clicking **Edit** launches the Device Editor for the selected virtual device:



The **Device Editor** screen lists the properties of the virtual device under the **Property** column, with the corresponding values of each property in the **Value** column. When you select a property, a detailed description of that property is displayed on the right.

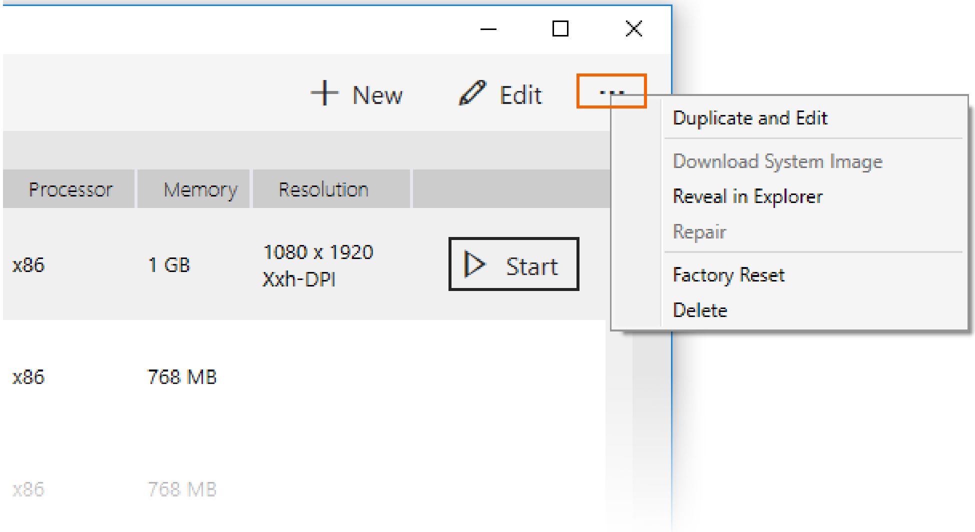
To change a property, edit its value in the **Value** column. For example, in the following screenshot the `hw.lcd.density` property is being changed from **480** to **240**:



After you have made the necessary configuration changes, click the **Save** button. For more information about changing virtual device properties, see Editing Android Virtual Device Properties.

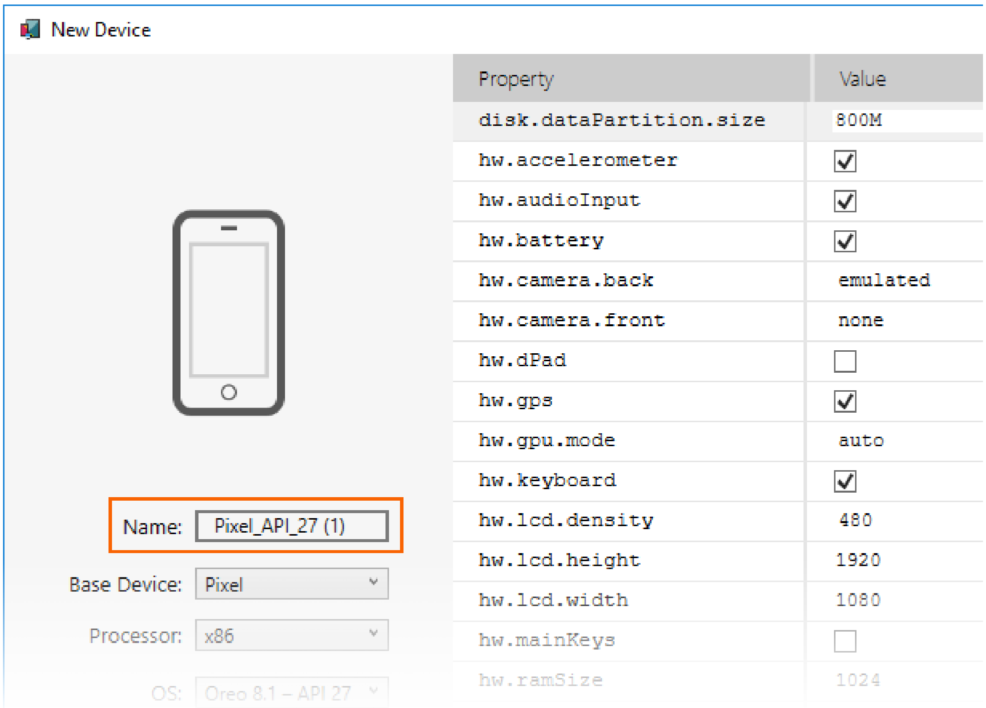
Additional Options

Additional options for working with devices are available from the **Additional Options** (…) pull-down menu in the upper right-hand corner:

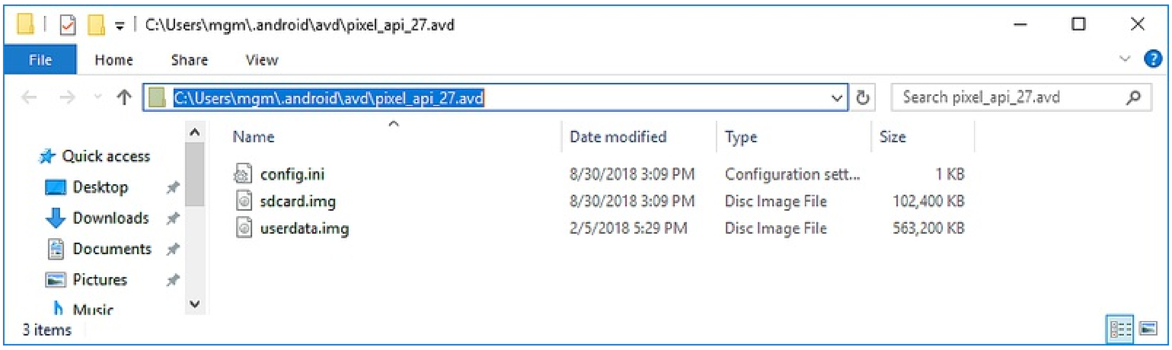


The additional options menu contains the following items:

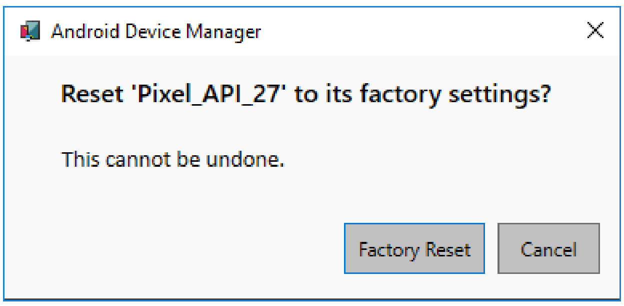
- **Duplicate and Edit** – Duplicates the currently-selected device and opens it in the **New Device** screen with a different unique name. For example, selecting **Pixel_API_27** and clicking **Duplicate and Edit** appends a counter to the name:



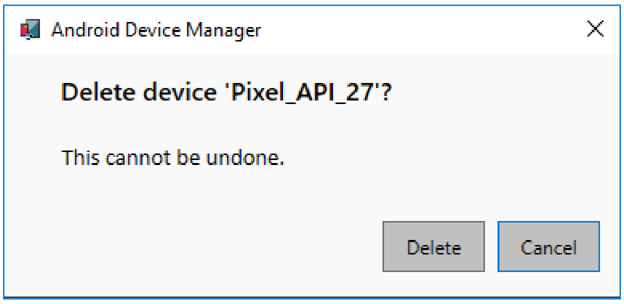
- **Reveal in Explorer** – Opens a Windows Explorer window in the folder that holds the files for the virtual device. For example, selecting **Pixel_API_27** and clicking **Reveal in Explorer** opens a window like the following example:



- **Factory Reset** – Resets the selected device to its default settings, erasing any user changes made to the internal state of the device while it was running (this also erases the current Quick Boot snapshot, if any). This change does not alter modifications that you make to the virtual device during creation and editing. A dialog box will appear with the reminder that this reset cannot be undone. Click **Factory Reset** to confirm the reset:



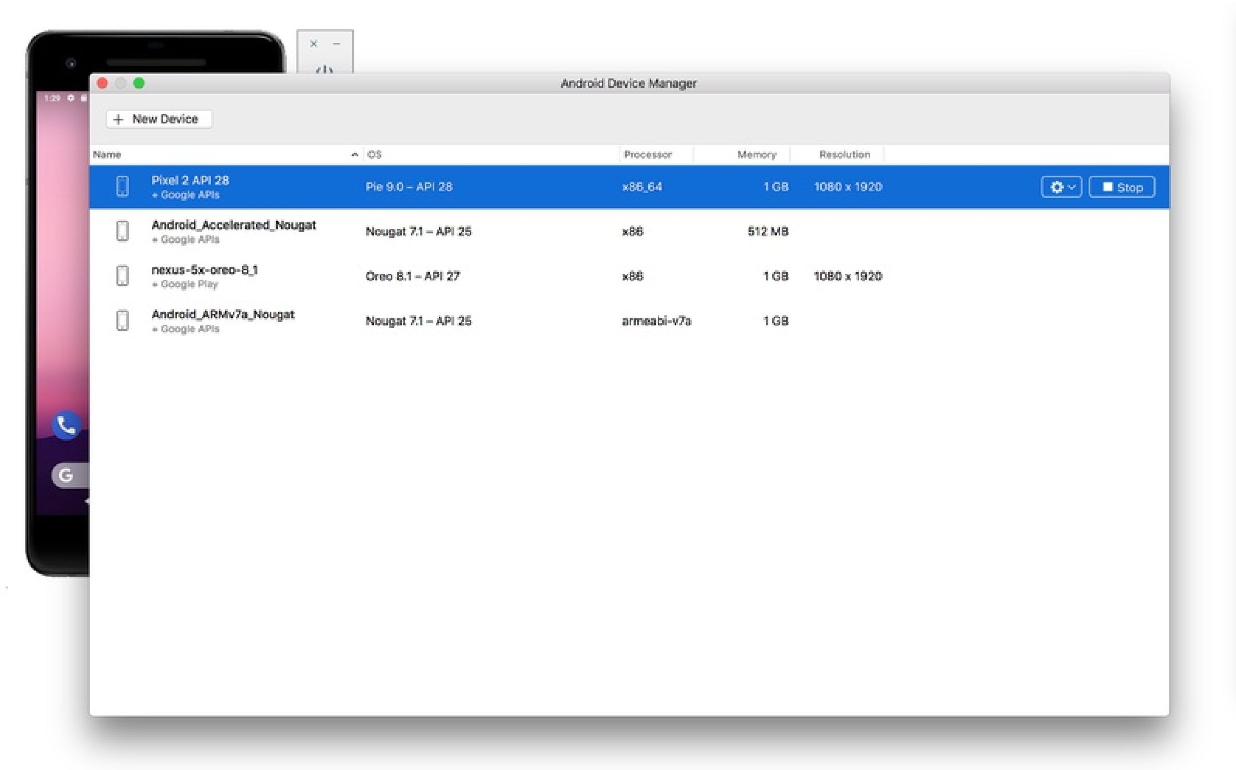
- **Delete** – Permanently deletes the selected virtual device. A dialog box will appear with the reminder that deleting a device cannot be undone. Click **Delete** if you are certain that you want to delete the device.



> **NOTE**
>
> If you are using a Mac with an Apple chip, such as the M1, you will need to install the Android Emulator for M1 preview from GitHub.

Android Device Manager on macOS

This article explains how to use the Android Device Manager to create, duplicate, customize, and launch Android virtual devices.



You use the Android Device Manager to create and configure *Android Virtual Devices* (AVDs) that run in the Android Emulator. Each AVD is an emulator configuration that simulates a physical Android device. This makes it possible to run and test your app in a variety of configurations that simulate different physical Android devices.

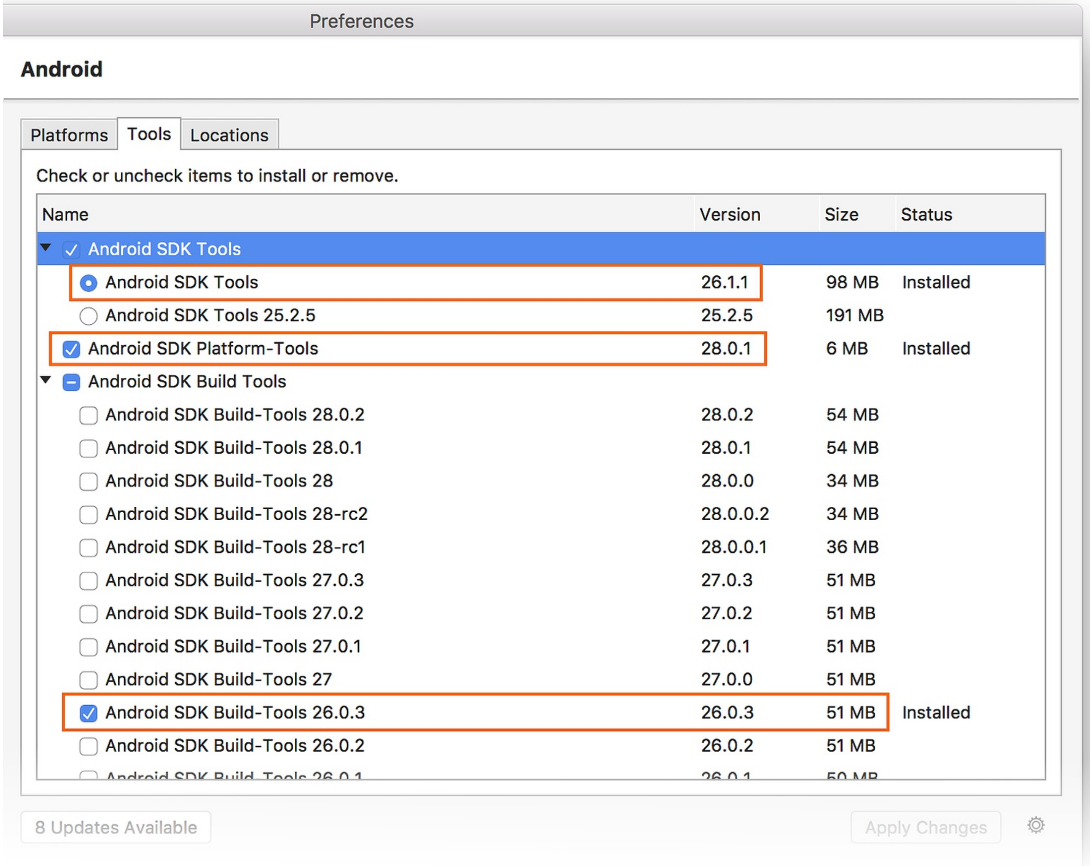
Requirements

To use the Android Device Manager, you will need the following items:

- Visual Studio for Mac 7.6 or later.
- The Android SDK must be installed (see Setting up the Android SDK for Xamarin.Android).
- The following packages must be installed (via the Android SDK Manager):
  - **SDK tools version 26.1.1** or later

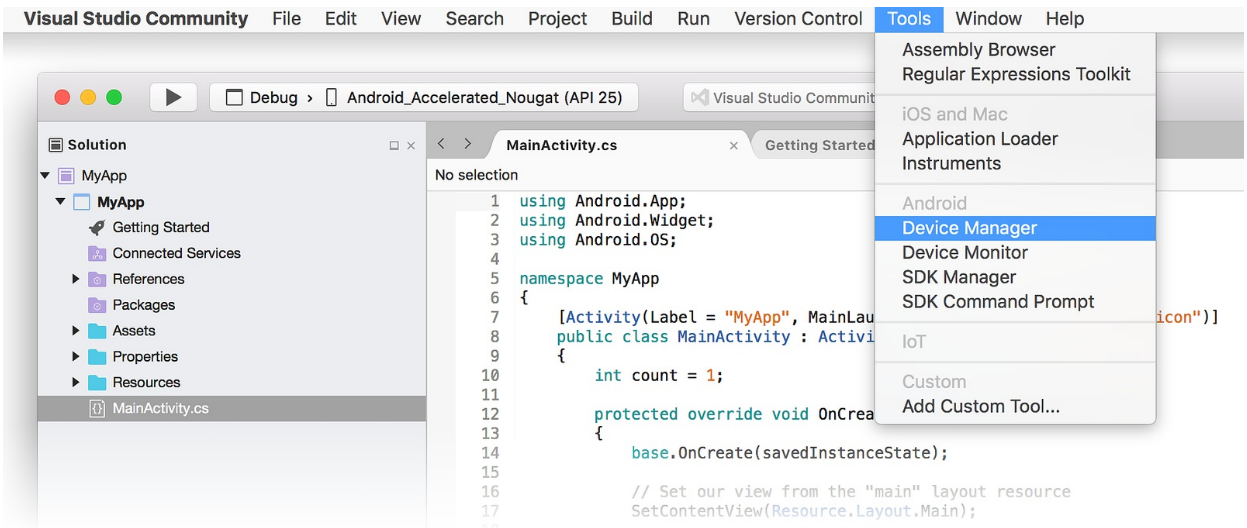
- **Android SDK Platform-Tools 28.0.1** or later
- **Android SDK Build-Tools 26.0.3** or later

These packages should be displayed with **Installed** status as seen in the following screenshot:

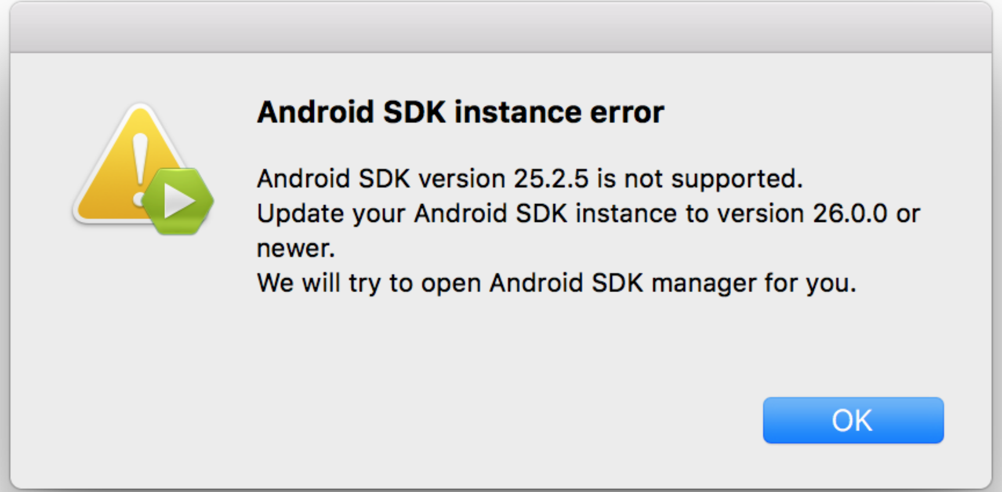


Launching the Device Manager

Launch the Android Device Manager by clicking **Tools > Device Manager**:

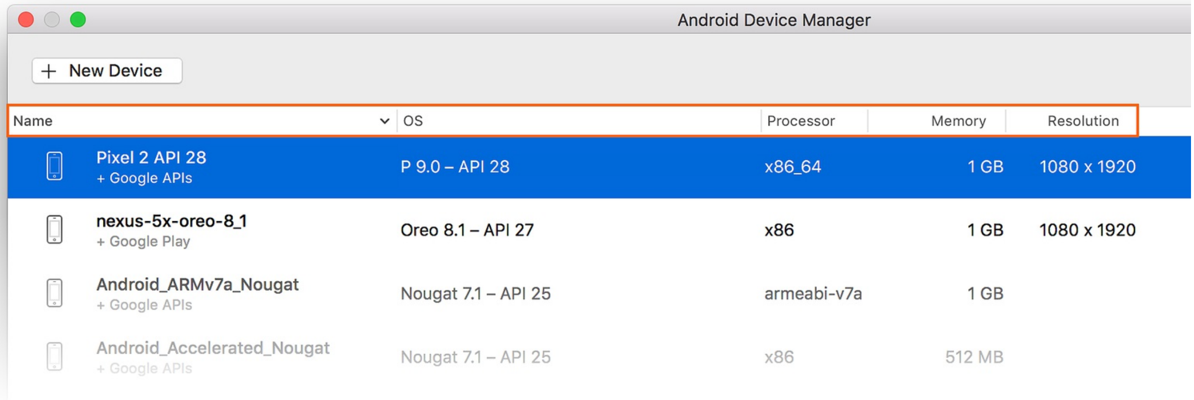


If the following error dialog is presented on launch, see the Troubleshooting section for workaround instructions:

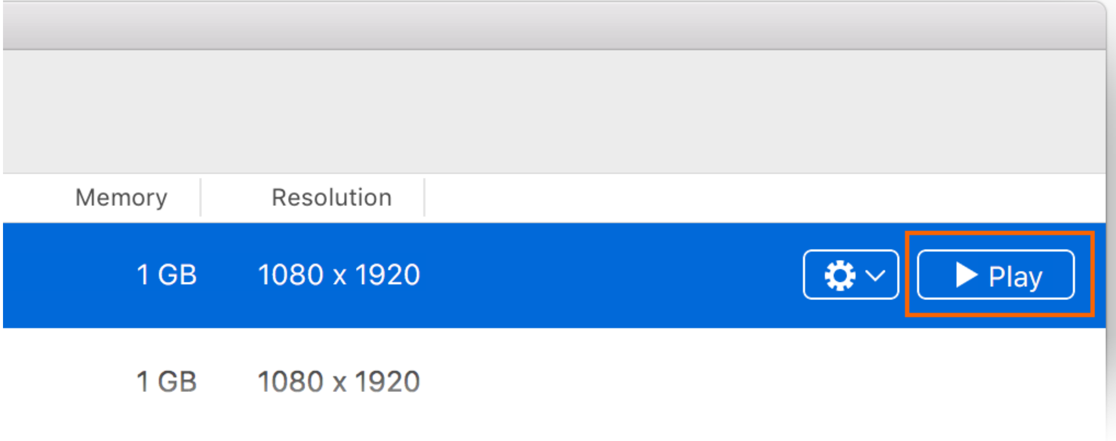


Main Screen

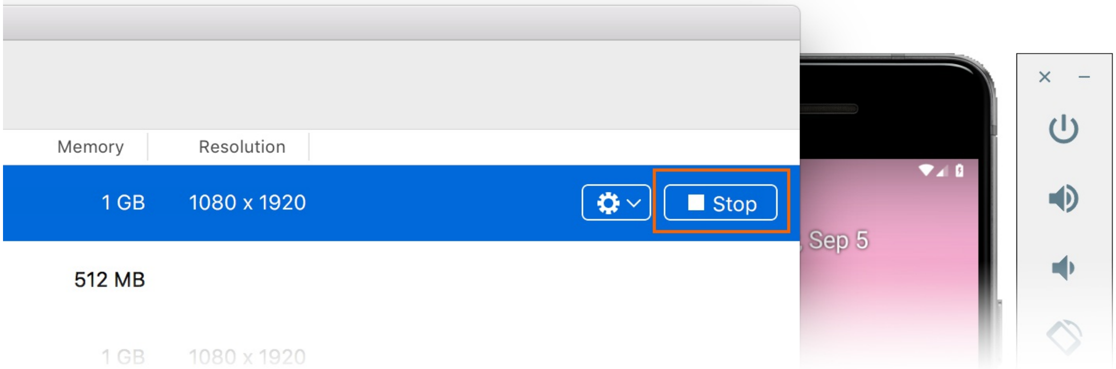
When you first launch the Android Device Manager, it presents a screen that displays all currently-configured virtual devices. For each virtual device, the **Name**, **OS** (Android Version), **Processor**, **Memory** size, and screen **Resolution** are displayed:



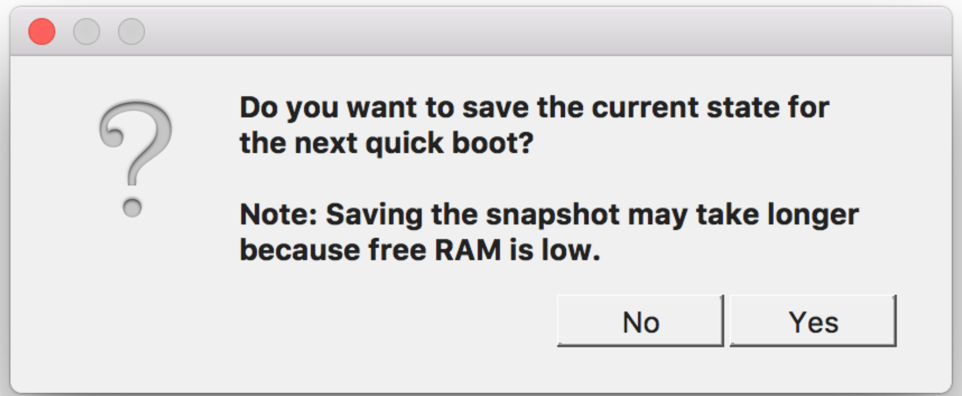
When you select a device in the list, the **Play** button appears on the right. You can click the **Play** button to launch the emulator with this virtual device:



After the emulator starts with the selected virtual device, the **Play** button changes to a **Stop** button that you can use to halt the emulator:



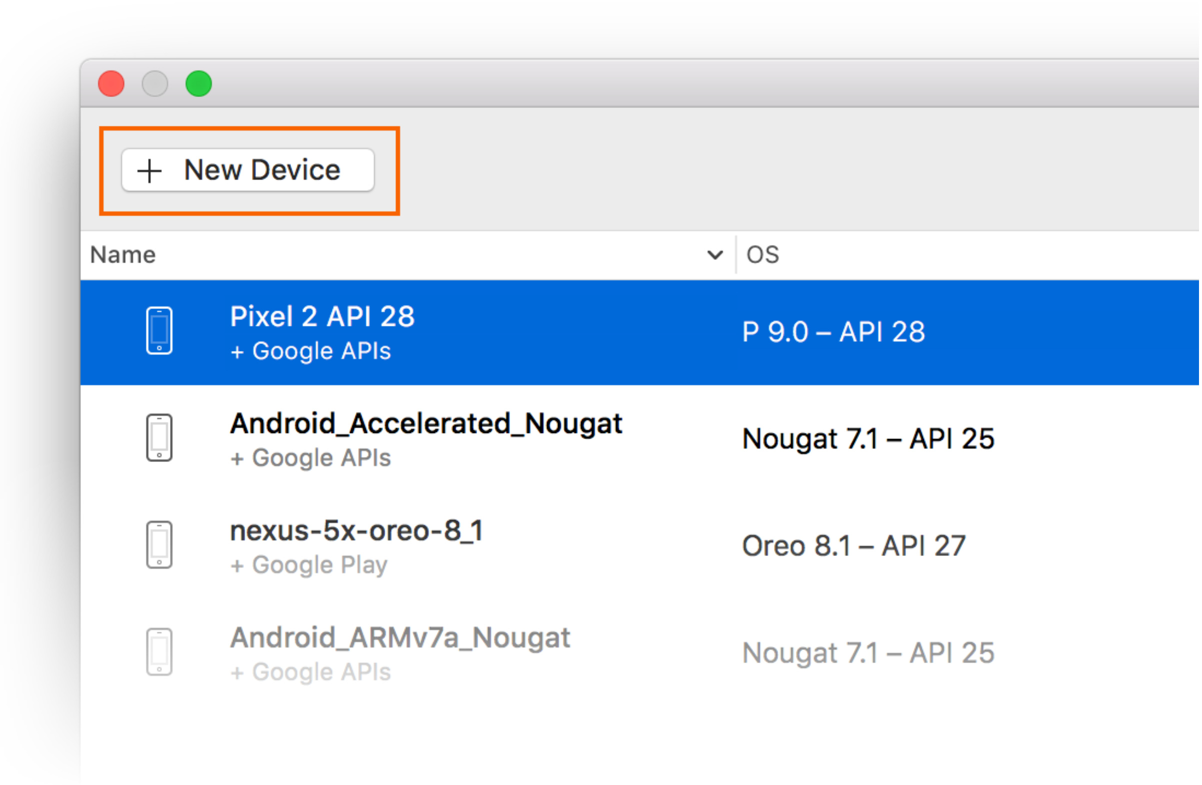
When you stop the emulator, you may get a prompt asking if you want to save the current state for the next quick boot:



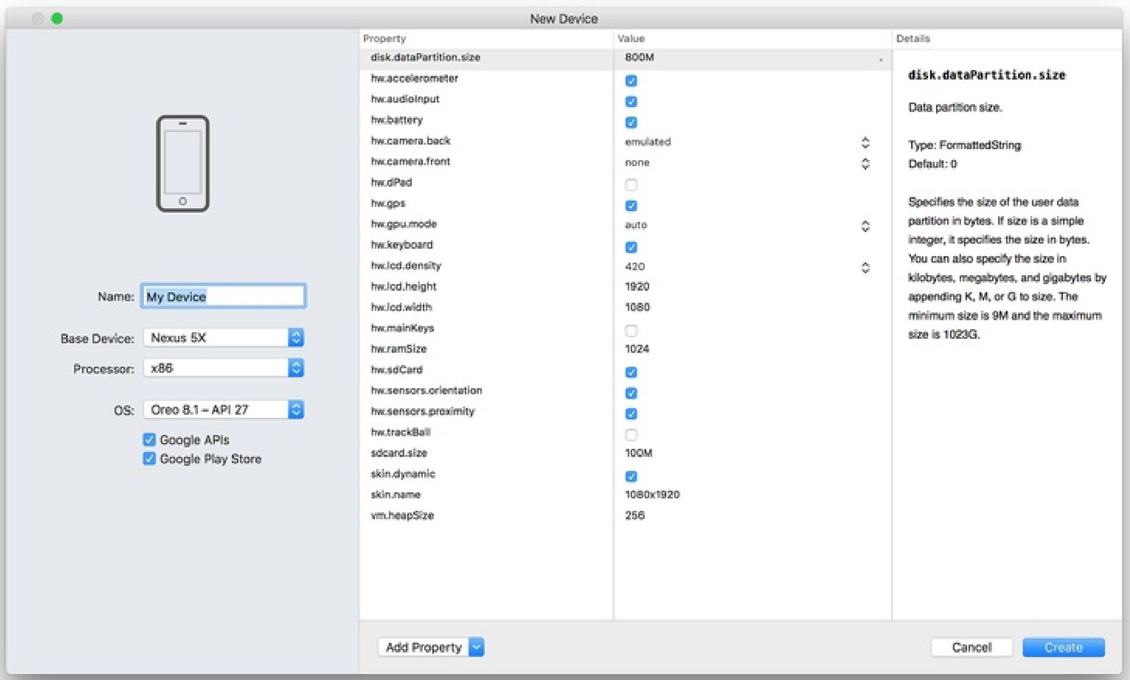
Saving the current state will make the emulator boot faster when this virtual device is launched again. For more information about Quick Boot, see Quick Boot.

New Device

To create a new device, click the **New Device** button (located in the upper left-hand area of the screen):

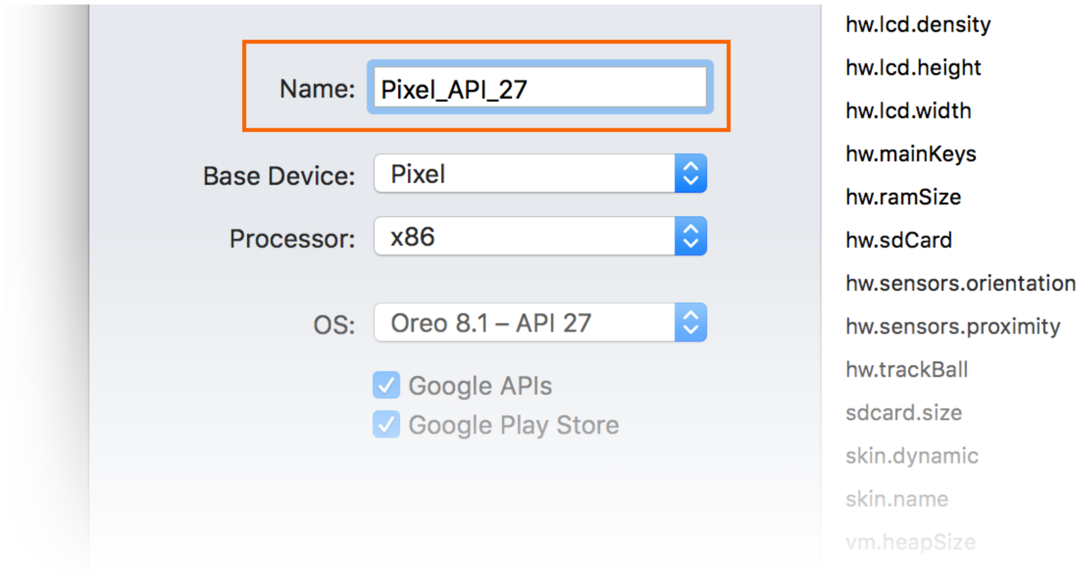


Clicking **New Device** launches the **New Device** screen:



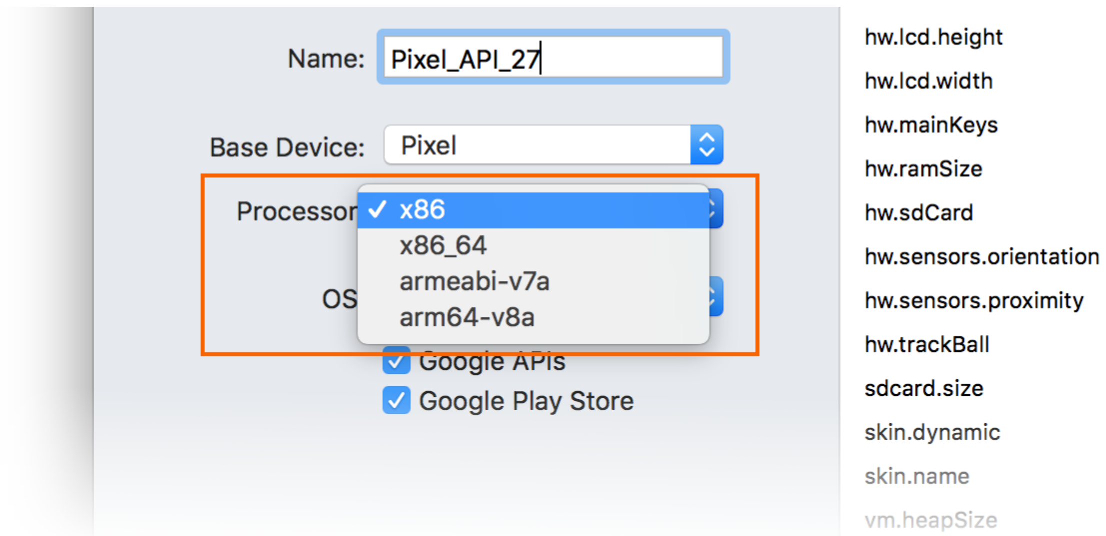
Use the following steps to configure a new device in the **New Device** screen:

1. Give the device a new name. In the following example, the new device is named **Pixel_API_27**:

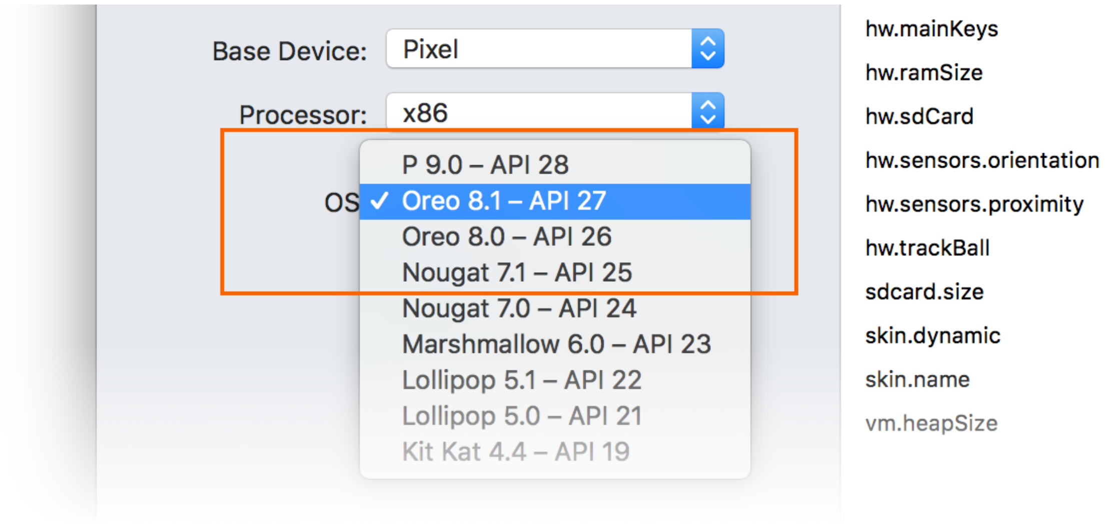


2. Select a physical device to emulate by clicking the **Base Device** pull-down menu:

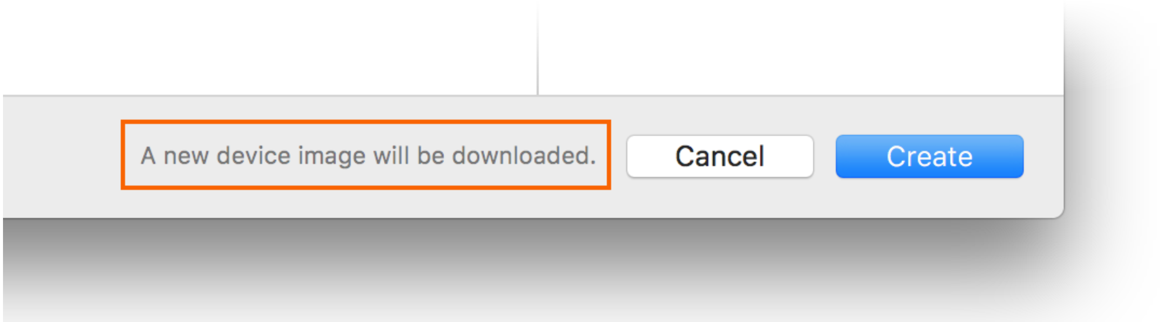
3. Select a processor type for this virtual device by clicking the **Processor** pull-down menu. Selecting **x86** will provide the best performance because it enables the emulator to take advantage of hardware acceleration. The **x86_64** option will also make use of hardware acceleration, but it runs slightly slower than **x86** (**x86_64** is normally used for testing 64-bit apps):



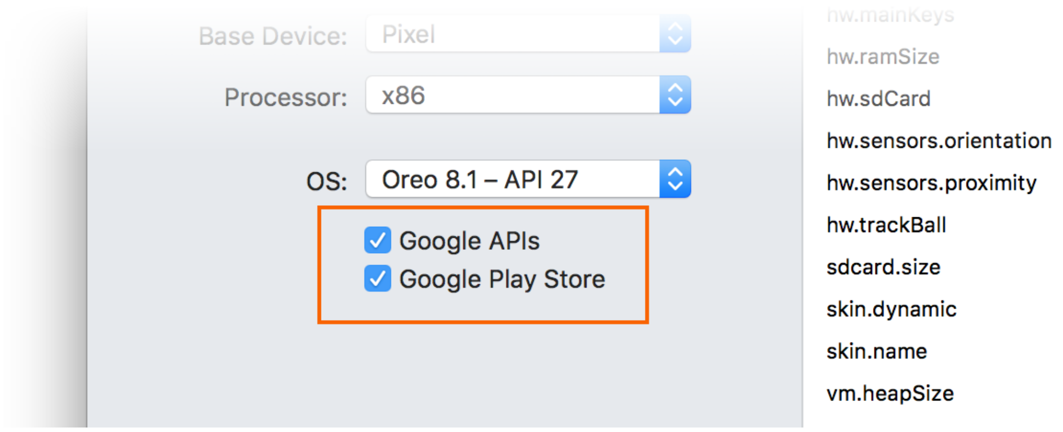
4. Select the Android version (API level) by clicking the **OS** pull-down menu. For example, select **Oreo 8.1 - API 27** to create a virtual device for API level 27:



If you select an Android API level that has not yet been installed, the Device Manager will display **A new device will be downloaded** message at the bottom of the screen – it will download and install the necessary files as it creates the new virtual device:



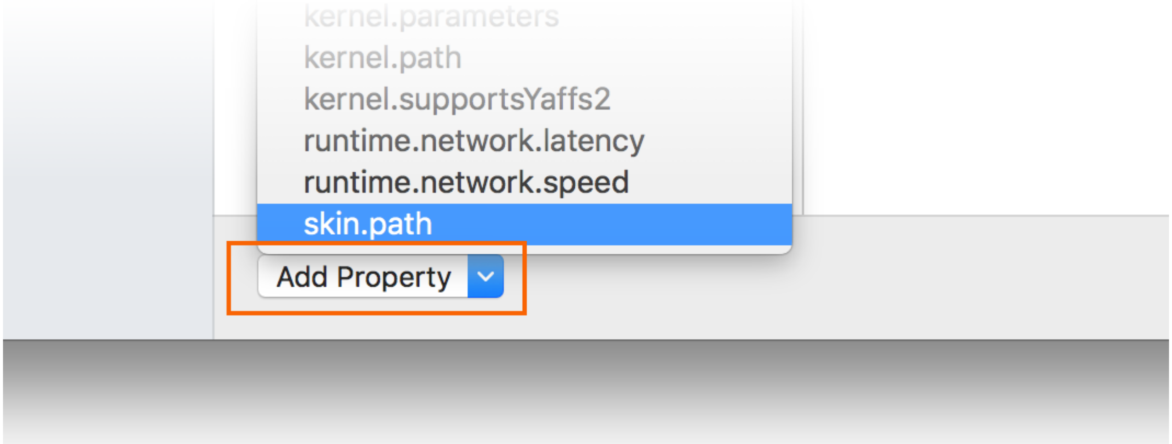
5. If you want to include Google Play Services APIs in your virtual device, enable the **Google APIs** option. To include the Google Play Store app, enable the **Google Play Store** option:



Note that Google Play Store images are available only for some base device types such as Pixel, Pixel 2, Nexus 5, and Nexus 5X.

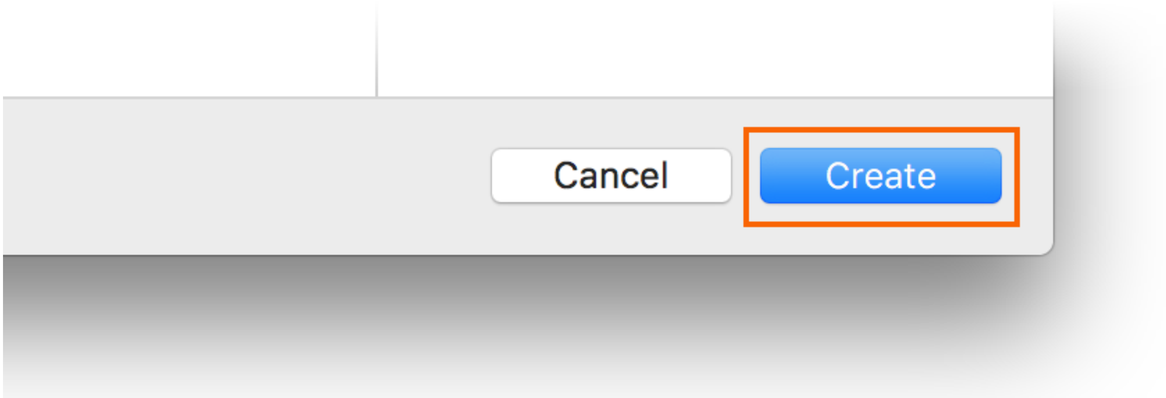
6. Edit any properties that you need to modify. To make changes to properties, see Editing Android Virtual Device Properties.

7. Add any additional properties that you need to explicitly set. The **New Device** screen lists only the most commonly-modified properties, but you can click the **Add Property** pull-down menu (at the bottom) to add additional properties:

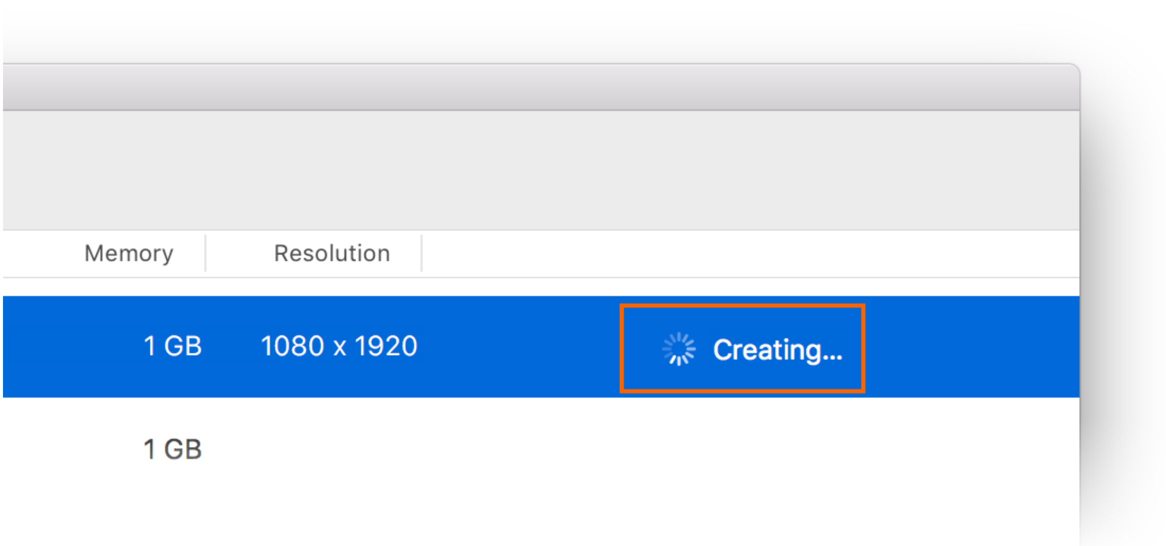


You can also define a custom property by clicking **Custom...** at the top of this property list.

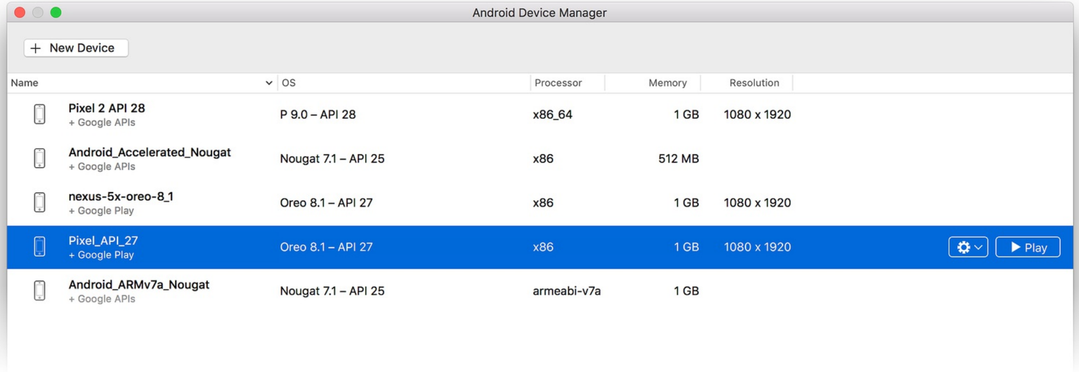
8. Click the **Create** button (lower right-hand corner) to create the new device:



9. The Android Device Manager adds the new device to the list of installed virtual devices while displaying a **Creating** progress indicator during device creation:

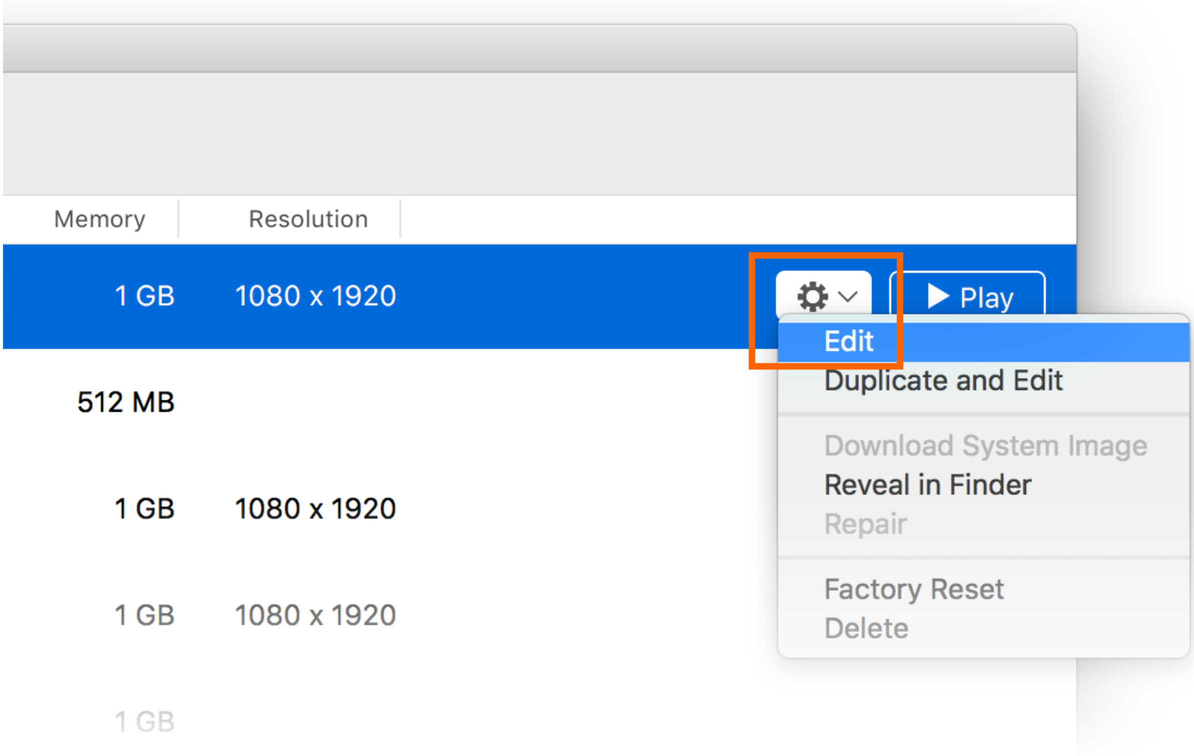


10. When the creation process is complete, the new device is shown in the list of installed virtual devices with a **Start** button, ready to launch:

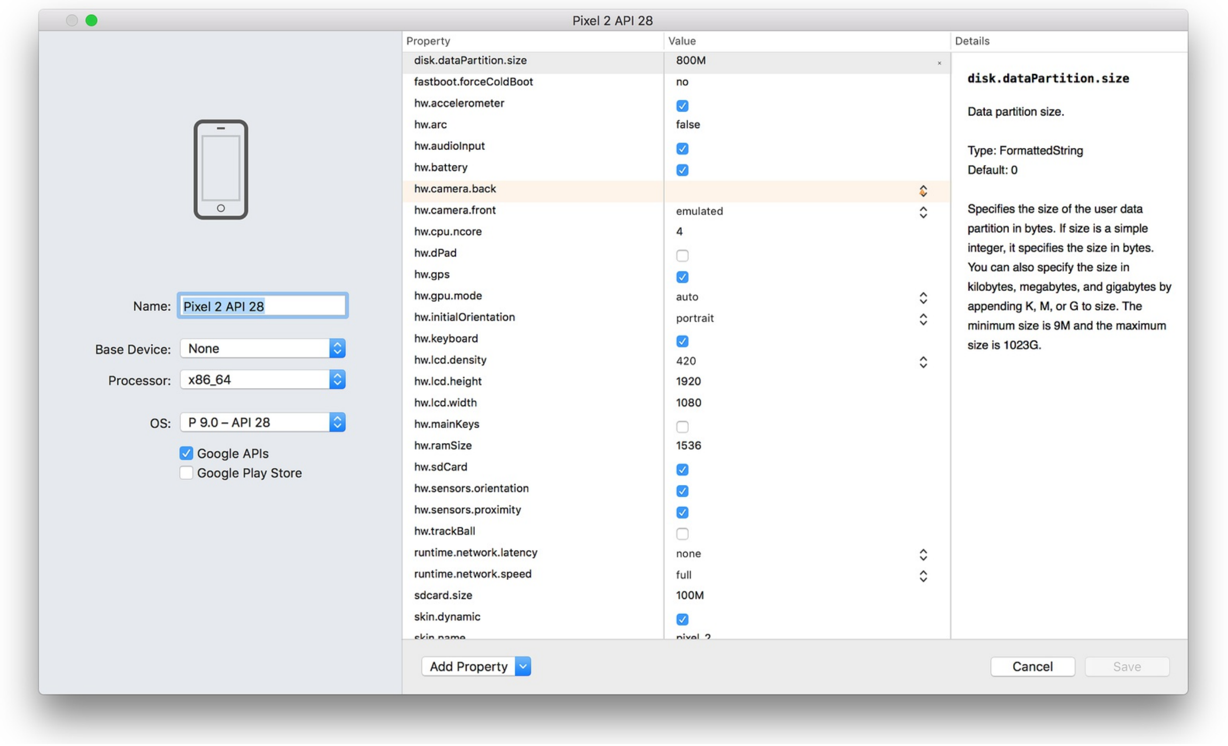


Edit Device

To edit an existing virtual device, select the **Additional Options** pull-down menu (gear icon) and select **Edit**:

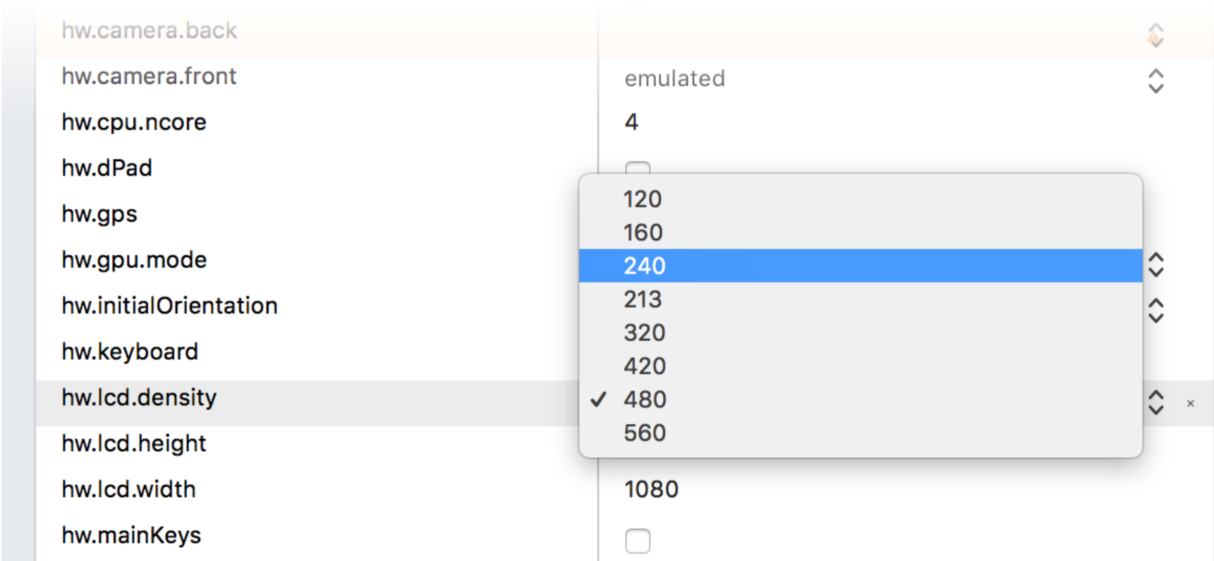


Clicking **Edit** launches the Device Editor for the selected virtual device:



The **Device Editor** screen lists the properties of the virtual device under the **Property** column, with the corresponding values of each property in the **Value** column. When you select a property, a detailed description of that property is displayed on the right.

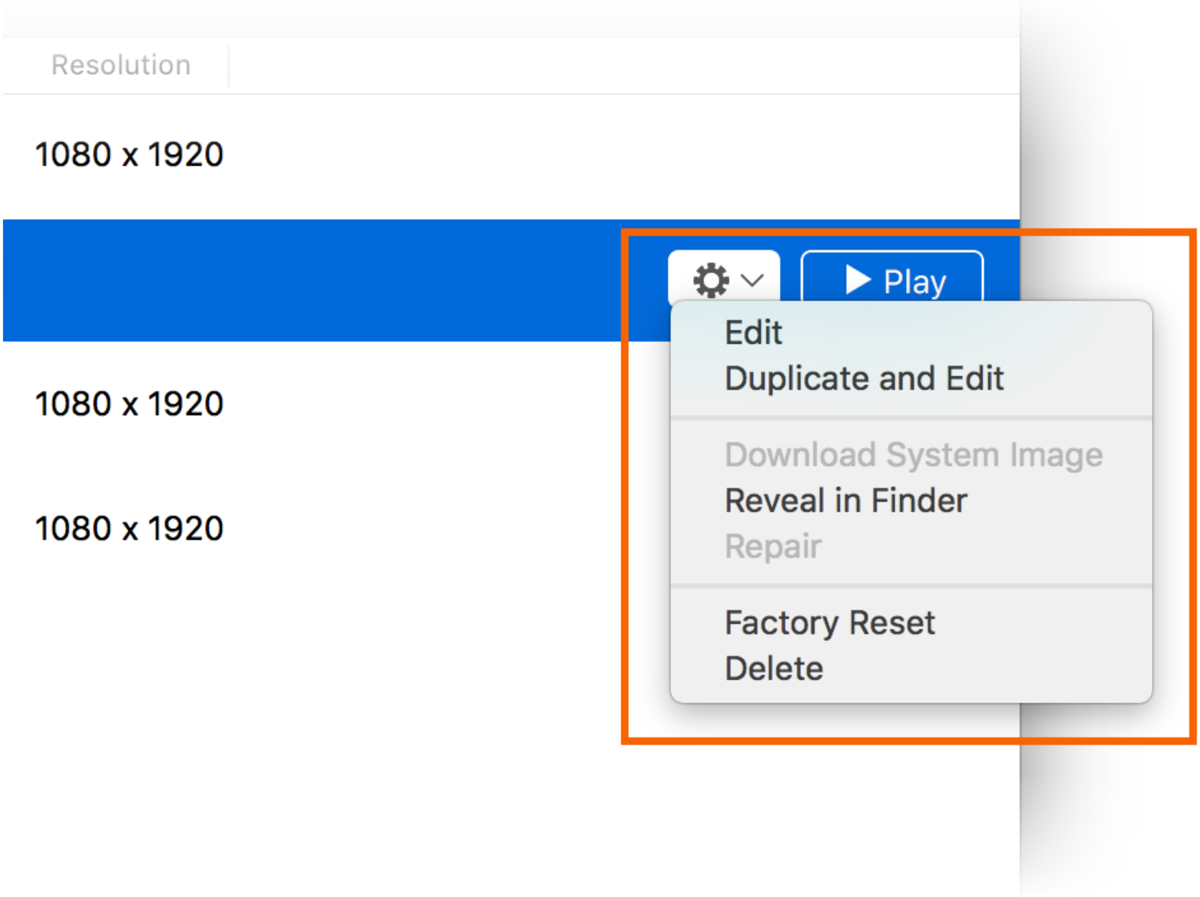
To change a property, edit its value in the **Value** column. For example, in the following screenshot the `hw.lcd.density` property is being changed from **480** to **240**:



After you have made the necessary configuration changes, click the **Save** button. For more information about changing virtual device properties, see Editing Android Virtual Device Properties.

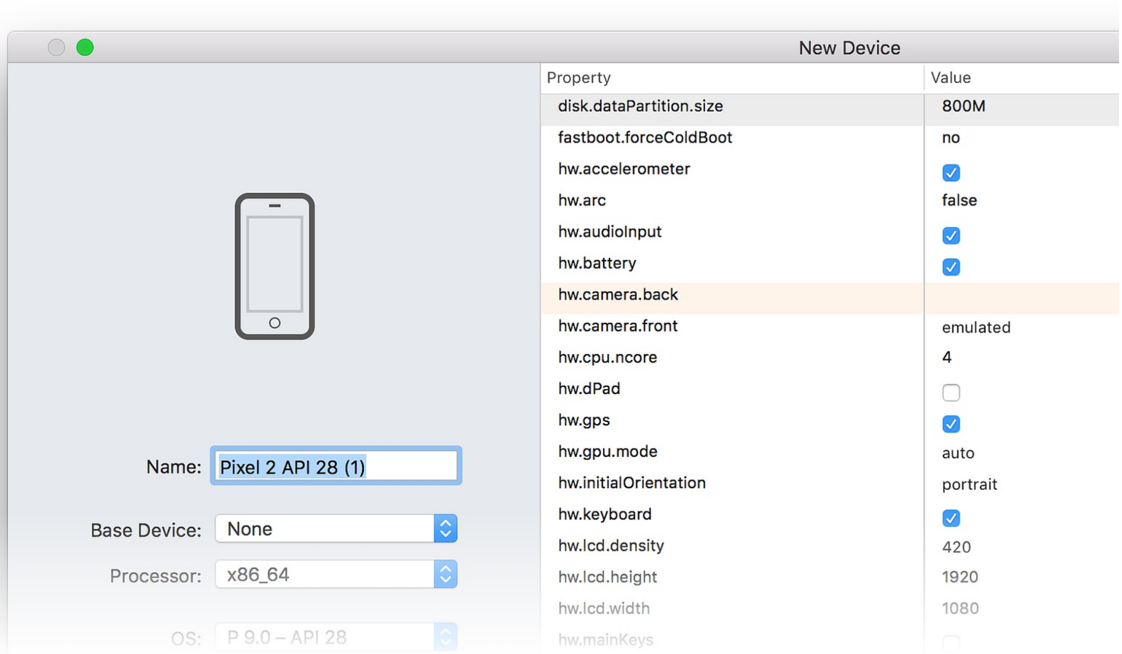
Additional Options

Additional options for working with a device are available from the pull-down menu located to the left of the **Play** button:

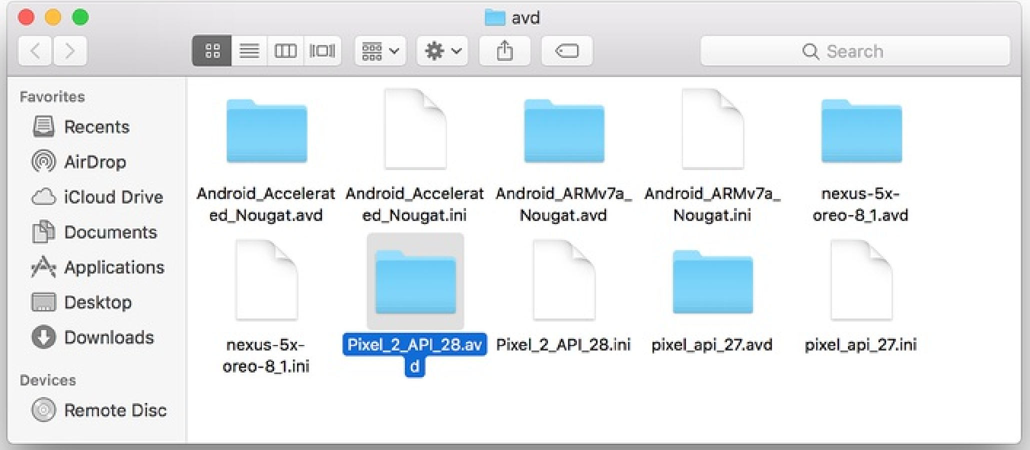


The additional options menu contains the following items:

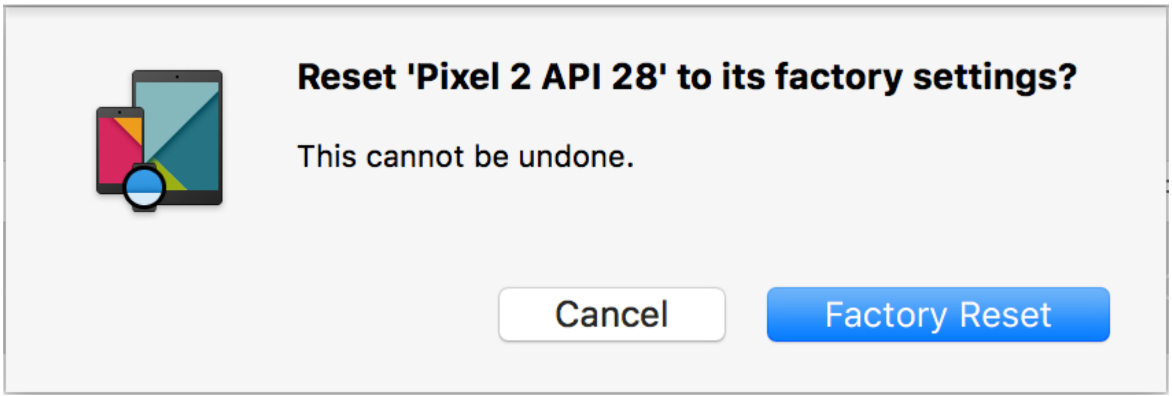
- **Edit** – Opens the currently-selected device in the device editor as described earlier.
- **Duplicate and Edit** – Duplicates the currently-selected device and opens it in the **New Device** screen with a different unique name. For example, selecting **Pixel 2 API 28** and clicking **Duplicate and Edit** appends a counter to the name:



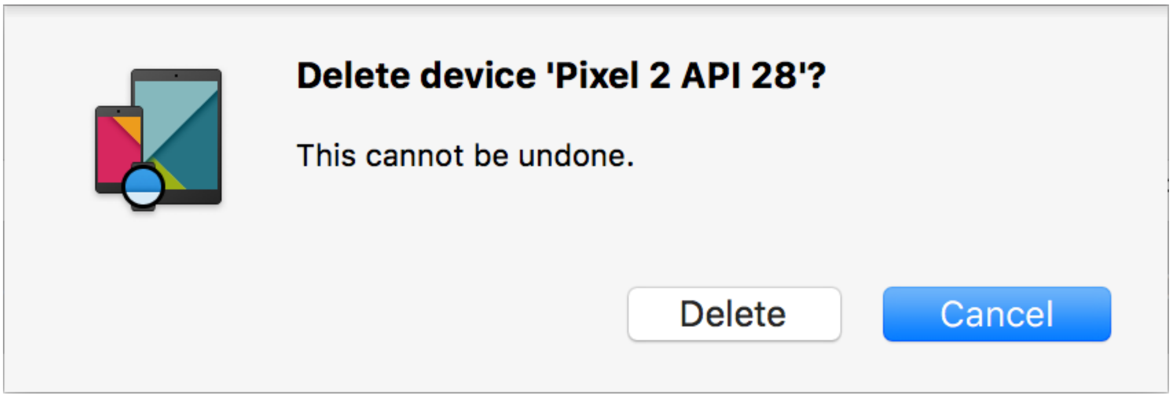
- **Reveal in Finder** – Opens a macOS Finder window in the folder that holds the files for the virtual device. For example, selecting **Pixel 2 API 28** and clicking **Reveal in Finder** opens a window like the following example:



- **Factory Reset** – Resets the selected device to its default settings, erasing any user changes made to the internal state of the device while it was running (this also erases the current Quick Boot snapshot, if any). This change does not alter modifications that you make to the virtual device during creation and editing. A dialog box will appear with the reminder that this reset cannot be undone. Click **Factory Reset** to confirm the reset.



- **Delete** – Permanently deletes the selected virtual device. A dialog box will appear with the reminder that deleting a device cannot be undone. Click **Delete** if you are certain that you want to delete the device.



Troubleshooting

The following sections explain how to diagnose and work around problems that may occur when using the Android Device Manager to configure virtual devices.

- Visual Studio

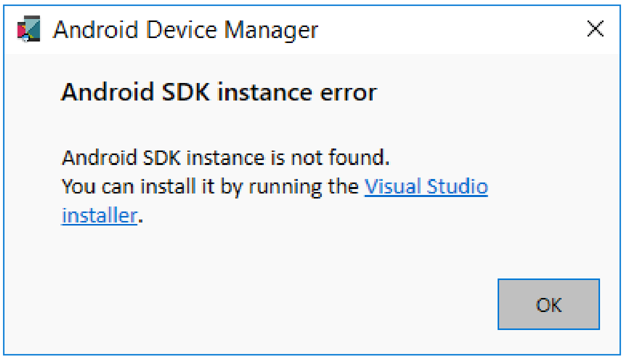
- Visual Studio for Mac

Android SDK in Non-Standard Location

Typically, the Android SDK is installed at the following location:

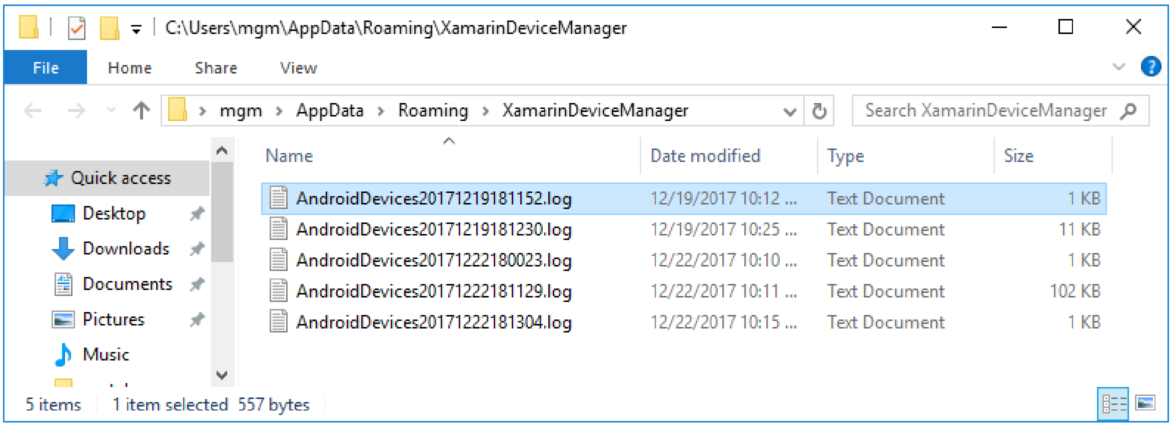
C:\Program Files (x86)\Android\android-sdk

If the SDK is not installed at this location, you may get this error when you launch the Android Device Manager:

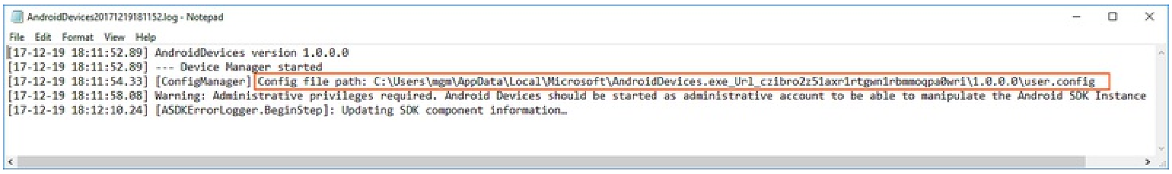


To work around this problem, use the following steps:

1. From the Windows desktop, navigate to **C:\Users*username*\AppData\Roaming\XamarinDeviceManager**:



2. Double-click to open one of the log files and locate the **Config file path**. For example:



3. Navigate to this location and double-click **user.config** to open it.

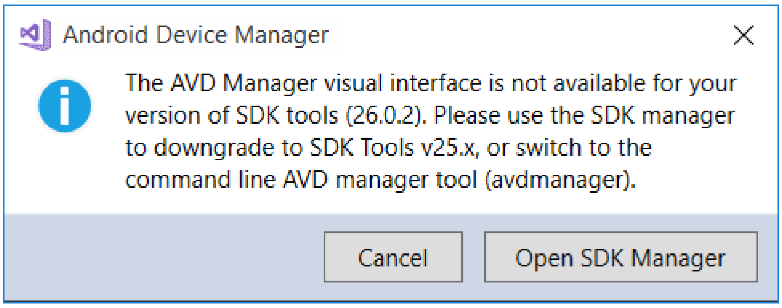
4. In **user.config**, locate the `<UserSettings>` element and add an **AndroidSdkPath** attribute to it. Set this attribute to the path where the Android SDK is installed on your computer and save the file. For example, `<UserSettings>` would look like the following if the Android SDK was installed at **C:\Programs\Android\SDK**:

```
<UserSettings SdkLibLastWriteTimeUtcTicks="636409365200000000" AndroidSdkPath="C:ProgramsAndroidSDK"
/>
```

After making this change to **user.config**, you should be able to launch the Android Device Manager.

Wrong Version of Android SDK Tools

If Android SDK tools 26.1.1 or later is not installed, you may see this error dialog on launch:



If you see this error dialog, click **Open SDK Manager** to open the Android SDK Manager. In the Android SDK Manager, click the **Tools** tab and install the following packages:

- **Android SDK Tools 26.1.1** or later
- **Android SDK Platform-Tools 27.0.1** or later
- **Android SDK Build-Tools 27.0.3** or later

Snapshot disables WiFi on Android Oreo

If you have an AVD configured for Android Oreo with simulated Wi-Fi access, restarting the AVD after a snapshot may cause Wi-Fi access to become disabled.

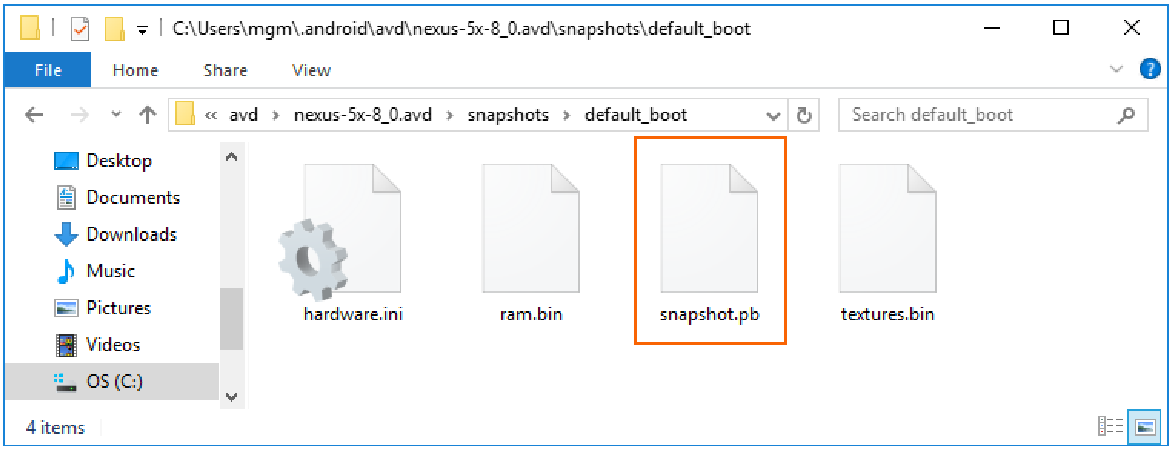
To work around this problem,

1. Select the AVD in the Android Device Manager.

2. From the additional options menu, click **Reveal in Explorer**.

3. Navigate to **snapshots > default_boot**.

4. Delete the **snapshot.pb** file:



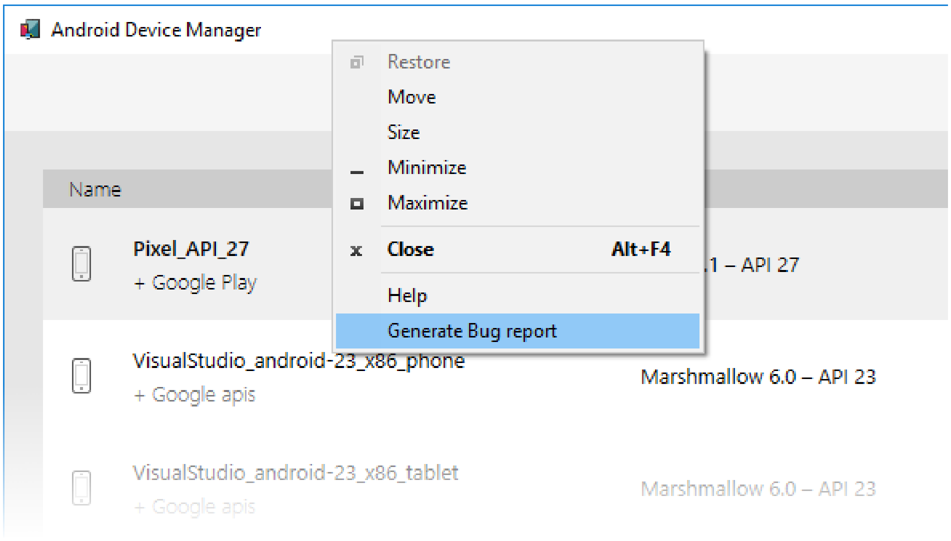
5. Restart the AVD.

After these changes are made, the AVD will restart in a state that allows Wi-Fi to work again.

Generating a Bug Report

- Visual Studio
- Visual Studio for Mac

If you find a problem with the Android Device Manager that cannot be resolved using the above troubleshooting tips, please file a bug report by right-clicking the title bar and selecting **Generate Bug Report**:



Summary

This guide introduced the Android Device Manager available in Visual Studio Tools for Xamarin and Visual Studio for Mac. It explained essential features such as starting and stopping the Android emulator, selecting an Android virtual device (AVD) to run, creating new virtual devices, and how to edit a virtual device. It explained how to edit profile hardware properties for further customization, and it provided troubleshooting tips for common problems.

Related Links

- Changes to the Android SDK Tooling
- Debugging on the Android Emulator
- SDK Tools Release Notes (Google)
- avdmanager
- sdkmanager

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

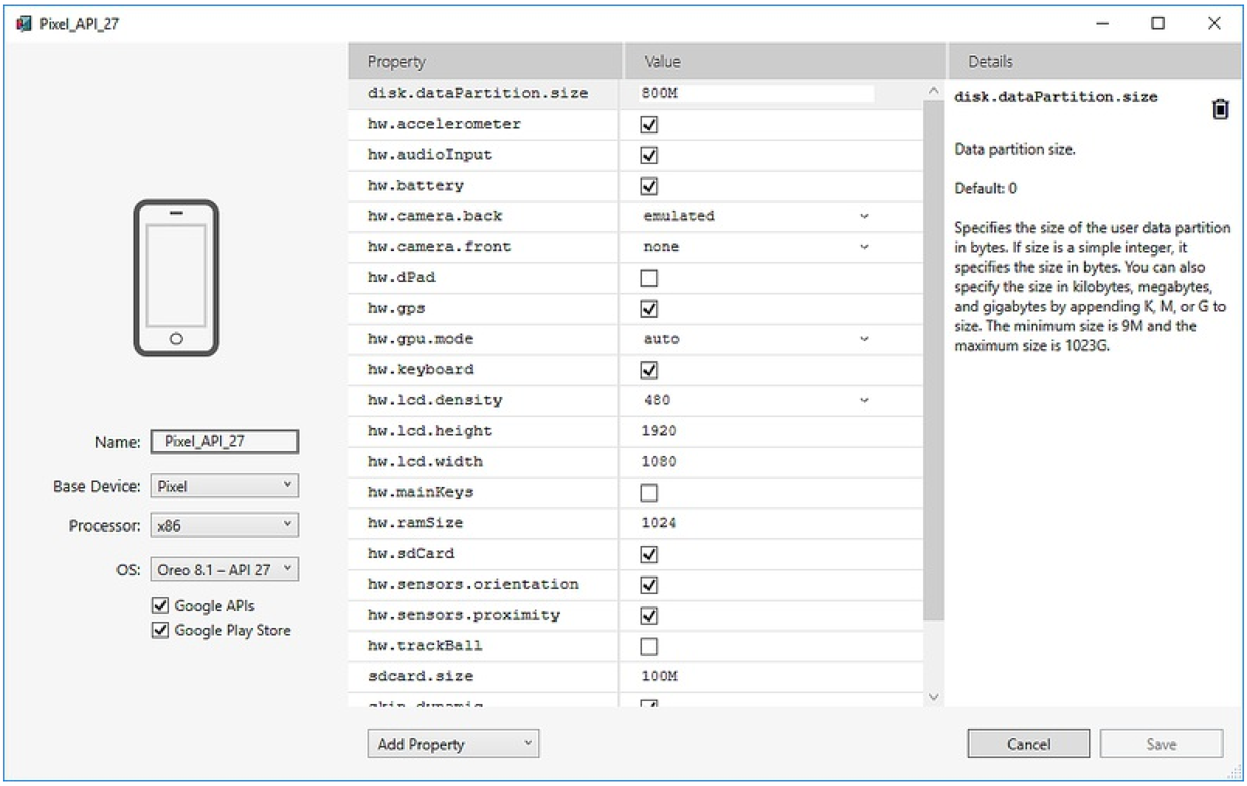
Editing Android Virtual Device Properties

7/8/2021 • 16 minutes to read • Edit Online

This article explains how to use the Android Device Manager to edit the profile properties of an Android virtual device.

Android Device Manager on Windows

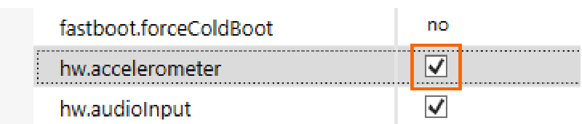
The **Android Device Manager** supports the editing of individual Android virtual device profile properties. The **New Device** and **Device Edit** screens list the properties of the virtual device in the first column, with the corresponding values of each property in the second column (as seen in this example):



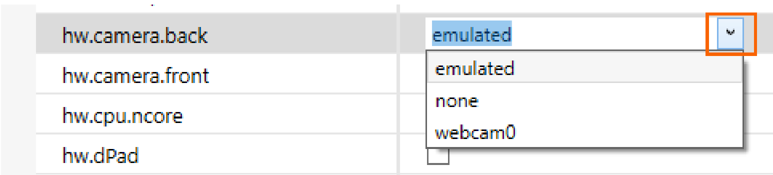
When you select a property, a detailed description of that property is displayed on the right. You can modify *hardware profile properties* and *AVD properties*. Hardware profile properties (such as `hw.ramSize` and `hw.accelerometer`) describe the physical characteristics of the emulated device. These characteristics include screen size, the amount of available RAM, whether or not an accelerometer is present. AVD properties specify the operation of the AVD when it runs. For example, AVD properties can be configured to specify how the AVD uses your development computer's graphics card for rendering.

You can change properties by using the following guidelines:

- To change a boolean property, click the check mark to the right of the boolean property:



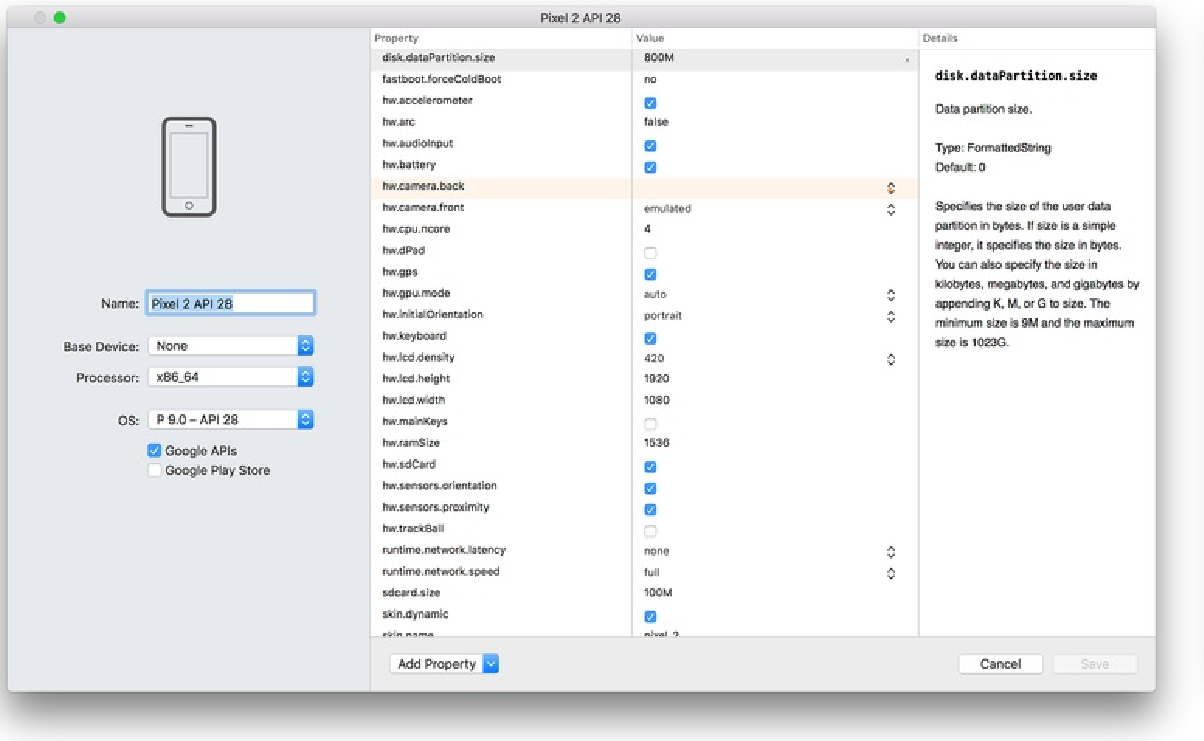
- To change an *enum* (enumerated) property, click the down-arrow to the right of the property and choose a new value.



- To change a string or integer property, double-click the current string or integer setting in the value column and enter a new value.

Android Device Manager on macOS

The **Android Device Manager** supports the editing of individual Android virtual device profile properties. The **New Device** and **Device Edit** screens list the properties of the virtual device in the first column, with the corresponding values of each property in the second column (as seen in this example):



When you select a property, a detailed description of that property is displayed on the right. You can modify *hardware profile properties* and *AVD properties*. Hardware profile properties (such as `hw.ramSize` and `hw.accelerometer`) describe the physical characteristics of the emulated device. These characteristics include screen size, the amount of available RAM, whether or not an accelerometer is present. AVD properties specify the operation of the AVD when it runs. For example, AVD properties can be configured to specify how the AVD uses your development computer's graphics card for rendering.

You can change properties by using the following guidelines:

- To change a boolean property, click the check mark to the right of the boolean property:

- To change an *enum* (enumerated) property, click the pull-down menu to the right of the property and choose a new value.

- To change a string or integer property, double-click the current string or integer setting in the value column and enter a new value.

The following table provides a detailed explanation of the properties listed in the **New Device** and **Device Editor** screens:

PROPERTY	DESCRIPTION	OPTIONS
`abi.type`	**ABI type** – Specifies the ABI (application binary interface) type of the emulated device. The **x86** option is for the instruction set commonly referred to as "x86" or "IA-32." The **x86_64** option is for the 64-bit x86 instruction set. The **armeabi-v7a** option is for the ARM instruction set with v7-a ARM extensions. The **arm64-v8a** option is for the ARM instruction set that supports AArch64.	x86, x86_64, armeabi-v7a, arm64-v8a
`disk.cachePartition`	**Cache partition** – Determines whether the emulated device will use a **/cache** partition on the device. The **/cache** partition (which is initially empty) is the location where Android stores frequently accessed data and app components. If set to **no**, the emulator will not use a **/cache** partition and the other `disk.cache` settings will be ignored.	yes, no
`disk.cachePartition.path`	**Cache partition path** – Specifies a cache partition image file on your development computer. The emulator will use this file for the **/cache** partition. Enter an absolute path or a path relative to the emulator's data directory. If not set, the emulator creates an empty temporary file called **cache.img** on your development computer. If the file does not exist, it is created as an empty file. This option is ignored if `disk.cachePartition` is set to **no**.	

PROPERTY	DESCRIPTION	OPTIONS
`disk.cachePartition.size`	**Cache partition size** – The size of the cache partition file (in bytes). Normally you do not need to set this option unless the app will be downloading very large files that are larger than the default cache size of 66 megabytes. This option is ignored if `disk.cachePartition` is set to **no**. If this value is an integer, it specifies the size in bytes. You can also specify the size in kilobytes, megabytes, and gigabytes by appending **K**, **M**, or **G** to the value. The minimum size is **9M** and the maximum size is **1023G**.	
`disk.dataPartition.initPath`	**Initial path to the data partition** – Specifies the initial contents of the data partition. After wiping user data, the emulator copies the contents of the specified file to user data (by default, **userdata-qemu.img**) instead of using **userdata.img** as the initial version.	
`disk.dataPartition.path`	**Path to the data partition** – Specifies the user data partition file. To configure a persistent user data file, enter a filename and a path on your development computer. If the file doesn't exist, the emulator creates an image from the default file **userdata.img**, stores it in the filename specified by `disk.dataPartition.path`, and persists user data to it when the emulator shuts down. If you don't specify a path, the default file is named **userdata-qemu.img**. The special value **<temp>** causes the emulator to create and use a temporary file. If `disk.dataPartition.initPath` is set, its content will be copied to the `disk.dataPartition.path` file at boot-time. Note that this option cannot be left blank.	
`disk.dataPartition.size`	**Data partition size** – Specifies the size of the user data partition in bytes. If this value is an integer, it specifies the size in bytes. You can also specify the size in kilobytes, megabytes, and gigabytes by appending **K**, **M**, or **G** to the value. The minimum size is **9M** and the maximum size is **1023G**.	

PROPERTY	DESCRIPTION	OPTIONS
`disk.ramdisk.path`	**Ramdisk path** – Path to the boot partition (ramdisk) image. The ramdisk image is a subset of the system image that is loaded by the kernel before the system image is mounted. The ramdisk image typically contains boot-time binaries and initialization scripts. If this option is not specified, the default is **ramdisk.img** in the emulator system directory.	
`disk.snapStorage.path`	**Snapshot storage path** – Path to the snapshot storage file where all snapshots are stored. All snapshots made during execution will be saved to this file. Only snapshots that are saved to this file can be restored during the emulator run. If this option is not specified, the default is snapshots.img in the emulator data directory.	
`disk.systemPartition.initPath`	**System partition init path** – Path to the read-only copy of the system image file; specifically, the partition containing the system libraries and data corresponding to the API level and any variant. If this path is not specified, the default is system.img in the emulator system directory.	
`disk.systemPartition.path`	**System partition path** – Path to the read/write system partition image. If this path is not set, a temporary file will be created and initialized from the contents of the file specified by `disk.systemPartition.initPath`.	
`disk.systemPartition.size`	**System partition size** – The ideal size of the system partition (in bytes). The size is ignored if the actual system partition image is larger than this setting; otherwise, it specifies the maximum size that the system partition file can grow to. If this value is an integer, it specifies the size in bytes. You can also specify the size in kilobytes, megabytes, and gigabytes by appending **K**, **M**, or **G** to the value. The minimum size is **9M** and the maximum size is **1023G**.	
`hw.accelerometer`	**Accelerometer** – Determines whether the emulated device contains an accelerometer sensor. The accelerometer helps the device determine orientation (used for auto-rotation). The accelerometer reports the acceleration of the device along three sensor axes.	yes, no

PROPERTY	DESCRIPTION	OPTIONS
`hw.audioInput`	**Audio recording support** – Determines whether the emulated device can record audio.	yes, no
`hw.audioOutput`	**Audio playback support** – Determines whether the emulated device can play audio.	yes, no
`hw.battery`	**Battery support** – Determines whether the emulated device can run on a battery.	yes, no
`hw.camera`	**Camera support** – Determines whether the emulated device has a camera.	yes, no
`hw.camera.back`	**Back-facing camera** – Configures the back-facing camera (the lens faces away from the user). If you are using a webcam on your development computer to simulate the back-facing camera on the emulated device, this value must be set to webcam*n*, where *n* selects the webcam (if you have only one webcam, choose **webcam0**). If set to emulated, the emulator simulates the camera in software. To disable the back-facing camera, set this value to none. If you enable the back-facing camera, be sure to also enable `hw.camera`.	emulated, none, webcam0
`hw.camera.front`	**Front-facing camera** – Configures the front-facing camera (the lens faces towards the user). If you are using a webcam on your development computer to simulate the front-facing camera on the emulated device, this value must be set to webcam*n*, where *n* selects the webcam (if you have only one webcam, choose **webcam0**). If set to emulated, the emulator simulates a camera in software. To disable the front-facing camera, set this value to none. If you enable the front-facing camera, be sure to also enable `hw.camera`.	emulated, none, webcam0
`hw.camera.maxHorizontalPixels`	**Maximum horizontal camera pixels** – Configures the maximum horizontal resolution of the emulated device's camera (in pixels).	
`hw.camera.maxVerticalPixels`	**Maximum vertical camera pixels** – Configures the maximum vertical resolution of the emulated device's camera (in pixels).	

PROPERTY	DESCRIPTION	OPTIONS
`hw.cpu.arch`	**CPU architecture** – The CPU architecture to be emulated by the virtual device. If you are using Intel HAXM for hardware acceleration, select **x86** for a 32-bit CPU. Select **x86_64** for a 64-bit HAXM-accelerated device. (Be sure to install the corresponding Intel x86 system image in the SDK Manager: for example, Intel x86 Atom or Intel x86 Atom_64.) To simulate an ARM CPU, select **arm** for 32-bit or select **arm64** for a 64-bit ARM CPU. Keep in mind that ARM-based virtual devices will run much slower than those that are x86-based because hardware acceleration is not available for ARM.	x86, x86_64, arm, arm64
`hw.cpu.model`	**CPU model** – This value is normally left unset (it will be set to a value that is derived from `hw.cpu.arch` if it is not explicitly set). However, it can be set to an emulator-specific string for experimental use.	
`hw.dPad`	**DPad keys** – Determines whether the emulated device supports directional pad (DPad) keys. A DPad typically has four keys to indicate directional control.	yes, no
`hw.gps`	**GPS support** – Determines whether the emulated device has a GPS (Global Positioning System) receiver.	yes, no
`hw.gpu.enabled`	**GPU emulation** – Determines whether the emulated device supports GPU emulation. When enabled, GPU emulation uses Open GL for Embedded Systems (OpenGL ES) for rendering both 2D and 3D graphics on the screen, and the associated GPU Emulation Mode setting determines how the GPU emulation is implemented.	yes, no

PROPERTY	DESCRIPTION	OPTIONS
`hw.gpu.mode`	**GPU emulation mode** – Determines how GPU emulation is implemented by the emulator. If you select auto, the emulator will choose hardware and software acceleration based on your development computer setup. If you select host, the emulator will use your development computer's graphics processor to perform GPU emulation for faster rendering. If your GPU is not compatible with the emulator and you are on Windows, you can try angle instead of host. The angle mode uses DirectX to provide performance similar to host. If you select mesa, the emulator will use the Mesa 3D software library to render graphics. Select mesa if you have problems rendering via your development computer's graphics processor. The swiftshader mode can be used to render graphics in software with slightly less performance than using your computer's GPU. The off option (disable graphics hardware emulation) is a deprecated option that can cause improper rendering for some items and is therefore not recommended.	auto, host, mesa, angle, swiftshader, off
`hw.gsmModem`	**GSM modem support** – Determines whether the emulated device includes a modem that supports the GSM (Global System for Mobile Communications) telephony radio system.	yes, no
`hw.initialOrientation`	**Initial screen orientation** – Configures the initial orientation of the screen on the emulated device (portrait or landscape mode). In portrait mode, the screen is taller than it is wide. In landscape mode, the screen is wider than it is tall. When running the emulated device, you can change the orientation if both portrait and landscape are supported in the device profile.	portrait, landscape
`hw.keyboard`	**Keyboard support** – Determines whether the emulated device supports a QWERTY keyboard.	yes, no

PROPERTY	DESCRIPTION	OPTIONS
`hw.keyboard.charmap`	**Keyboard charmap name** – The name of the hardware charmap for this device. NOTE: This should always be the default **qwerty2** unless you have modified the system image accordingly. This name is sent to the kernel at boot time. Using an incorrect name will result in an unusable virtual device.	
`hw.keyboard.lid`	**Keyboard lid support** – If keyboard support is enabled, this setting determines whether the QWERTY keyboard can be closed/hidden or opened/visible. This setting will be ignored if hw.keyboard is set to false. NOTE: the default value is false if the emulated device targets API level 12 or higher.	yes, no
`hw.lcd.backlight`	**LCD backlight** – Determines whether an LCD backlight is simulated by the emulated device.	yes, no
`hw.lcd.density`	**LCD density** – The density of the emulated LCD display, measured in density-independent pixels, or dp (dp is a virtual pixel unit). When the setting is 160 dp, each dp corresponds to one physical pixel. At runtime, Android uses this value to select and scale the appropriate resources/assets for correct display rendering.	120, 160, 240, 213, 320
`hw.lcd.depth`	**LCD color depth** – The color bit-depth of the emulated framebuffer that holds the bitmap for driving the LCD display. This value can be 16 bits (65,536 possible colors) or 32 bits (16,777,216 colors plus transparency). The 32-bit setting can make the emulator run slightly slower but with better color accuracy.	16, 32
`hw.lcd.height`	**LCD pixel height** – The number of pixels that make up the vertical dimension of the emulated LCD display.	
`hw.lcd.width`	**LCD pixel width** – The number of pixels that make up the horizontal dimension of the emulated LCD display.	

PROPERTY	DESCRIPTION	OPTIONS
`hw.mainKeys`	**Hardware Back/Home keys** – Determines whether the emulated device supports hardware Back and Home navigation buttons. You can set this value to **yes** if the buttons are implemented only in software. If `hw.mainKeys` is set to **yes**, the emulator will not display navigation buttons on the screen, but you can use the emulator side panel to "press" these buttons.	yes, no
`hw.ramSize`	**Device RAM Size** – The amount of physical RAM on the emulated device, in megabytes. The default value will be computed from the screen size or the skin version. Increasing the size can provide faster emulator operation, but at the expense of demanding more resources from your development computer.	
`hw.screen`	**Touch screen type** – Defines the type of screen on the emulated device. A multi-touch screen can track two or more fingers on the touch interface. A touch screen can detect only single-finger touch events. A no-touch screen does not detect touch events.	touch, multi-touch, no-touch
`hw.sdCard`	**SDCard support** – Determines whether the emulated device supports insertion and removal of virtual SD (Secure Digital) cards. The emulator uses mountable disk images stored on your development computer to simulate the partitions of actual SD card devices (see hw.sdCard.path).	yes, no
`sdcard.size`	**SDCard size** – Specifies the size of the virtual SD card file at the location specified by `hw.sdCard.path`. available on the device (in bytes). If this value is an integer, it specifies the size in bytes. You can also specify the size in kilobytes, megabytes, and gigabytes by appending **K**, **M**, or **G** to the value. The minimum size is **9M** and the maximum size is **1023G**.	
`hw.sdCard.path`	**SDCard Image Path** – Specifies the filename and path to an SD card partition image file on your development computer. For example, this path could be set to **C:\sd\sdcard.img** on Windows.	

PROPERTY	DESCRIPTION	OPTIONS
`hw.sensors.magnetic_field`	**Magnetic Field Sensor** – Determines whether the emulated device supports a magnetic field sensor. The magnetic field sensor (also known as magnetometer) reports the ambient geomagnetic field as measured along three sensor axes. Enable this setting for apps that need access to a compass reading. For example, a navigation app might use this sensor to detect which direction the user faces.	yes, no
`hw.sensors.orientation`	**Orientation Sensor** – Determines whether the emulated device provides orientation sensor values. The orientation sensor measures degrees of rotation that a device makes around all three physical axes (x, y, z). Note that the orientation sensor was deprecated as of Android 2.2 (API level 8).	yes, no
`hw.sensors.proximity`	**Proximity Sensor** – Determines whether the emulated device supports a proximity sensor. This sensor measures the proximity of an object relative to the view screen of a device. This sensor is typically used to determine whether a handset is being held up to a person's ear.	yes, no
`hw.sensors.temperature`	**Temperature Sensor** – Determines whether the emulated device supports a temperature sensor. This sensor measures the temperature of the device in degrees Celsius (°C).	yes, no
`hw.touchScreen`	**Touch-screen support** – Determines whether the emulated device supports a touch screen. The touch screen is used for direct manipulation of objects on the screen.	yes, no
`hw.trackBall`	**Trackball support** – Determines whether the emulated device supports a trackball.	yes, no
`hw.useext4`	**EXT4 file system support** – Determines whether the emulated device uses the Linux EXT4 file system for partitions. Because the file system type is now auto-detected, this option is deprecated and ignored.	no

PROPERTY	DESCRIPTION	OPTIONS
`kernel.newDeviceNaming`	**Kernel new device naming** – Used to specify whether the kernel requires a new device naming scheme. This is typically used with Linux 3.10 kernels and later. If set to **autodetect**, the emulator will automatically detect whether the kernel requires a new device naming scheme.	autodetect, yes, no
`kernel.parameters`	**Kernel parameters** – Specifies the string of Linux kernel boot parameters. By default, this setting is left blank.	
`kernel.path`	**Kernel path** – Specifies the path to the Linux kernel. If this path is not specified, the emulator looks in the emulator system directory for kernel-ranchu.	
`kernel.supportsYaffs2`	**YAFFS2 partition support** – Determines whether the kernel supports YAFFS2 (Yet Another Flash File System 2) partitions. Typically, this applies only to kernels before Linux 3.10. If set to **autodetect** the emulator will automatically detect whether the kernel can mount YAFFS2 file systems.	autodetect, yes, no
`skin.name`	**Skin name** – The name for an Android emulator skin. A skin is a collection of files that defines the visual and control elements of an emulator display; it describes what the window of the AVD will look like on your development computer. A skin describes screen size, buttons, and the overall design, but it does not affect the operation of your app.	
`skin.path`	**Skin path** – Path to the directory that contains the emulator skin files specified in skin.name This directory contains hardware.ini layout files, and image files for the display elements of the skin.	
`skin.dynamic`	**Skin dynamic** – Whether or not the skin is dynamic. The emulator skin is a dynamic skin if the emulator is to construct a skin of a given size based on a specified width and height.	no

For more information about these properties, see Hardware Profile Properties.

Android emulator troubleshooting

7/8/2021 • 15 minutes to read • Edit Online

This article describes the most common warning messages and issues that occur while configuring and running the Android Emulator. In addition, it describes solutions for resolving these errors as well as various troubleshooting tips to help you diagnose emulator problems.

Deployment issues on Windows

Some error messages may be displayed by the emulator when you deploy your app. The most common errors and solutions are explained here.

Deployment errors

If you see an error about a failure to install the APK on the emulator or a failure to run the Android Debug Bridge (**adb**), verify that the Android SDK can connect to your emulator. To verify emulator connectivity, use the following steps:

1. Launch the emulator from the **Android Device Manager** (select your virtual device and click **Start**).
2. Open a command prompt and go to the folder where **adb** is installed. If the Android SDK is installed at its default location, **adb** is located at **C:\Program Files (x86)\Android\android-sdk\platform-tools\adb.exe**; if not, modify this path for the location of the Android SDK on your computer.
3. Type the following command:

```
adb devices
```

4. If the emulator is accessible from the Android SDK, the emulator should appear in the list of attached devices. For example:

```
List of devices attached
emulator-5554   device
```

5. If the emulator does not appear in this list, start the **Android SDK Manager**, apply all updates, then try launching the emulator again.

MMIO access error

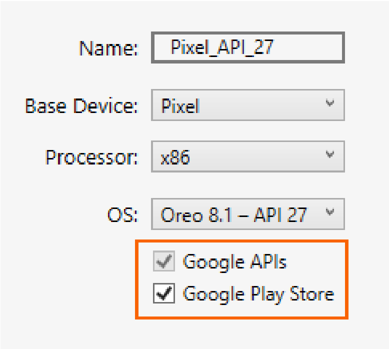
If the message **An MMIO access error has occurred** is displayed, restart the emulator.

Missing Google Play Services

If the virtual device you are running in the emulator does not have Google Play Services or Google Play Store installed, this condition is often caused by creating a virtual device without including these packages. When you create a virtual device (see Managing Virtual Devices with the Android Device Manager), be sure to select one or both of the following options:

- **Google APIs** – includes Google Play Services in the virtual device.
- **Google Play Store** – includes Google Play Store in the virtual device.

For example, this virtual device will include Google Play Services and Google Play Store:



NOTE

Google Play Store images are available only for some base device types such as Pixel, Pixel 2, Nexus 5, and Nexus 5X.

Performance issues

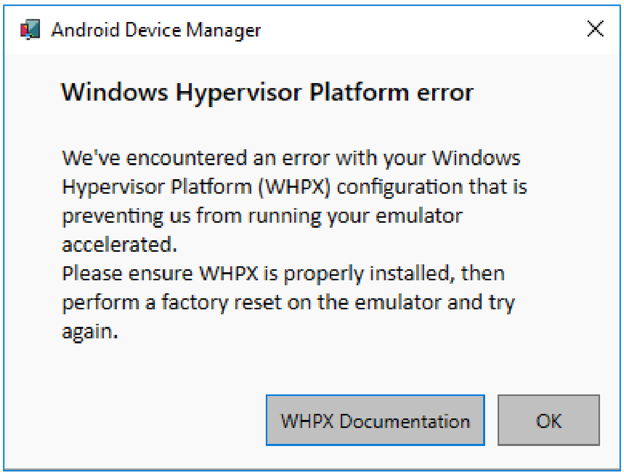
Performance issues are typically caused by one of the following problems:

- The emulator is running without hardware acceleration.
- The virtual device running in the emulator is not using an x86-based system image.

The following sections cover these scenarios in more detail.

Hardware acceleration is not enabled

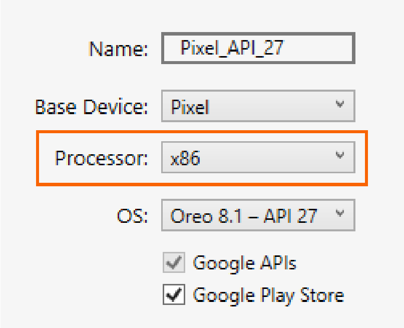
If hardware acceleration is not enabled, starting a virtual device from the Device Manager will produce a dialog with an error message indicating that the Windows Hypervisor Platform (WHPX) is not configured properly:



If this error message is displayed, see Hardware acceleration issues below for steps you can take to verify and enable hardware acceleration.

Acceleration is enabled but the emulator runs too slowly

A common cause for this problem is not using an x86-based image in your virtual device (AVD). When you create a virtual device (see Managing Virtual Devices with the Android Device Manager), be sure to select an x86-based system image:



Hardware acceleration issues

Whether you are using Hyper-V or HAXM for hardware acceleration, you may run into configuration problems or conflicts with other software on your computer. You can verify that hardware acceleration is enabled (and which acceleration method the emulator is using) by opening a command prompt and entering the following command:

```
"C:\Program Files (x86)\Android\android-sdk\emulator\emulator-check.exe" accel
```

This command assumes that the Android SDK is installed at the default location of **C:\Program Files (x86)\Android\android-sdk**; if not, modify the above path for the location of the Android SDK on your computer.

Hardware acceleration not available

If Hyper-V is available, a message like the following example will be returned from the **emulator-check.exe accel** command:

```
HAXM is not installed, but Windows Hypervisor Platform is available.
```

If HAXM is available, a message like the following example will be returned:

```
HAXM version 6.2.1 (4) is installed and usable.
```

If hardware acceleration is not available, a message like the following example will be displayed (the emulator looks for HAXM if it is unable to find Hyper-V):

```
HAXM is not installed on this machine
```

If hardware acceleration is not available, see Accelerating with Hyper-V to learn how to enable hardware acceleration on your computer.

Incorrect BIOS settings

If the BIOS has not been configured properly to support hardware acceleration, a message similar to the following example will be displayed when you run the **emulator-check.exe accel** command:

```
VT feature disabled in BIOS/UEFI
```

To correct this problem, reboot into your computer's BIOS and enable the following options:

- Virtualization Technology (may have a different label depending on motherboard manufacturer).
- Hardware Enforced Data Execution Prevention.

If hardware acceleration is enabled and the BIOS is configured properly, the emulator should run successfully with hardware acceleration. However, problems may still result due to issues that are specific to Hyper-V and HAXM, as explained next.

Hyper-V issues

In some cases, enabling both **Hyper-V** and **Windows Hypervisor Platform** in the **Turn Windows features on or off** dialog may not properly enable Hyper-V. To verify that Hyper-V is enabled, use the following steps:

1. Enter **powershell** in the Windows search box.

2. Right-click **Windows PowerShell** in the search results and select **Run as administrator**.

3. In the PowerShell console, enter the following command:

   ```
   Get-WindowsOptionalFeature -FeatureName Microsoft-Hyper-V-All -Online
   ```

   If Hyper-V is not enabled, a message similar to the following example will be displayed to indicate that the state of Hyper-V is **Disabled**:

   ```
   FeatureName      : Microsoft-Hyper-V-All
   DisplayName      : Hyper-V
   Description      : Provides services and management tools for creating and running virtual machines
   and their resources.
   RestartRequired  : Possible
   State            : Disabled
   CustomProperties :
   ```

4. In the PowerShell console, enter the following command:

   ```
   Get-WindowsOptionalFeature -FeatureName HypervisorPlatform -Online
   ```

   If the Hypervisor is not enabled, a message similar to the following example will be displayed to indicate that the state of HypervisorPlatform is **Disabled**:

   ```
   FeatureName      : HypervisorPlatform
   DisplayName      : Windows Hypervisor Platform
   Description      : Enables virtualization software to run on the Windows hypervisor
   RestartRequired  : Possible
   State            : Disabled
   CustomProperties :
   ```

If Hyper-V and/or HypervisorPlatform are not enabled, use the following PowerShell commands to enable them:

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V -All
Enable-WindowsOptionalFeature -Online -FeatureName HypervisorPlatform -All
```

After these commands complete, reboot.

For more information about enabling Hyper-V (including techniques for enabling Hyper-V using the Deployment Image Servicing and Management tool), see Install Hyper-V.

HAXM issues

HAXM issues are often the result of conflicts with other virtualization technologies, incorrect settings, or an out-of-date HAXM driver.

HAXM process is not running

If HAXM is installed, you can verify that the HAXM process is running by opening a command prompt and entering the following command:

```
sc query intelhaxm
```

If the HAXM process is running, you should see output similar to the following result:

```
SERVICE_NAME: intelhaxm
    TYPE               : 1  KERNEL_DRIVER
    STATE              : 4  RUNNING
                            (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
    WIN32_EXIT_CODE    : 0  (0x0)
    SERVICE_EXIT_CODE  : 0  (0x0)
    CHECKPOINT         : 0x0
    WAIT_HINT          : 0x0
```

If `STATE` is not set to `RUNNING`, see How to Use the Intel Hardware Accelerated Execution Manager to resolve the problem.

HAXM virtualization conflicts

HAXM can conflict with other technologies that use virtualization, such as Hyper-V, Windows Device Guard, and some antivirus software:

- **Hyper-V** – If you are using a version of Windows before the **Windows 10 April 2018 update (build 1803)** and Hyper-V is enabled, follow the steps in Disabling Hyper-V so that HAXM can be enabled.

- **Device Guard** – Device Guard and Credential Guard can prevent Hyper-V from being disabled on Windows machines. To disable Device Guard and Credential Guard, see Disabling Device Guard.

- **Antivirus Software** – If you are running antivirus software that uses hardware-assisted virtualization (such as Avast), disable or uninstall this software, reboot, and retry the Android emulator.

Incorrect BIOS settings

If you are using HAXM on a Windows PC, HAXM will not work unless virtualization technology (Intel VT-x) is enabled in the BIOS. If VT-x is disabled, you will get an error similar to the following when you attempt to start the Android Emulator:

This computer meets the requirements for HAXM, but Intel Virtualization Technology (VT-x) is not turned on.

To correct this error, boot the computer into the BIOS, enable both VT-x and SLAT (Second-Level Address Translation), then restart the computer back into Windows.

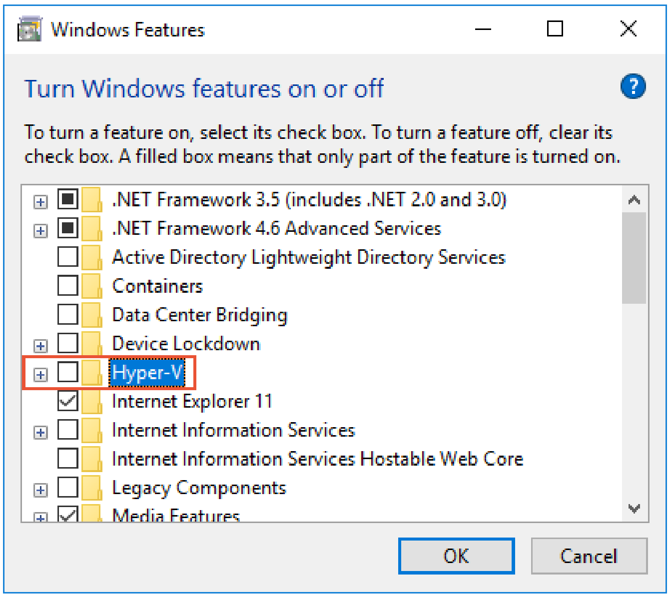
Disabling Hyper-V

If you are using a version of Windows before the **Windows 10 April 2018 Update (build 1803)** and Hyper-V is enabled, you must disable Hyper-V and reboot your computer to install and use HAXM. If you are using **Windows 10 April 2018 Update (build 1803)** or later, Android Emulator version 27.2.7 or later can use Hyper-V (instead of HAXM) for hardware acceleration, so it is not necessary to disable Hyper-V.

You can disable Hyper-V from the Control Panel by following these steps:

1. Enter **windows features** in the Windows search box and select **Turn Windows features on or off** in the search results.

2. Uncheck **Hyper-V**:



3. Restart the computer.

Alternately, you can use the following PowerShell command to disable the Hyper-V Hypervisor:

```
Disable-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V-Hypervisor
```

Intel HAXM and Microsoft Hyper-V cannot both be active at the same time. Unfortunately, there is no way to switch between Hyper-V and HAXM without restarting your computer.

In some cases, using the above steps will not succeed in disabling Hyper-V if Device Guard and Credential Guard are enabled. If you are unable to disable Hyper-V (or it seems to be disabled but HAXM installation still fails), use the steps in the next section to disable Device Guard and Credential Guard.

Disabling Device Guard

Device Guard and Credential Guard can prevent Hyper-V from being disabled on Windows machines. This situation is often a problem for domain-joined machines that are configured and controlled by an owning organization. On Windows 10, use the following steps to see if **Device Guard** is running:

1. Enter **System info** in the Windows search box and select **System Information** in the search results.

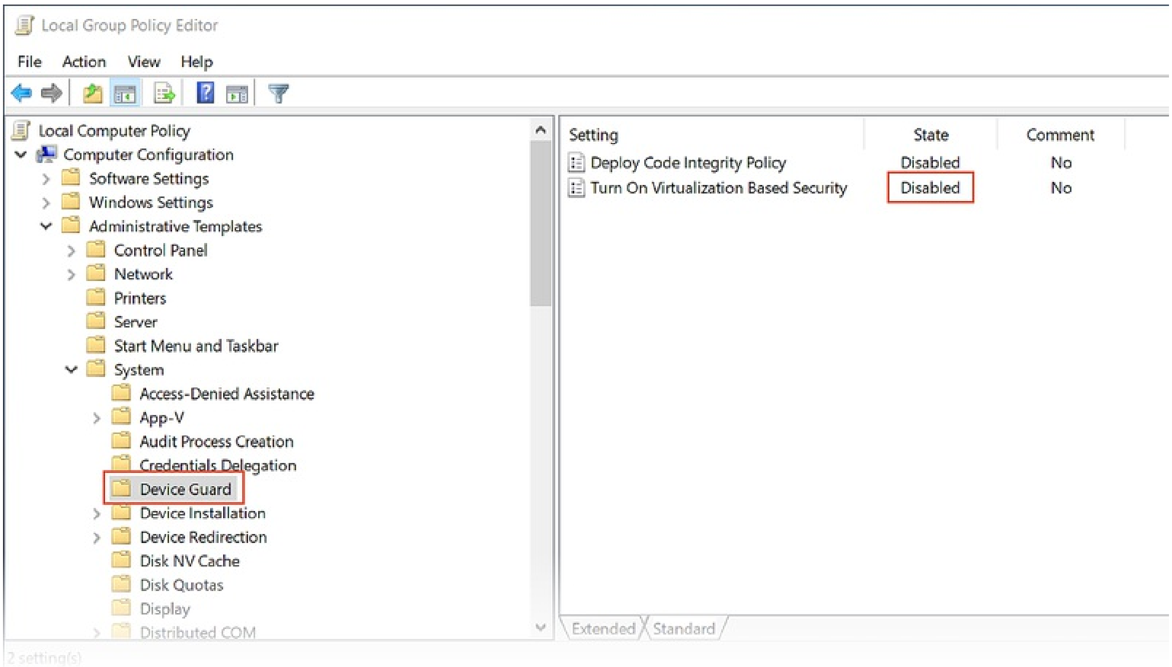
2. In the **System Summary**, look to see if **Device Guard Virtualization based security** is present and is in the **Running** state:

If Device Guard is enabled, use the following steps to disable it:

1. Ensure that **Hyper-V** is disabled (under **Turn Windows Features on or off**) as described in the previous section.

2. In the Windows Search Box, enter **gpedit.msc** and select the **Edit group policy** search result. These steps launch the **Local Group Policy Editor**.

3. In the **Local Group Policy Editor**, navigate to **Computer Configuration > Administrative Templates > System > Device Guard**:



4. Change **Turn On Virtualization Based Security** to **Disabled** (as shown above) and exit the **Local Group Policy Editor**.

5. In the Windows Search Box, enter **cmd**. When **Command Prompt** appears in the search results, right-click **Command Prompt** and select **Run as Administrator**.

6. Copy and paste the following commands into the command prompt window (if drive Z: is in use, pick an unused drive letter to use instead):

```
mountvol Z: /s
copy %WINDIR%\System32\SecConfig.efi Z:\EFI\Microsoft\Boot\SecConfig.efi /Y
bcdedit /create {0cb3b571-2f2e-4343-a879-d86a476d7215} /d "DebugTool" /application osloader
bcdedit /set {0cb3b571-2f2e-4343-a879-d86a476d7215} path "\EFI\Microsoft\Boot\SecConfig.efi"
bcdedit /set {bootmgr} bootsequence {0cb3b571-2f2e-4343-a879-d86a476d7215}
bcdedit /set {0cb3b571-2f2e-4343-a879-d86a476d7215} loadoptions DISABLE-LSA-ISO,DISABLE-VBS
bcdedit /set {0cb3b571-2f2e-4343-a879-d86a476d7215} device partition=Z:
mountvol Z: /d
```

7. Restart your computer. On the boot screen, you should see a prompt similar to the following message:

   Do you want to disable Credential Guard?

   Press the indicated key to disable Credential Guard as prompted.

8. After the computer reboots, check again to ensure that Hyper-V is disabled (as described in the previous steps).

If Hyper-V is still not disabled, the policies of your domain-joined computer may prevent you from disabling Device Guard or Credential Guard. In this case, you can request an exemption from your domain administrator to allow you to opt out of Credential Guard. Alternately, you can use a computer that is not domain-joined if you must use HAXM.

Additional troubleshooting tips

The following suggestions are often helpful in diagnosing Android emulator issues.

Starting the emulator from the command line

If the emulator is not already running, you can start it from the command line (rather than from within Visual Studio) to view its output. Typically, Android emulator AVD images are stored at the following location (replace *username* with your Windows user name):

C:\Users*username*\\.android\avd

You can launch the emulator with an AVD image from this location by passing in the folder name of the AVD. For example, this command launches an AVD named **Pixel_API_27**:

```
"C:\Program Files (x86)\Android\android-sdk\emulator\emulator.exe" -partition-size 512 -no-boot-anim -
verbose -feature WindowsHypervisorPlatform -avd Pixel_API_27 -prop monodroid.avdname=Pixel_API_27
```

This example assumes that the Android SDK is installed at the default location of **C:\Program Files (x86)\Android\android-sdk**; if not, modify the above path for the location of the Android SDK on your computer.

When you run this command, it will produce many lines of output while the emulator starts up. In particular, lines such as the following example will be printed if hardware acceleration is enabled and working properly (in this example, HAXM is used for hardware acceleration):

```
emulator: CPU Acceleration: working
emulator: CPU Acceleration status: HAXM version 6.2.1 (4) is installed and usable.
```

Viewing Device Manager logs

Often you can diagnose emulator problems by viewing the Device Manager logs. These logs are written to the following location:

C:\Users*username*\\AppData\Local\Xamarin\Logs\16.0

You can view each **DeviceManager.log** file by using a text editor such as Notepad. The following example log entry indicates that HAXM was not found on the computer:

```
Component Intel x86 Emulator Accelerator (HAXM installer) r6.2.1 [Extra: (Intel Corporation)] not present on
the system
```

Deployment issues on macOS

Some error messages may be displayed by the emulator when you deploy your app. The most common errors and solutions are explained below.

Deployment errors

If you see an error about a failure to install the APK on the emulator or a failure to run the Android Debug Bridge (**adb**), verify that the Android SDK can connect to your emulator. To verify connectivity, use the following steps:

1. Launch the emulator from the **Android Device Manager** (select your virtual device and click **Start**).
2. Open a command prompt and go to the folder where **adb** is installed. If the Android SDK is installed at its default location, **adb** is located at **~/Library/Developer/Xamarin/android-sdk-macosx/platform-tools/adb**; if not, modify this path for the location of the Android SDK on your computer.

3. Type the following command:

```
adb devices
```

4. If the emulator is accessible from the Android SDK, the emulator should appear in the list of attached devices. For example:

```
List of devices attached
emulator-5554	device
```

5. If the emulator does not appear in this list, start the **Android SDK Manager**, apply all updates, then try launching the emulator again.

MMIO access error

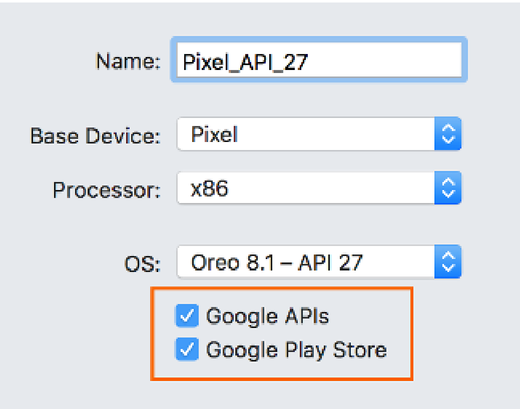
If **An MMIO access error has occurred** is displayed, restart the emulator.

Missing Google Play Services

If the virtual device you are running in the emulator does not have Google Play Services or Google Play Store installed, this condition is usually caused by creating a virtual device without including these packages. When you create a virtual device (see Managing Virtual Devices with the Android Device Manager), be sure to select one or both of the following:

- **Google APIs** – includes Google Play Services in the virtual device.
- **Google Play Store** – includes Google Play Store in the virtual device.

For example, this virtual device will include Google Play Services and Google Play Store:



> **NOTE**
>
> Google Play Store images are available only for some base device types such as Pixel, Pixel 2, Nexus 5, and Nexus 5X.

Performance issues

Performance issues are typically caused by one of the following problems:

- The emulator is running without hardware acceleration.
- The virtual device running in the emulator is not using an x86-based system image.

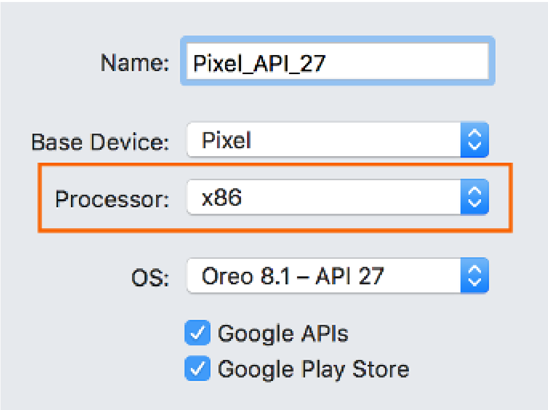
The following sections cover these scenarios in more detail.

Hardware acceleration is not enabled

If hardware acceleration is not enabled, a dialog may pop up with a message such as **device will run unaccelerated** when you deploy your app to the Android emulator. If you are not certain whether hardware acceleration is enabled on your computer (or you would like to know which technology is providing the acceleration), see Hardware acceleration issues below for steps you can take to verify and enable hardware acceleration.

Acceleration is enabled but the emulator runs too slowly

A common cause for this problem is not using an x86-based image in your virtual device. When you create virtual device (see Managing Virtual Devices with the Android Device Manager), be sure to select an x86-based system image:



Hardware acceleration issues

Whether you are using the Hypervisor Framework or HAXM for hardware acceleration of the emulator, you may run into problems caused by installation issues or an out-of-date version of macOS. The following sections can help you resolve this issue.

Hypervisor Framework issues

If you are using macOS 10.10 or later on a newer Mac, the Android emulator will automatically use the Hypervisor Framework for hardware acceleration. However, some older Macs or Macs running a version of macOS earlier than 10.10 may not provide Hypervisor Framework support.

To determine whether or not your Mac supports the Hypervisor Framework, open a Terminal and enter the following command:

```
sysctl kern.hv_support
```

If your Mac supports the Hypervisor Framework, the above command will return the following result:

```
kern.hv_support: 1
```

If the Hypervisor Framework is not available on your Mac, you can follow the steps in Accelerating with HAXM to use HAXM for acceleration instead.

HAXM issues

If the Android Emulator does not start properly, this problem is often caused by problems with HAXM. HAXM issues are often the result of conflicts with other virtualization technologies, incorrect settings, or an out-of-date HAXM driver. Try reinstalling the HAXM driver, using the steps detailed in Installing HAXM.

Additional troubleshooting tips

The following suggestions are often helpful in diagnosing Android emulator issues.

Starting the emulator from the command line

If the emulator is not already running, you can start it from the command line (rather than from within Visual Studio for Mac) to view its output. Typically, Android emulator AVD images are stored at the following location:

~/.android/avd

You can launch the emulator with an AVD image from this location by passing in the folder name of the AVD. For example, this command launches an AVD named **Pixel_2_API_28**:

```
~/Library/Developer/Xamarin/android-sdk-macosx/emulator/emulator -partition-size 512 -no-boot-anim -verbose
-feature WindowsHypervisorPlatform -avd Pixel_2_API_28 -prop monodroid.avdname=Pixel_2_API_28
```

If the Android SDK is installed at its default location, the emulator is located in the **~/Library/Developer/Xamarin/android-sdk-macosx/emulator** directory; if not, modify this path for the location of the Android SDK on your Mac.

When you run this command, it will produce many lines of output while the emulator starts up. In particular, lines such as the following example will be printed if hardware acceleration is enabled and working properly (in this example, Hypervisor Framework is used for hardware acceleration):

```
emulator: CPU Acceleration: working
emulator: CPU Acceleration status: Hypervisor.Framework OS X Version 10.13
```

Viewing Device Manager logs

Often you can diagnose emulator problems by viewing the Device Manager logs. These logs are written to the following location:

~/Library/Logs/XamarinDeviceManager

You can view each **Android Devices.log** file by double-clicking it to open it in the Console app. The following example log entry indicates that HAXM was not found:

```
Component Intel x86 Emulator Accelerator (HAXM installer) r6.2.1 [Extra: (Intel Corporation)] not present on
the system
```

Set Up Device for Development

7/8/2021 • 5 minutes to read • Edit Online

This article explains how to setup an Android device and connect it to a computer so that the device may be used to run and debug Xamarin.Android applications.

After testing on an Android emulator, you will want to see and test your apps running on an Android device. You will need to enable debugging and connect the device to the computer.

Each of these steps will be covered in more detail in the sections below.

Enable Debugging on the Device

A device must be enabled for debugging in order to test an Android application. Developer options on Android have been hidden by default since version 4.2, and enabling them can vary based on the Android version.

Android 9.0+

For Android 9.0 and higher, debugging is enabled by following these steps:

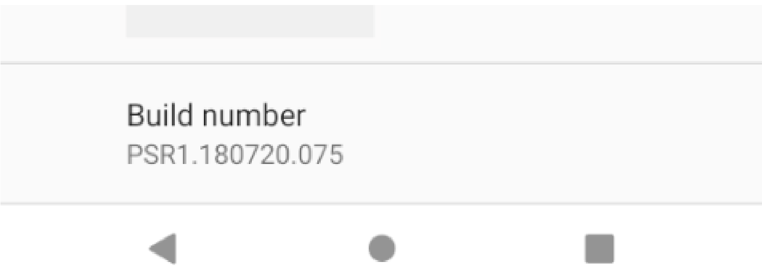
1. Go to the **Settings** screen.
2. Select **About Phone** .
3. Tap **Build Number** 7 times until **You are now a developer!** is visible.

Android 8.0 and Android 8.1

1. Go to the **Settings** screen.
2. Select **System**.
3. Select **About Phone**
4. Tap **Build Number** 7 times until **You are now a developer!** is visible.

Android 7.1 and lower

1. Go to the **Settings** screen.
2. Select **About Phone**.
3. Tap **Build Number** 7 times until **You are now a developer!** is visible.



Verify that USB debugging is enabled

After enabling developer mode on your device, you must ensure that USB debugging is enabled on the device. This also varies based on the Android version.

Android 9.0+

Navigate to **Settings > System > Advanced > Developer Options** and enable **USB Debugging**.

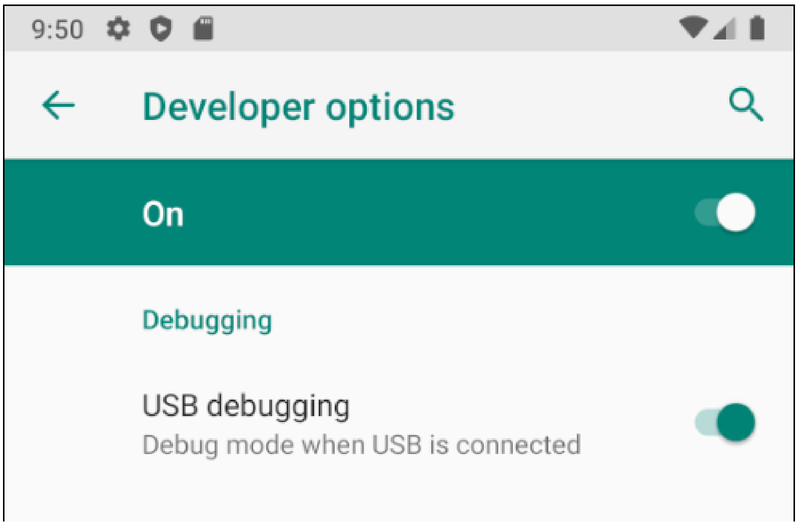
Android 8.0 and Android 8.1

Navigate to **Settings > System > Developer Options** and enable **USB Debugging**.

Android 7.1 and lower

Navigate to **Settings > Developer Options** and enable **USB Debugging**.

Once the **Developer Options** tab is available under **Settings > System**, open it to reveal developer settings:

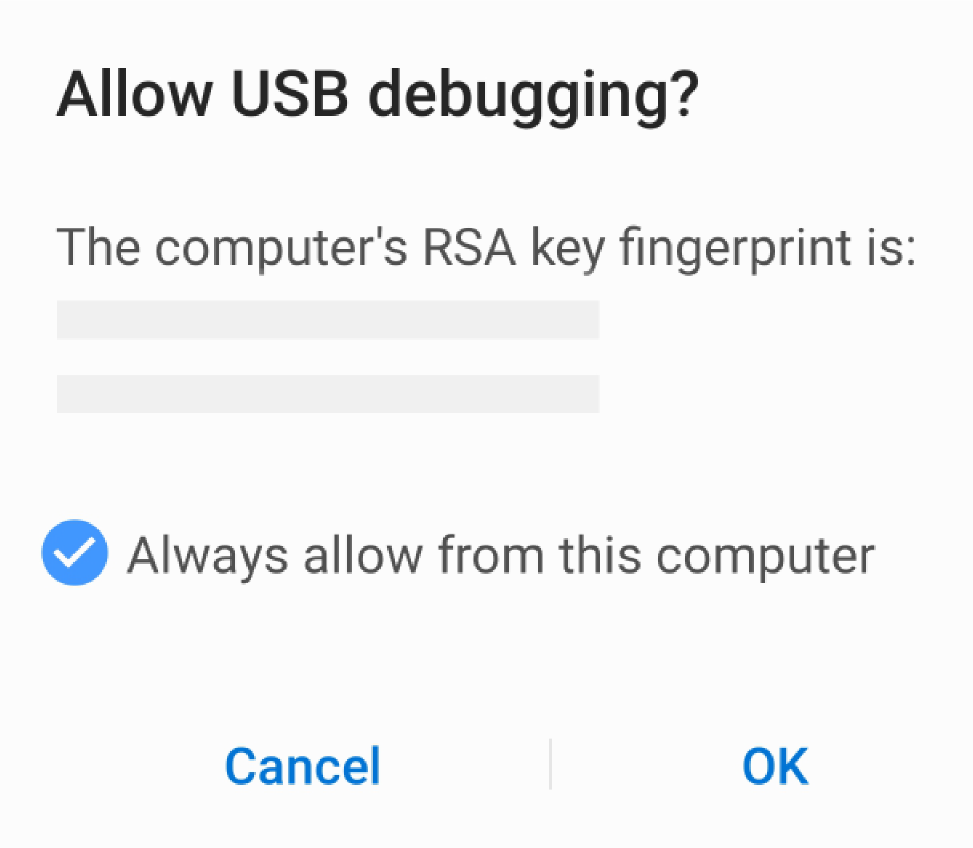


This is the place to enable developer options such as USB debugging and stay awake mode.

Connect the device to the computer

The final step is to connect the device to the computer. The easiest and most reliable way is to do so over USB.

You will receive a prompt to trust the computer on your device if you have not used it for debugging before. You can also check **Always allow from this computer** to prevent requiring this prompt each time you connect the device.



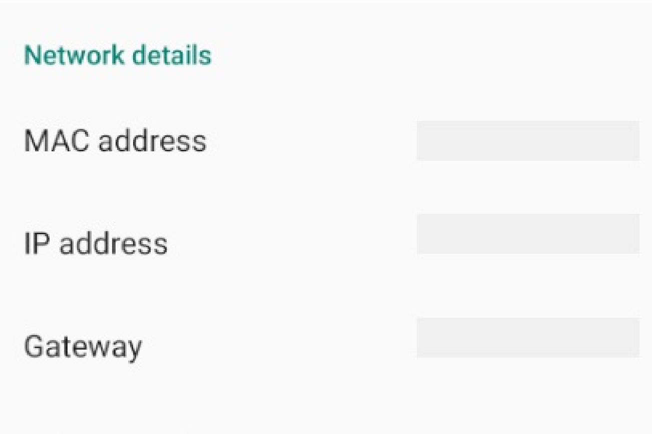
Alternate connection via Wifi

It is possible to connect an Android device to a computer without using a USB cable, over WiFi. This technique requires more effort but could be useful when the device is too far from the computer to remain constantly plugged-in via cable.

Connecting over WiFi

By default, the Android Debug Bridge (*ADB*) is configured to communicate with an Android device via USB. It is possible to reconfigure it to use TCP/IP instead of USB. To do this, both the device and the computer must be on the same WiFi network. To setup your environment to debug over WiFi complete the following steps from the command line:

1. Determine the IP address of your Android device. One way to find out the IP address is to look under **Settings > Network & internet > Wi-Fi**, then tap on the WiFi network that the device is connected to, and then tap on **Advanced**. This will open a dropdown showing information about the network connection, similar to what is seen in the screenshot below:

   On some versions of Android the IP address won't be listed there but can be found instead under **Settings > About phone > Status**.

2. Connect your Android device to your computer via USB.

3. Next, restart ADB so that it using TCP on port 5555. From a command prompt, type the following command:

   ```
   adb tcpip 5555
   ```

   After this command is issued, your computer will not be able to listen to devices that are connected via USB.

4. Disconnect the USB cable connecting your device to your computer.

5. Configure ADB so that it will connect to your Android device on the port that was specified in step 1 above:

   ```
   adb connect 192.168.1.28:5555
   ```

   Once this command finishes the Android device is connected to the computer via WiFi.

   When you're finished debugging via WiFi, it's possible to reset ADB back to USB mode with the following command:

```
adb usb
```

It's possible to request ADB to list the devices that are connected to the computer. Regardless of how the devices are connected, you can issue the following command at the command prompt to see what is connected:

```
adb devices
```

Troubleshooting

In some cases you might find that your device cannot connect to the computer. In this case you may want to verify that USB drivers are installed.

Install USB Drivers

This step is not necessary for macOS; just connect the device to the Mac with a USB cable.

It may be necessary to install some extra drivers before a Windows computer will recognize an Android device connected by USB.

> **NOTE**
>
> These are the steps to set up a Google Nexus device and are provided as a reference. Steps for your specific device may vary, but will follow a similar pattern. Search the internet for your device if you have trouble.

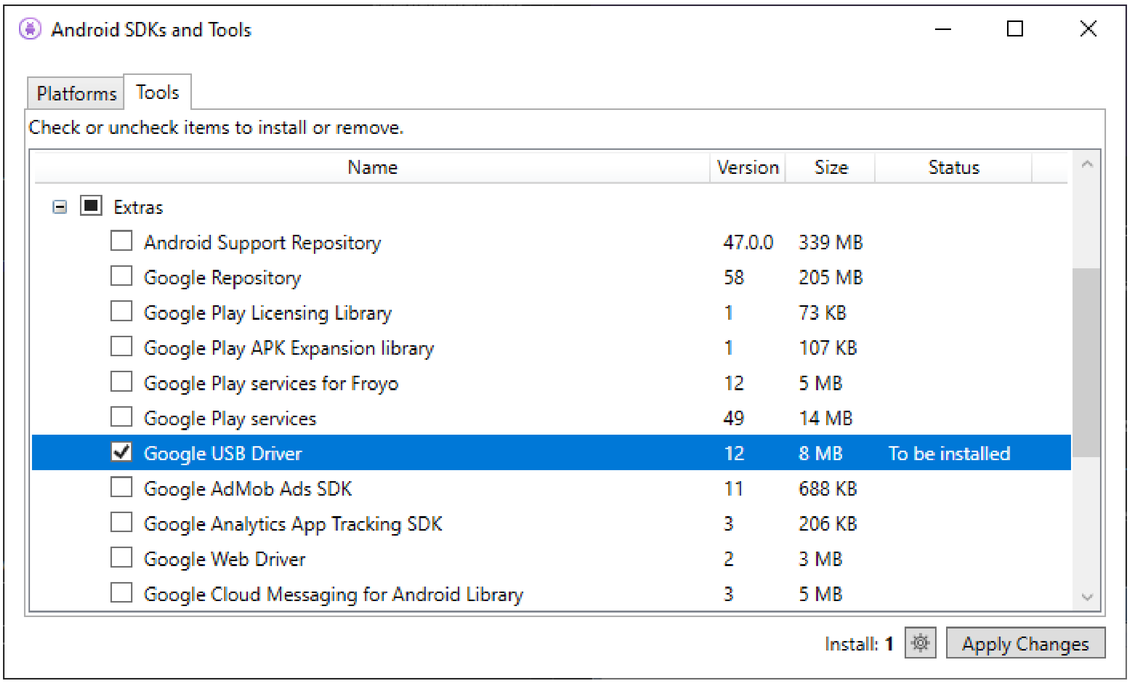
Run the **android.bat** application in the **[Android SDK install path]\tools** directory. By default, the Xamarin.Android installer will put the Android SDK in following location on a Windows computer:

`C:\Users\[username]\AppData\Local\Android\android-sdk`

Download the USB Drivers

Google Nexus devices (with the exception of the Galaxy Nexus) require the Google USB Driver. The driver for the Galaxy Nexus is distributed by Samsung. All other Android devices should use the USB driver from their respective manufacturer.

Install the **Google USB Driver** package by starting the Android SDK Manager, and expanding the **Extras** folder, as can be seen in the follow screenshot:



Check the **Google USB Driver** box, and click the **Apply Changes** button. The driver files are downloaded to the following location:

```
[Android SDK install path]\extras\google\usb\_driver
```

The default path for a Xamarin.Android installation is:

```
C:\Users\[username]\AppData\Local\Android\android-sdk\extras\google\usb_driver
```

Installing the USB Driver

After the USB drivers are downloaded, it is necessary to install them. To install the drivers on Windows 7:

1. Connect your device to the computer with a USB cable.
2. Right-click on the Computer from your desktop or Windows Explorer, and select **Manage** .
3. Select **Devices** in the left pane.
4. Locate and expand **Other Devices** in the right pane.
5. Right-click the device name and select **Update Driver Software** . This will launch the Hardware Update Wizard.
6. Select **Browse my computer for driver software** and click **Next** .
7. Click **Browse** and locate the USB driver folder (the Google USB driver is located in **[Android SDK install path]\extras\google\usb_driver**).
8. Click **Next** to install the driver.

Summary

This article discussed how to configure an Android device for development by enabling debugging on the device. It also covered how to connect the device to a computer using either USB or WiFi.

Related Links

- Android Debug Bridge
- Using Hardware Devices

- Samsung Driver Downloads
- OEM USB Drivers
- Google USB Driver
- XDA Developers : Windows 8 - ADB/fastboot driver problem solved

Microsoft's Mobile OpenJDK Distribution

7/8/2021 • 2 minutes to read • Edit Online

This guide describes the steps for switching to an internal distribution of OpenJDK. This distribution is intended for mobile development.

Overview

Beginning with Visual Studio 15.9 and Visual Studio for Mac 7.7, Visual Studio Tools for Xamarin has moved from Oracle's JDK to a **lightweight version of the OpenJDK that is intended solely for Android development**. This is a required migration as Oracle is ending support for commercial distribution of JDK 8 in 2019, and JDK 8 is a required dependency for all Android development.

The benefits of this move are:

- You will always have an OpenJDK version that works for Android development.
- Downloading Oracle's JDK 9 or greater won't affect the development experience.
- Reduced download size and footprint.
- No more issues with 3rd party servers and installers.

If you'd like to move to the improved experience sooner, builds of the Microsoft Mobile OpenJDK distribution are available for you to test on both Windows and Mac. The setup process is described below, and you can revert back to the Oracle JDK at any time.

Download

The mobile OpenJDK distribution is automatically installed for you if you select the Android SDK packages in the Visual Studio installer on Windows.

On Mac, the mobile OpenJDK will be installed for you as part of the Android workload for new installs. For existing Visual Studio for Mac users, you will be prompted to install it as part of your update. The IDE will prompt you to move to the new JDK, and will switch to using it at the next restart.

Troubleshooting

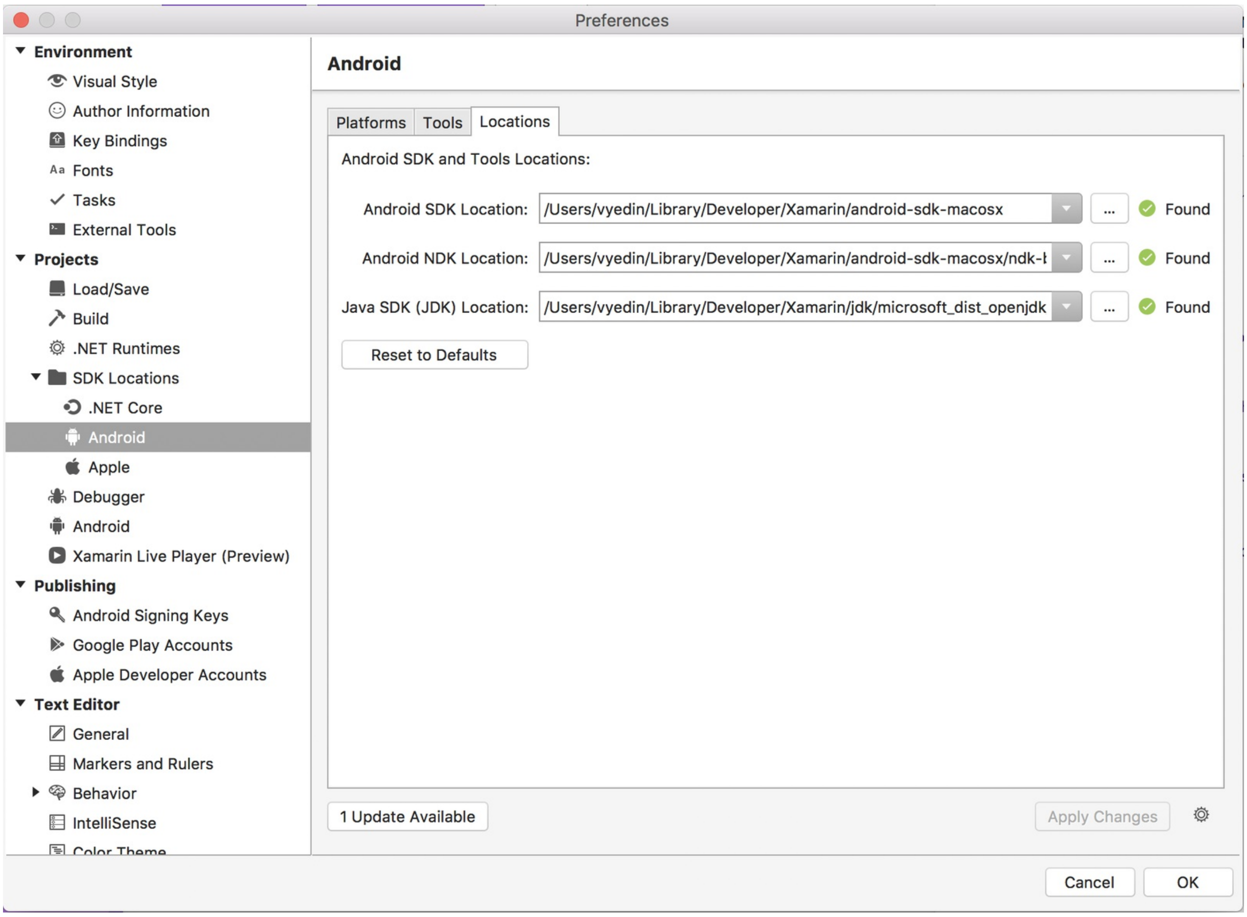
If you encounter issues with the setup on Mac or Windows, you can take the following steps for manual setup:

Check if OpenJDK is installed on the machine in the correct location:

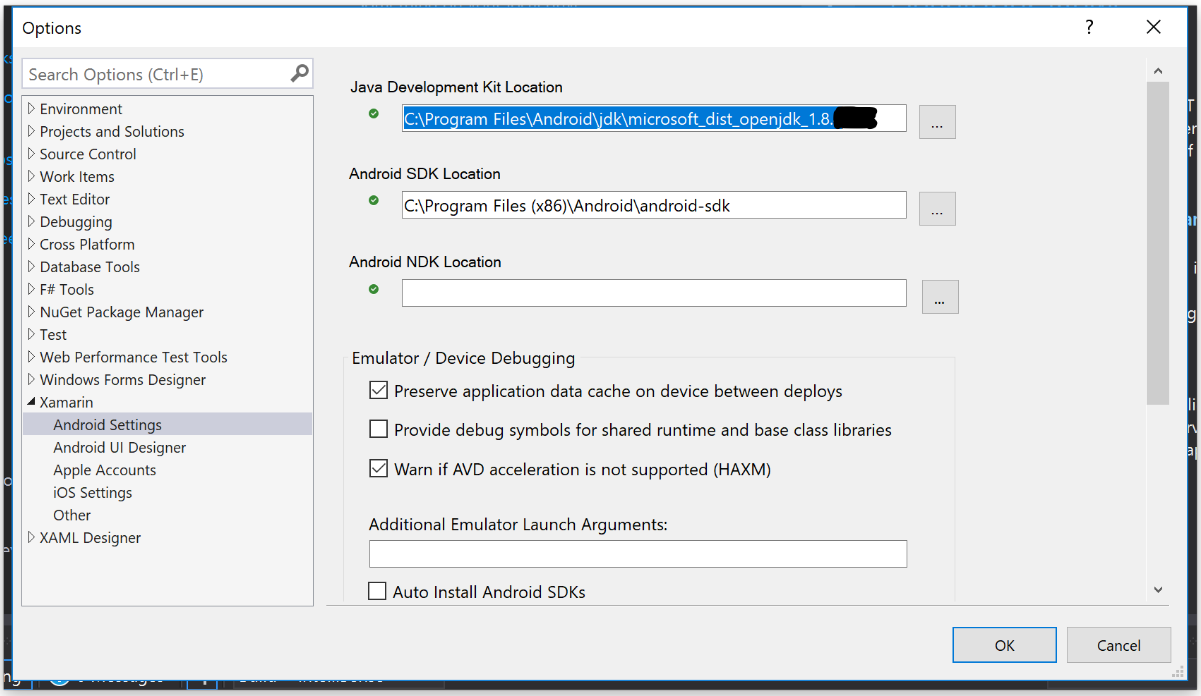
- **Mac** – **$HOME/Library/Developer/Xamarin/jdk/microsoft_dist_openjdk_1.8.0.x**
- **Windows** – **C:\Program Files\Android\jdk\microsoft_dist_openjdk_1.8.0.x**

Point the IDE to the new JDK:

- **Mac** – Click **Tools > SDK Manager > Locations** and change the **Java SDK (JDK) Location** to the full path of the OpenJDK installation. In the following example, this path is set to **$HOME/Library/Developer/Xamarin/jdk/microsoft_dist_openjdk_1.8.0.9** but your version may be newer.



- **Windows** – Click **Tools > Options > Xamarin > Android Settings** and change the **Java Development Kit Location** to the full path of the OpenJDK installation. In the following example, this path is set to **C:\Program Files\Android\jdk\microsoft_dist_openjdk_1.8.0.9**, but your version may be newer:



Known Issues

Package 'OpenJDKV1.RegKey,version=1.8.0.25,chip=x64' failed to install

This may be an issue in some corporate environments. OpenJDK is already on the machine - follow the troubleshooting steps above to point your IDE to the correct location. You can follow the status of the issues here.

Summary

In this article, you learned how to configure your IDE to use Microsoft's Mobile OpenJDK distribution, and how to troubleshoot should you encounter issues.

Hello, Android

11/2/2020 • 2 minutes to read • Edit Online

In this two-part guide, you will build your first Xamarin.Android application using Visual Studio for Mac or Visual Studio and develop an understanding of the fundamentals of Android application development with Xamarin. Along the way, the tools, concepts, and steps required to build and deploy a Xamarin.Android application will be introduced.

Part 1: Quickstart

In the first part of this guide you'll create an application that translates an alphanumeric phone number entered by the user into a numeric phone number, and then calls that number.

Part 2: Deep Dive

In the second part of this document, you'll review what was built and develop a fundamental understanding of how Android applications work.

Related Links

- Android Getting Started
- Debugging in Visual Studio
- Visual Studio for Mac Recipes - Debugging

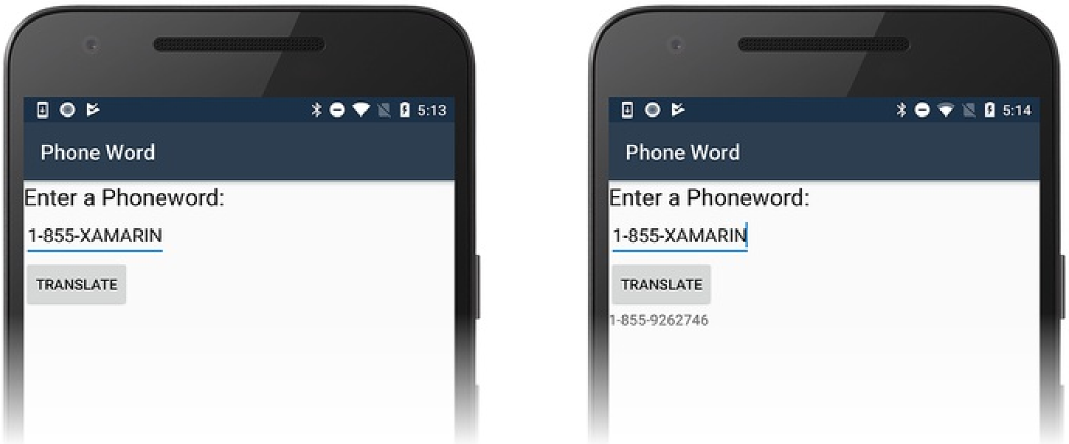
Hello, Android: Quickstart

7/8/2021 • 13 minutes to read • Edit Online

In this two-part guide, you will build your first Xamarin.Android application with Visual Studio and develop an understanding of the fundamentals of Android application development with Xamarin.

Download the sample

You will create an application that translates an alphanumeric phone number (entered by the user) into a numeric phone number and display the numeric phone number to the user. The final application looks like this:



Windows requirements

To follow along with this walkthrough, you will need the following:

- Windows 10.
- Visual Studio 2019 or Visual Studio 2017 (version 15.8 or later): Community, Professional, or Enterprise.

macOS requirements

To follow along with this walkthrough, you will need the following:

- The latest version of Visual Studio for Mac.
- A Mac running macOS High Sierra (10.13) or later.

This walkthrough assumes that the latest version of Xamarin.Android is installed and running on your platform of choice. For a guide to installing Xamarin.Android, refer to the Xamarin.Android Installation guides.

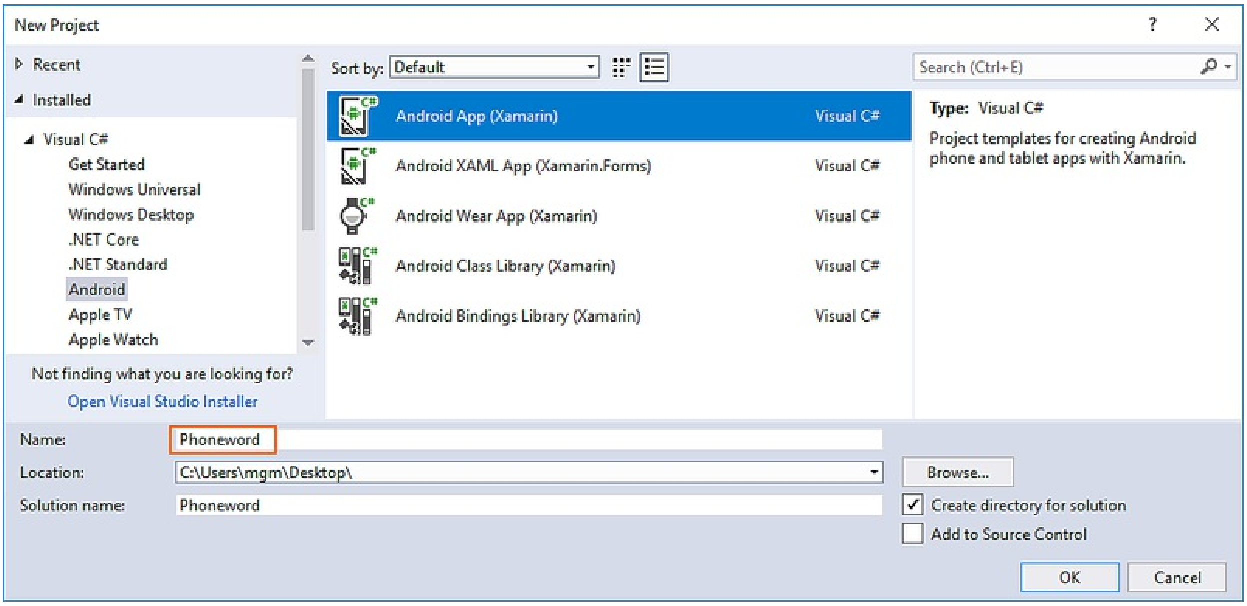
Configuring emulators

If you are using the Android emulator, we recommend that you configure the emulator to use hardware acceleration. Instructions for configuring hardware acceleration are available in Hardware Acceleration for Emulator Performance.

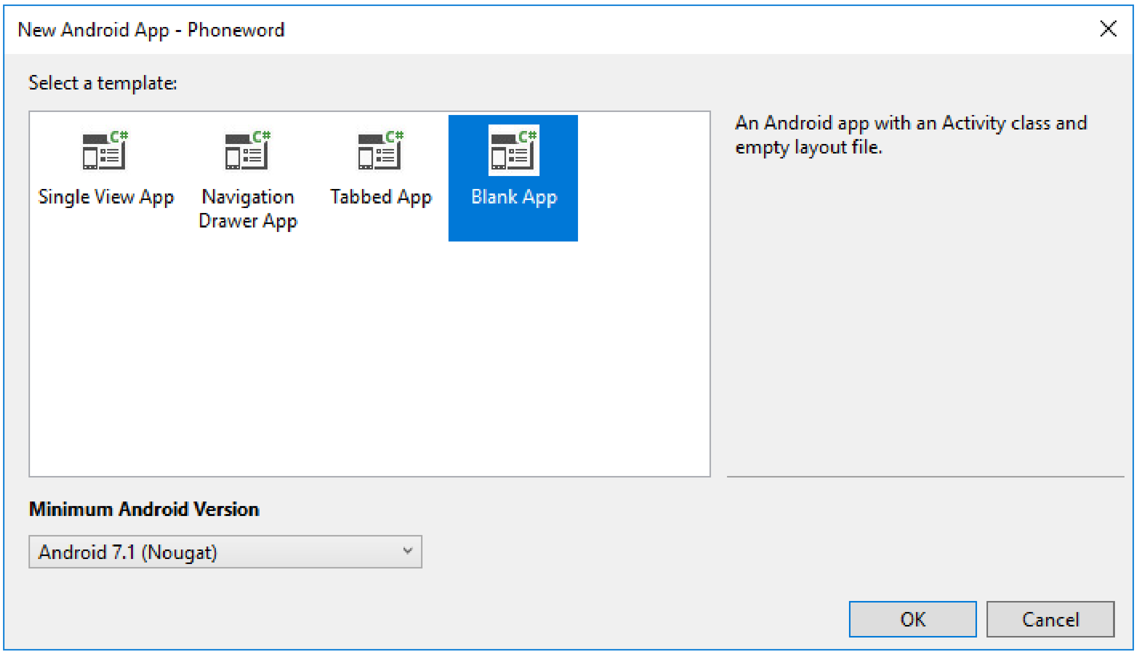
Create the project

Start Visual Studio. Click **File** > **New** > **Project** to create a new project.

In the **New Project** dialog, click the **Android App** template. Name the new project `Phoneword` and click **OK**:



In the **New Android App** dialog, click **Blank App** and click **OK** to create the new project:



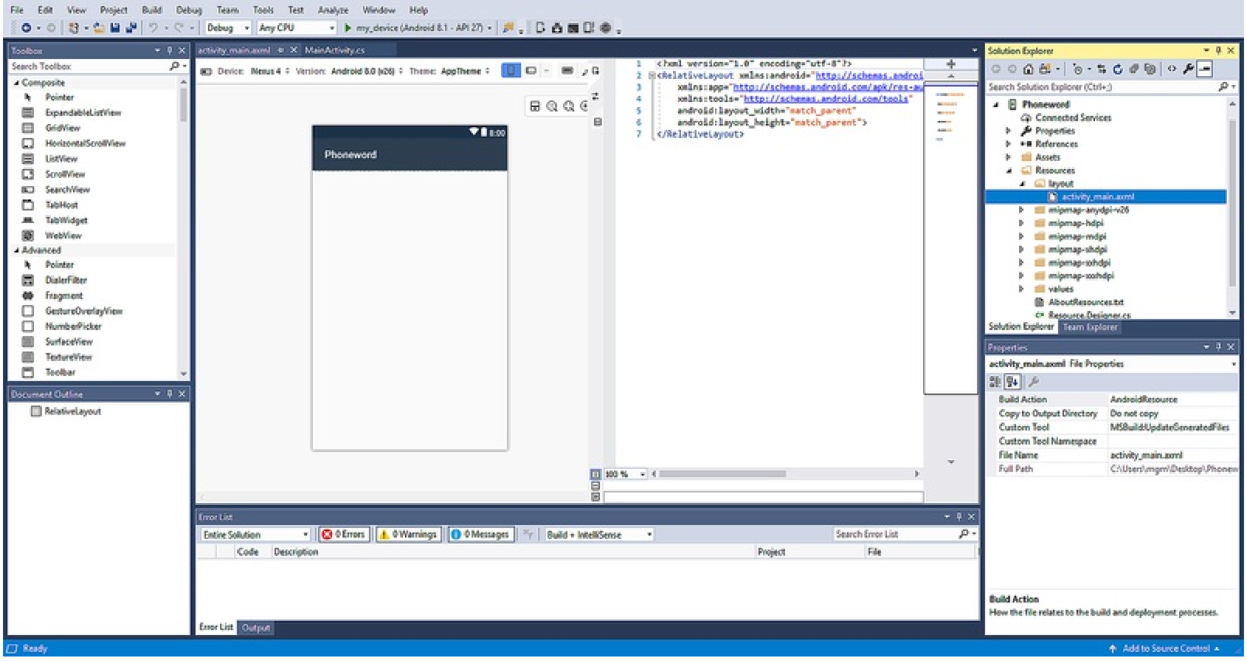
Create a layout

TIP

Newer releases of Visual Studio support opening .xml files inside the Android Designer.

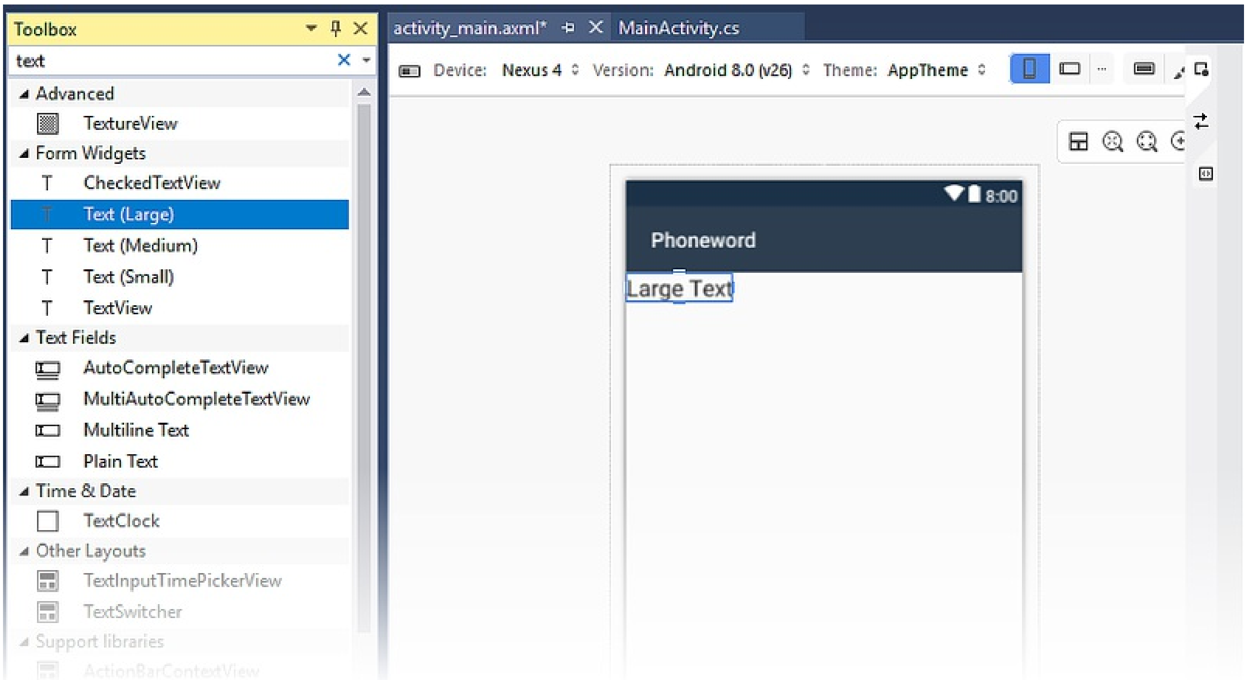
Both .axml and .xml files are supported in the Android Designer.

After the new project is created, expand the **Resources** folder and then the **layout** folder in the **Solution Explorer**. Double-click **activity_main.axml** to open it in the Android Designer. This is the layout file for the app's screen:

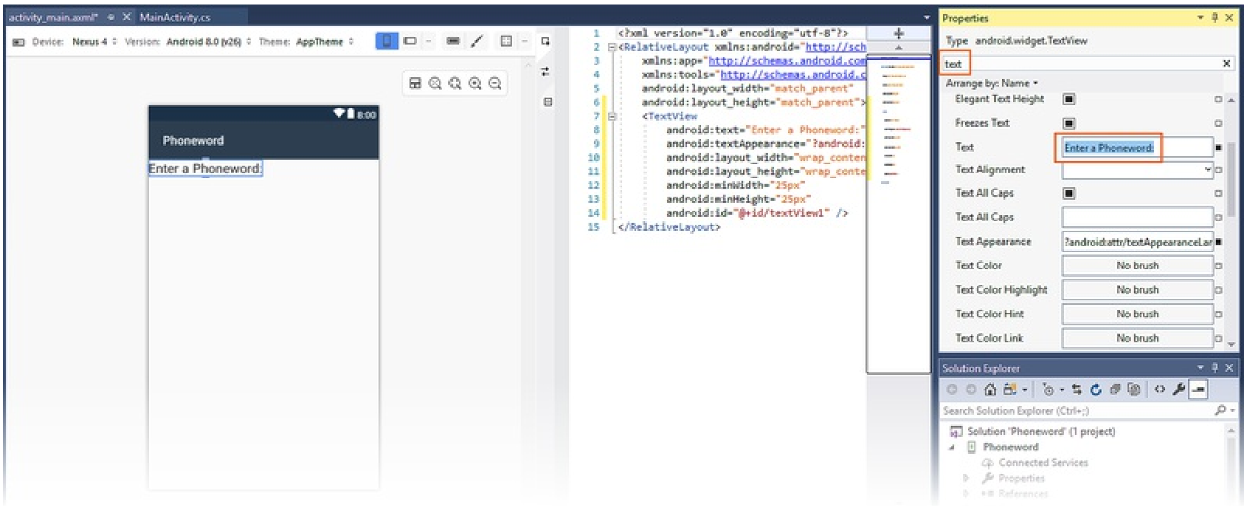


> **TIP**
>
> Newer releases of Visual Studio contain a slightly different app template.
>
> 1. Instead of **activity_main.axml**, the layout is in **content_main.axml**.
> 2. The default layout will be a `RelativeLayout`. For the rest of the steps on this page to work you should change the `<RelativeLayout>` tag to `<LinearLayout>` and add another attribute `android:orientation="vertical"` to the `LinearLayout` opening tag.

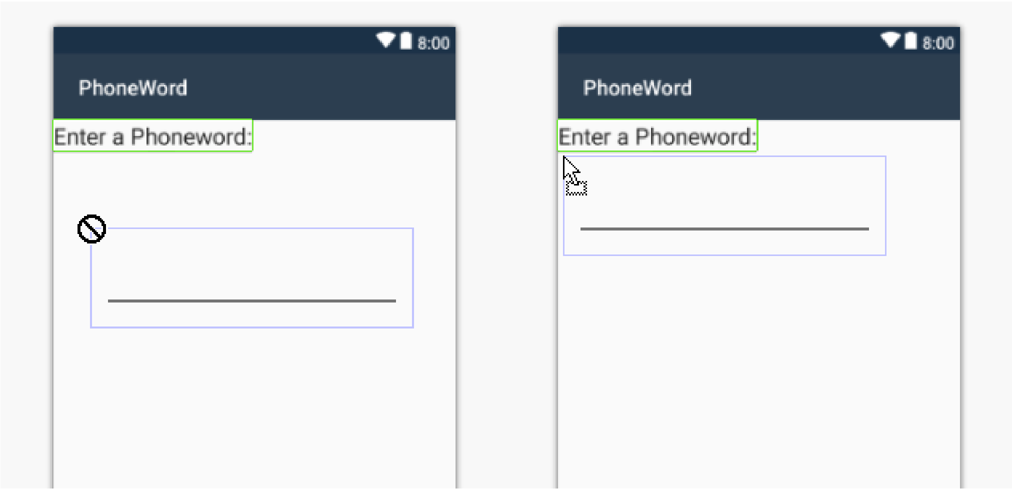
From the **Toolbox** (the area on the left), enter `text` into the search field and drag a **Text (Large)** widget onto the design surface (the area in the center):



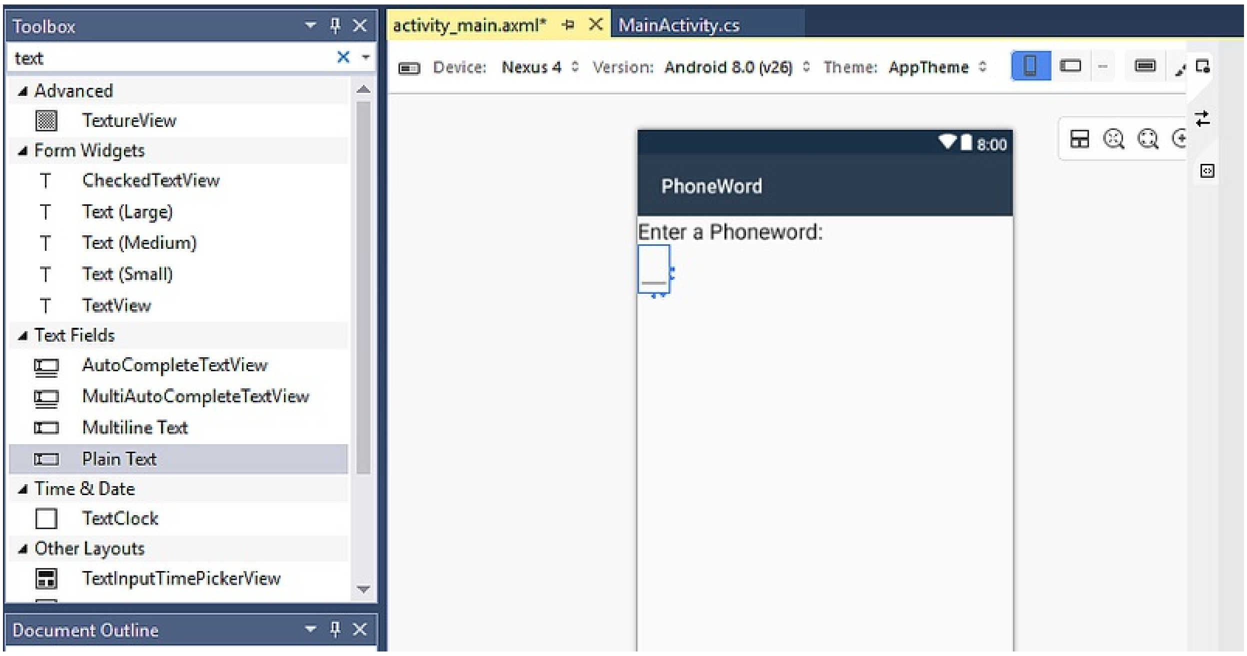
With the **Text (Large)** control selected on the design surface, use the **Properties** pane to change the `Text` property of the **Text (Large)** widget to `Enter a Phoneword:`:



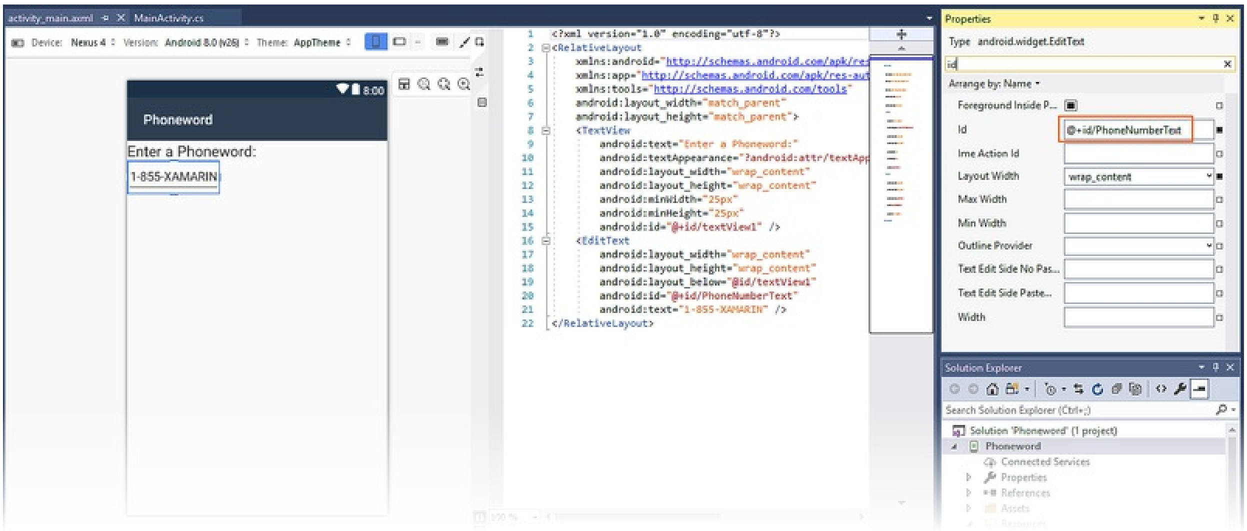
Drag a **Plain Text** widget from the **Toolbox** to the design surface and place it underneath the **Text (Large)** widget. Placement of the widget will not occur until you move the mouse pointer to a place in the layout that can accept the widget. In the screenshots below, the widget cannot be placed (as seen on the left) until the mouse pointer is moved just below the previous `TextView` (as shown on the right):



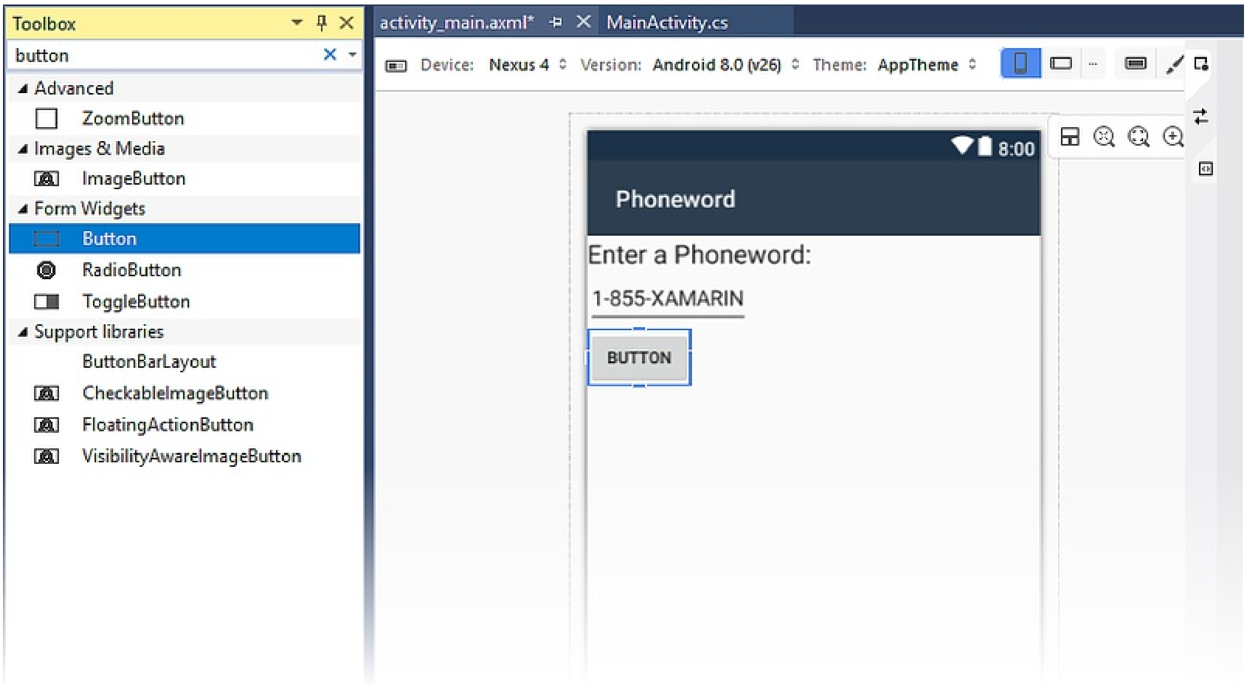
When the **Plain Text** (an `EditText` widget) is placed correctly, it will appear as illustrated in the following screenshot:



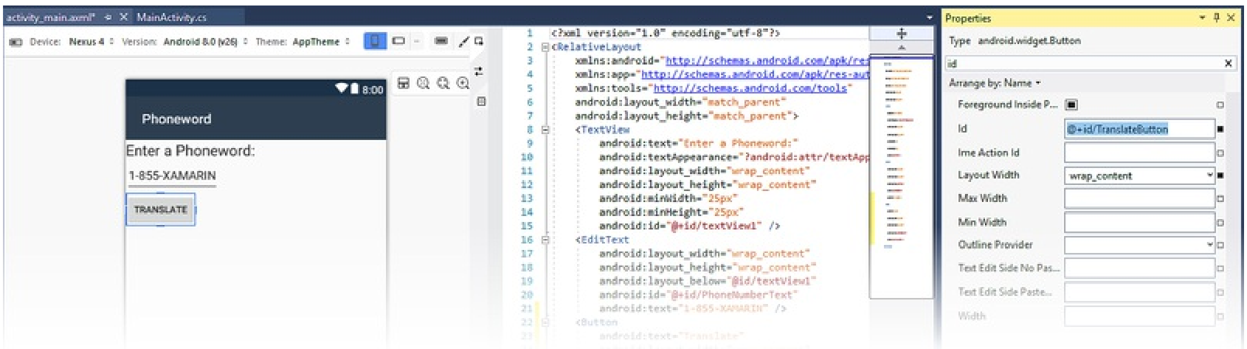
With the **Plain Text** widget selected on the design surface, use the **Properties** pane to change the `Id` property of the **Plain Text** widget to `@+id/PhoneNumberText` and change the `Text` property to `1-855-XAMARIN`:



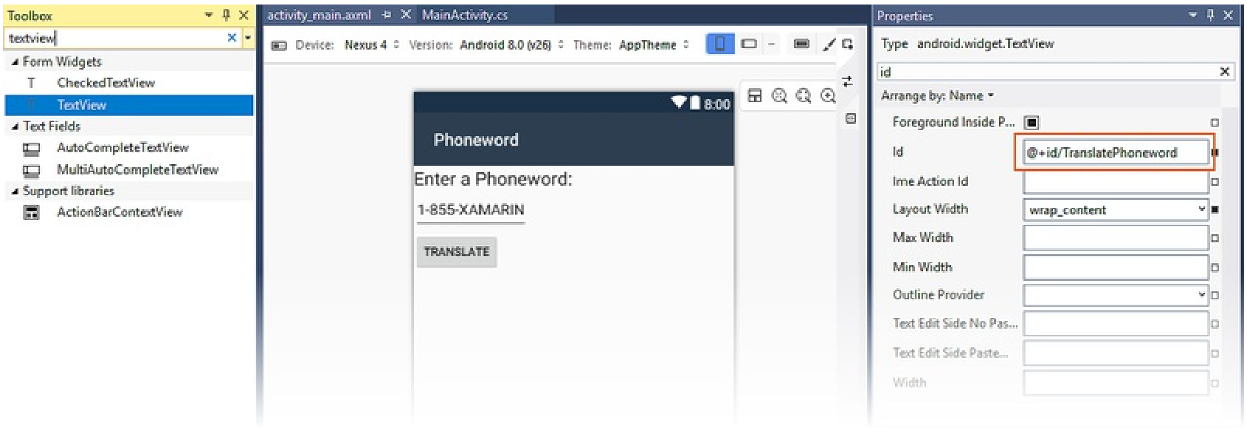
Drag a **Button** from the **Toolbox** to the design surface and place it underneath the **Plain Text** widget:



With the **Button** selected on the design surface, use the **Properties** pane to change its `Text` property to `Translate` and its `Id` property to `@+id/TranslateButton`:



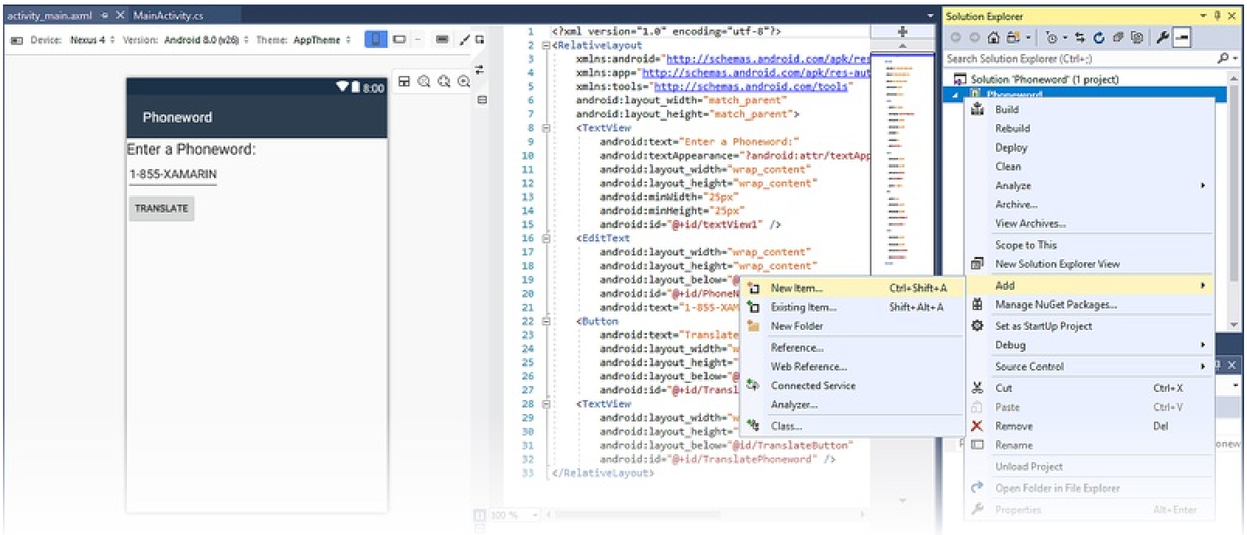
Drag a **TextView** from the **Toolbox** to the design surface and place it under the **Button** widget. Change the `Text` property of the **TextView** to an empty string and set its `Id` property to `@+id/TranslatedPhoneword`:



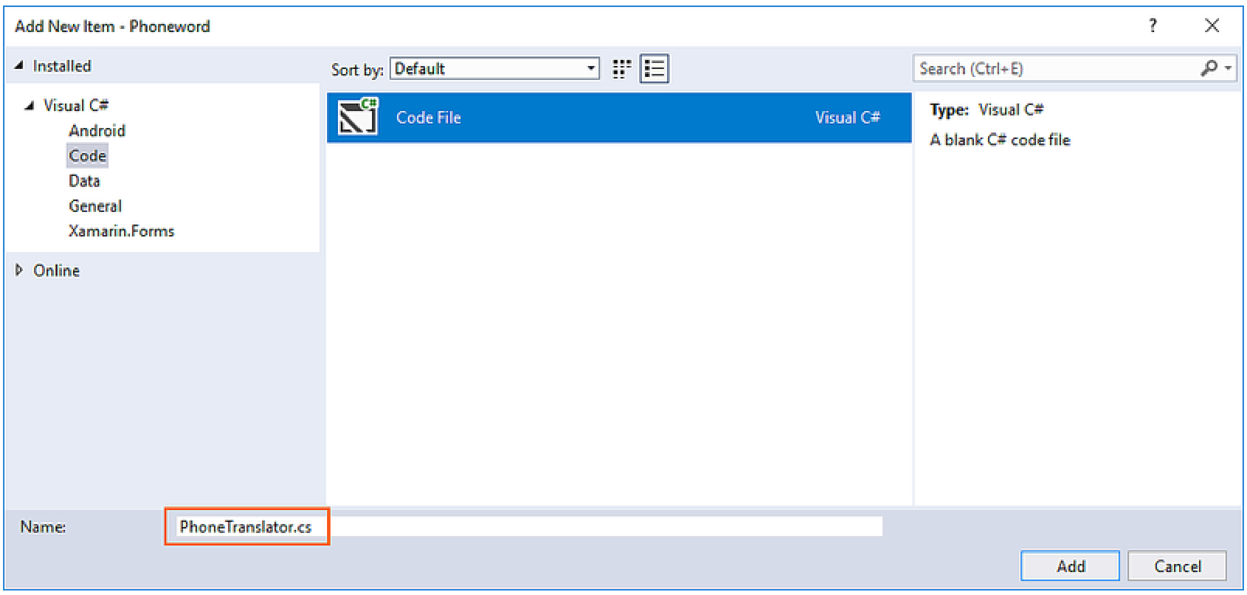
Save your work by pressing **CTRL+S**.

Write some code

The next step is to add some code to translate phone numbers from alphanumeric to numeric. Add a new file to the project by right-clicking the **Phoneword** project in the **Solution Explorer** pane and choosing **Add > New Item...** as shown below:



In the **Add New Item** dialog, select **Visual C# > Code > Code File** and name the new code file **PhoneTranslator.cs**:



This creates a new empty C# class. Insert the following code into this file:

```
using System.Text;
using System;
namespace Core
{
    public static class PhonewordTranslator
    {
        public static string ToNumber(string raw)
        {
            if (string.IsNullOrWhiteSpace(raw))
                return "";
            else
                raw = raw.ToUpperInvariant();

            var newNumber = new StringBuilder();
            foreach (var c in raw)
            {
                if (" -0123456789".Contains(c))
                {
                    newNumber.Append(c);
                }
                else
                {
                    var result = TranslateToNumber(c);
                    if (result != null)
                        newNumber.Append(result);
                }
                // otherwise we've skipped a non-numeric char
            }
            return newNumber.ToString();
        }
        static bool Contains (this string keyString, char c)
        {
            return keyString.IndexOf(c) >= 0;
        }
        static int? TranslateToNumber(char c)
        {
            if ("ABC".Contains(c))
                return 2;
            else if ("DEF".Contains(c))
                return 3;
            else if ("GHI".Contains(c))
                return 4;
            else if ("JKL".Contains(c))
                return 5;
            else if ("MNO".Contains(c))
                return 6;
            else if ("PQRS".Contains(c))
                return 7;
            else if ("TUV".Contains(c))
                return 8;
            else if ("WXYZ".Contains(c))
                return 9;
            return null;
        }
    }
}
```

Save the changes to the **PhoneTranslator.cs** file by clicking **File > Save** (or by pressing **CTRL+S**), then close the file.

Wire up the user interface

The next step is to add code to wire up the user interface by inserting backing code into the `MainActivity` class. Begin by wiring up the **Translate** button. In the `MainActivity` class, find the `OnCreate` method. The next step is to add the button code inside `OnCreate`, below the `base.OnCreate(savedInstanceState)` and `SetContentView(Resource.Layout.activity_main)` calls. First, modify the template code so that the `OnCreate` method resembles the following:

```
using Android.App;
using Android.OS;
using Android.Support.V7.App;
using Android.Runtime;
using Android.Widget;

namespace Phoneword
{
    [Activity(Label = "@string/app_name", Theme = "@style/AppTheme", MainLauncher = true)]
    public class MainActivity : AppCompatActivity
    {
        protected override void OnCreate(Bundle savedInstanceState)
        {
            base.OnCreate(savedInstanceState);

            // Set our view from the "main" layout resource
            SetContentView(Resource.Layout.activity_main);

            // New code will go here
        }
    }
}
```

Get a reference to the controls that were created in the layout file via the Android Designer. Add the following code inside the `OnCreate` method, after the call to `SetContentView`:

```
// Get our UI controls from the loaded layout
EditText phoneNumberText = FindViewById<EditText>(Resource.Id.PhoneNumberText);
TextView translatedPhoneWord = FindViewById<TextView>(Resource.Id.TranslatedPhoneword);
Button translateButton = FindViewById<Button>(Resource.Id.TranslateButton);
```

Add code that responds to user presses of the **Translate** button. Add the following code to the `OnCreate` method (after the lines added in the previous step):

```
// Add code to translate number
translateButton.Click += (sender, e) =>
{
    // Translate user's alphanumeric phone number to numeric
    string translatedNumber = Core.PhonewordTranslator.ToNumber(phoneNumberText.Text);
    if (string.IsNullOrWhiteSpace(translatedNumber))
    {
        translatedPhoneWord.Text = string.Empty;
    }
    else
    {
        translatedPhoneWord.Text = translatedNumber;
    }
};
```

Save your work by selecting **File > Save All** (or by pressing **CTRL-SHIFT-S**) and build the application by selecting **Build > Rebuild Solution** (or by pressing **CTRL-SHIFT-B**).

If there are errors, go through the previous steps and correct any mistakes until the application builds successfully. If you get a build error such as, *Resource does not exist in the current context*, verify that the namespace name in **MainActivity.cs** matches the project name (`Phoneword`) and then completely rebuild the solution. If you still get build errors, verify that you have installed the latest Visual Studio updates.

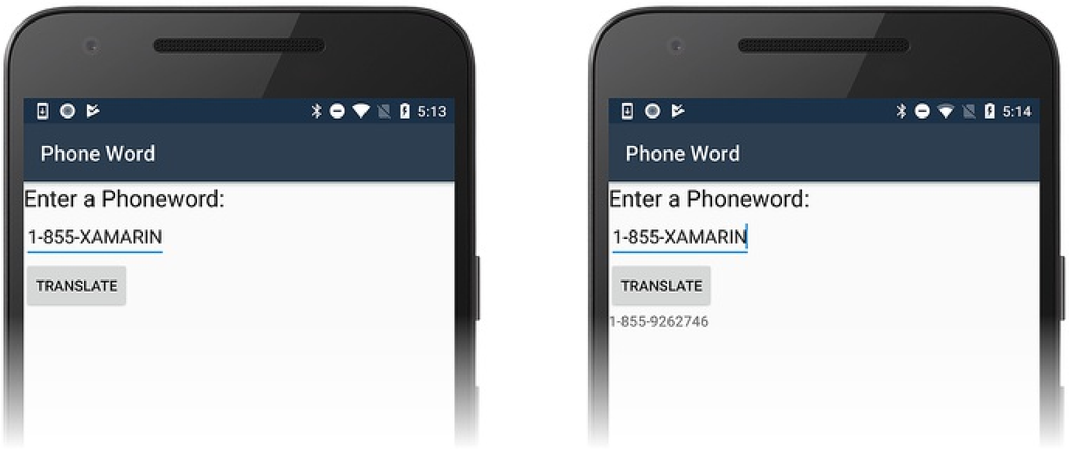
Set the app name

You should now have a working application – it's time to set the name of the app. Expand the **values** folder (inside the **Resources** folder) and open the file **strings.xml**. Change the app name string to `Phone Word` as shown here:

```
<resources>
    <string name="app_name">Phone Word</string>
    <string name="action_settings">Settings</string>
</resources>
```

Run the app

Test the application by running it on an Android device or emulator. Tap the **TRANSLATE** button to translate **1-855-XAMARIN** into a phone number:

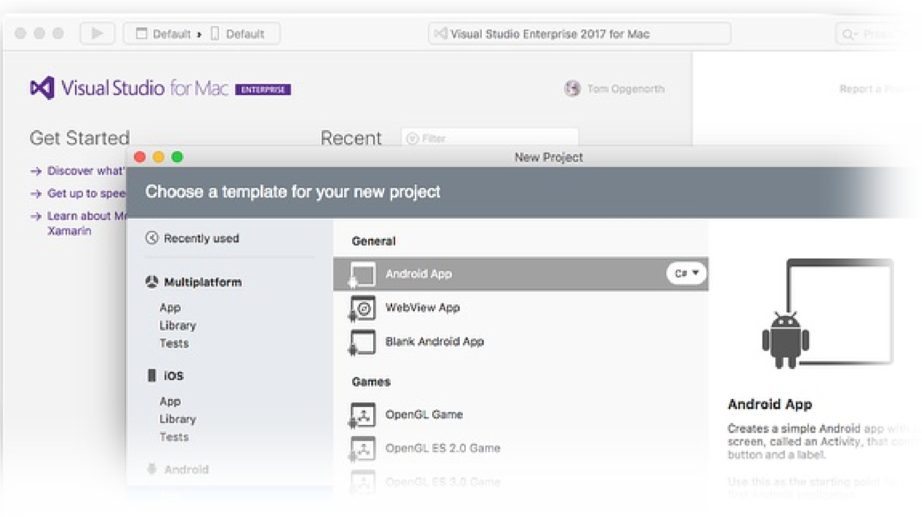


To run the app on an Android device, see how to set up your device for development.

Launch Visual Studio for Mac from the **Applications** folder or from **Spotlight**.

Click **New Project...** to create a new project.

In the **Choose a template for your new project** dialog, click **Android > App** and select the **Android App** template. Click **Next**.



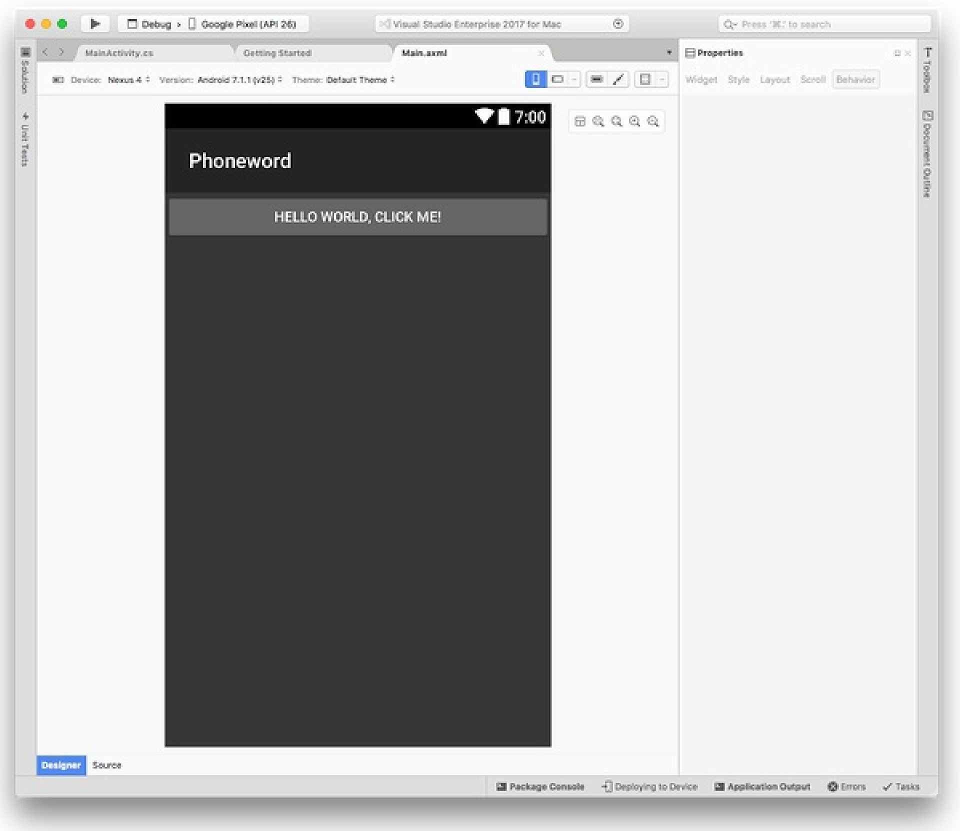
In the **Configure your Android app** dialog, name the new app `Phoneword` and click **Next**.

In the **Configure your new Android App** dialog, leave the Solution and Project names set to `Phoneword` and click **Create** to create the project.

Create a layout

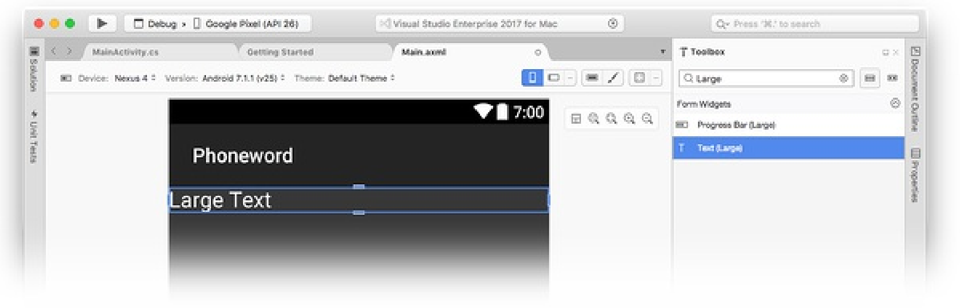
> **TIP**
>
> Newer releases of Visual Studio support opening .xml files inside the Android Designer.
>
> Both .axml and .xml files are supported in the Android Designer.

After the new project is created, expand the **Resources** folder and then the **layout** folder in the **Solution** pad. Double-click **Main.axml** to open it in the Android Designer. This is the layout file for the screen when it is viewed in the Android Designer:

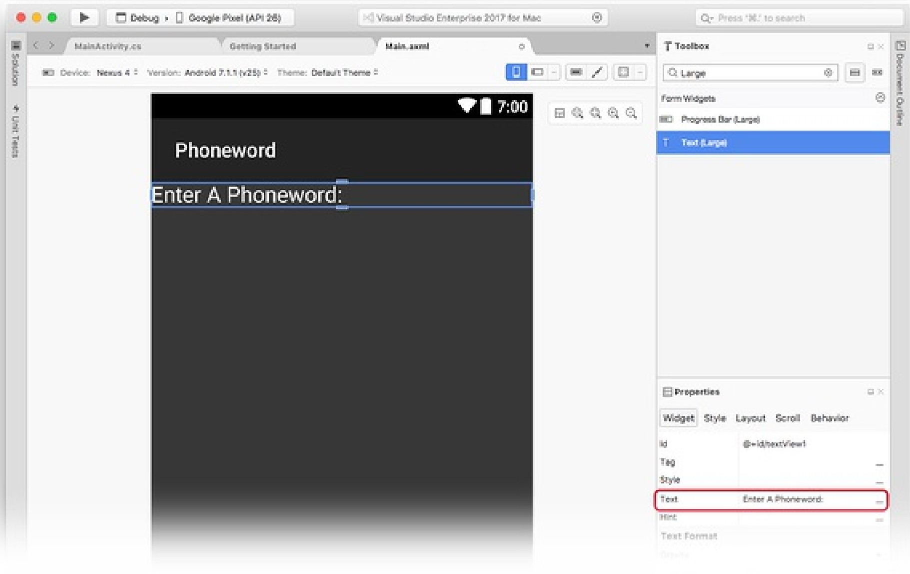


Select the **Hello World, Click Me! Button** on the design surface and press the **Delete** key to remove it.

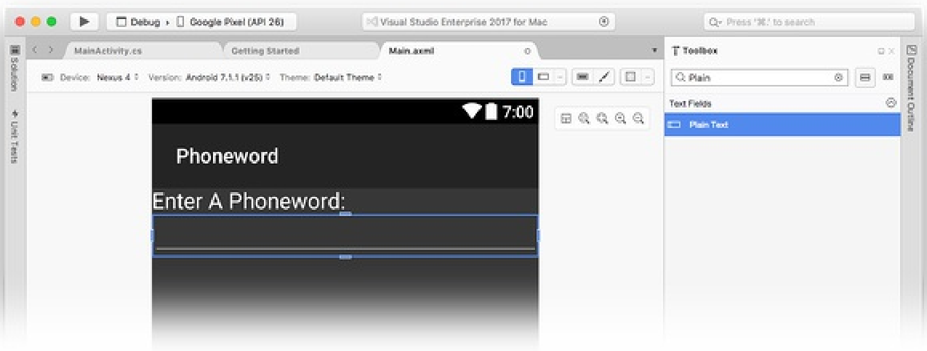
From the **Toolbox** (the area on the right), enter `text` into the search field and drag a **Text (Large)** widget onto the design surface (the area in the center):



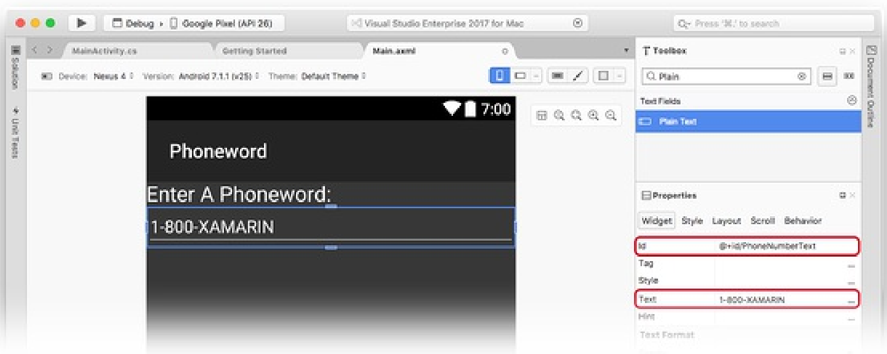
With the **Text (Large)** widget selected on the design surface, you can use the **Properties** pad to change the `Text` property of the **Text (Large)** widget to `Enter a Phoneword:` as shown below:



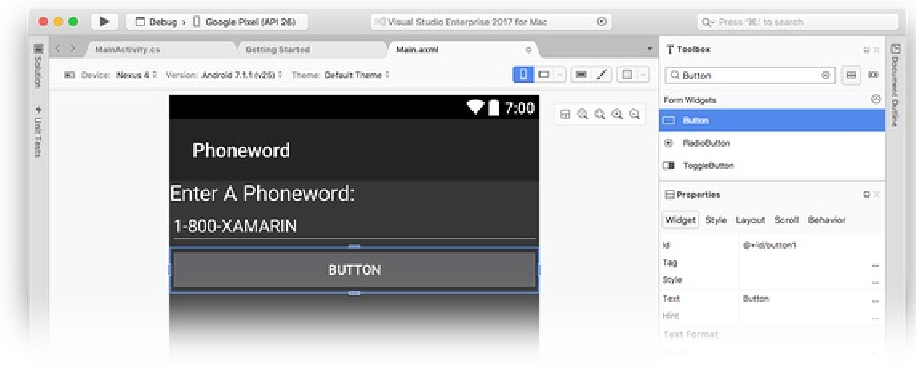
Next, drag a **Plain Text** widget from the **Toolbox** to the design surface and place it underneath the **Text (Large)** widget. Notice that you can use the search field to help locate widgets by name:



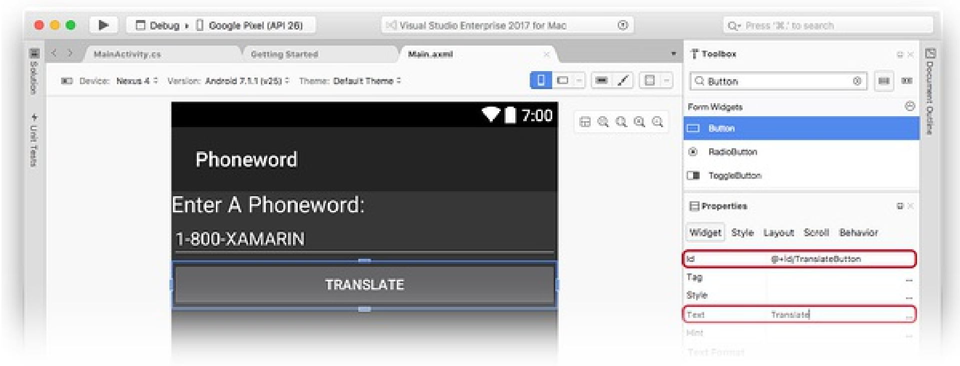
With the **Plain Text** widget selected on the design surface, you can use the **Properties** pad to change the `Id` property of the **Plain Text** widget to `@+id/PhoneNumberText` and change the `Text` property to `1-855-XAMARIN`:



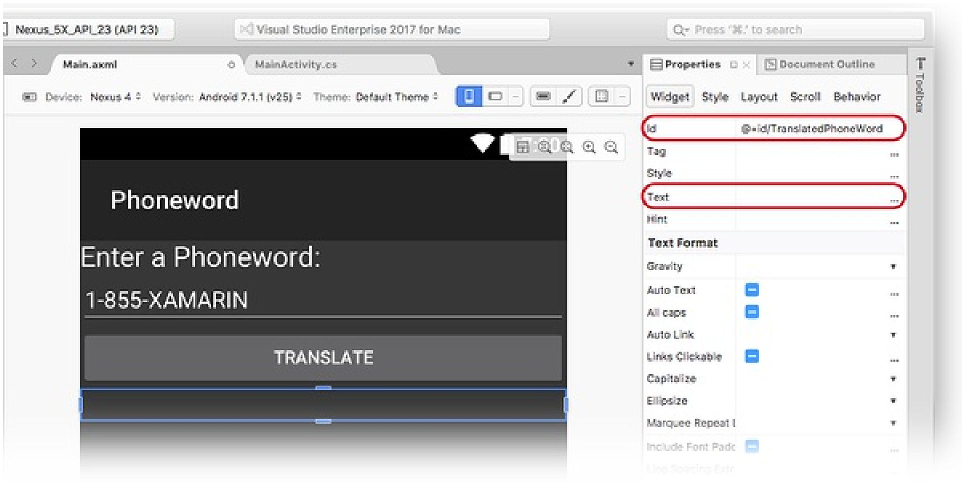
Drag a **Button** from the **Toolbox** to the design surface and place it underneath the **Plain Text** widget:



With the **Button** selected on the design surface, you can use the **Properties** pad to change the `Id` property of the **Button** to `@+id/TranslateButton` and change the `Text` property to `Translate`:



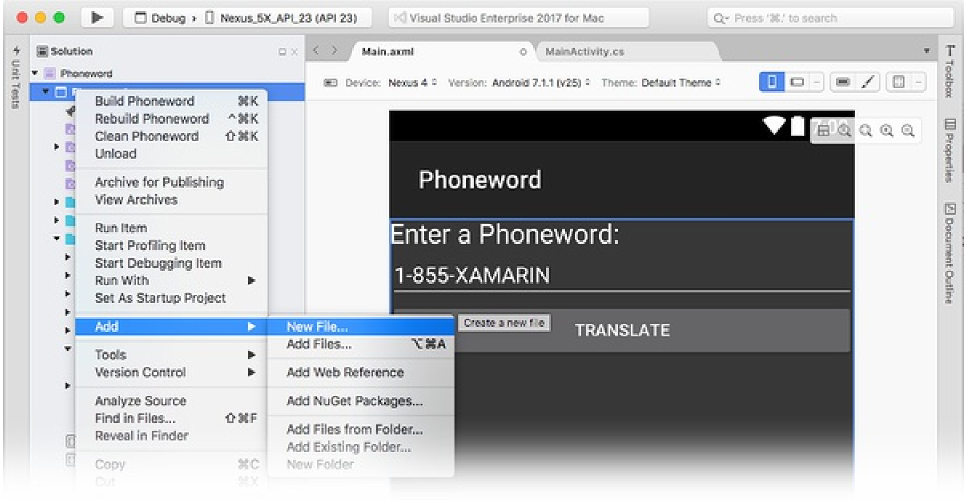
Drag a **TextView** from the **Toolbox** to the design surface and place it under the **Button** widget. With the **TextView** selected, set the `id` property of the **TextView** to `@+id/TranslatedPhoneWord` and change the `text` to an empty string:



Save your work by pressing ⌘ + **S**.

Write some code

Now, add some code to translate phone numbers from alphanumeric to numeric. Add a new file to the project by clicking the gear icon next to the **Phoneword** project in the **Solution** pad and choosing **Add > New File...**:



In the **New File** dialog, select **General > Empty Class**, name the new file **PhoneTranslator**, and click **New**. This creates a new empty C# class for us.

Remove all of the template code in the new class and replace it with the following code:

```
using System.Text;
using System;
namespace Core
{
    public static class PhonewordTranslator
    {
        public static string ToNumber(string raw)
        {
            if (string.IsNullOrWhiteSpace(raw))
                return "";
            else
                raw = raw.ToUpperInvariant();

            var newNumber = new StringBuilder();
            foreach (var c in raw)
            {
                if (" -0123456789".Contains(c))
                {
                    newNumber.Append(c);
                }
                else
                {
                    var result = TranslateToNumber(c);
                    if (result != null)
                        newNumber.Append(result);
                }
                // otherwise we've skipped a non-numeric char
            }
            return newNumber.ToString();
        }
        static bool Contains (this string keyString, char c)
        {
            return keyString.IndexOf(c) >= 0;
        }
        static int? TranslateToNumber(char c)
        {
            if ("ABC".Contains(c))
                return 2;
            else if ("DEF".Contains(c))
                return 3;
            else if ("GHI".Contains(c))
                return 4;
            else if ("JKL".Contains(c))
                return 5;
            else if ("MNO".Contains(c))
                return 6;
            else if ("PQRS".Contains(c))
                return 7;
            else if ("TUV".Contains(c))
                return 8;
            else if ("WXYZ".Contains(c))
                return 9;
            return null;
        }
    }
}
```

Save the changes to the **PhoneTranslator.cs** file by choosing **File > Save** (or by pressing **⌘ + S**), then close the file. Ensure that there are no compile-time errors by rebuilding the solution.

Wire up the user interface

The next step is to add code to wire up the user interface by adding the backing code into the `MainActivity` class. Double-click **MainActivity.cs** in the **Solution Pad** to open it.

Begin by adding an event handler to the **Translate** button. In the `MainActivity` class, find the `OnCreate` method. Add the button code inside `OnCreate`, below the `base.OnCreate(bundle)` and `SetContentView (Resource.Layout.Main)` calls. Remove any existing button handling code (i.e., code that references `Resource.Id.myButton` and creates a click handler for it) so that the `OnCreate` method resembles the following:

```
using System;
using Android.App;
using Android.Content;
using Android.Runtime;
using Android.Views;
using Android.Widget;
using Android.OS;

namespace Phoneword
{
    [Activity (Label = "Phone Word", MainLauncher = true)]
    public class MainActivity : Activity
    {
        protected override void OnCreate (Bundle bundle)
        {
            base.OnCreate (bundle);

            // Set our view from the "main" layout resource
            SetContentView (Resource.Layout.Main);

            // Our code will go here
        }
    }
}
```

Next, a reference is needed to the controls that were created in the layout file with the Android Designer. Add the following code inside the `OnCreate` method (after the call to `SetContentView`):

```
// Get our UI controls from the loaded layout
EditText phoneNumberText = FindViewById<EditText>(Resource.Id.PhoneNumberText);
TextView translatedPhoneWord = FindViewById<TextView>(Resource.Id.TranslatedPhoneWord);
Button translateButton = FindViewById<Button>(Resource.Id.TranslateButton);
```

Add code that responds to user presses of the **Translate** button by adding the following code to the `OnCreate` method (after the lines added in the last step):

```
// Add code to translate number
string translatedNumber = string.Empty;

translateButton.Click += (sender, e) =>
{
    // Translate user's alphanumeric phone number to numeric
    translatedNumber = PhonewordTranslator.ToNumber(phoneNumberText.Text);
    if (string.IsNullOrWhiteSpace(translatedNumber))
    {
        translatedPhoneWord.Text = string.Empty;
    }
    else
    {
        translatedPhoneWord.Text = translatedNumber;
    }
};
```

Save your work and build the application by selecting **Build > Build All** (or by pressing ⌘ + B). If the application compiles, you will get a success message at the top of Visual Studio for Mac:

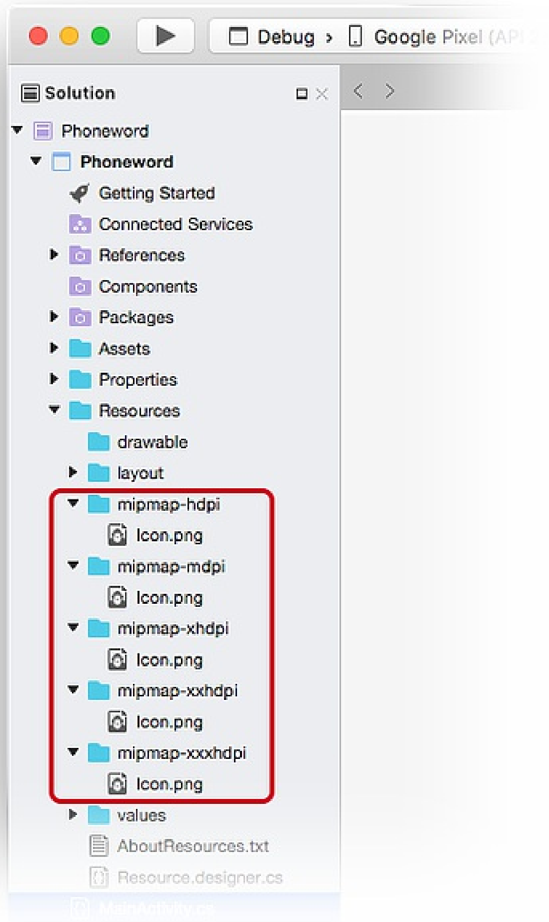
If there are errors, go through the previous steps and correct any mistakes until the application builds successfully. If you get a build error such as, *Resource does not exist in the current context*, verify that the namespace name in **MainActivity.cs** matches the project name (`Phoneword`) and then completely rebuild the solution. If you still get build errors, verify that you have installed the latest Xamarin.Android and Visual Studio for Mac updates.

Set the label and app icon

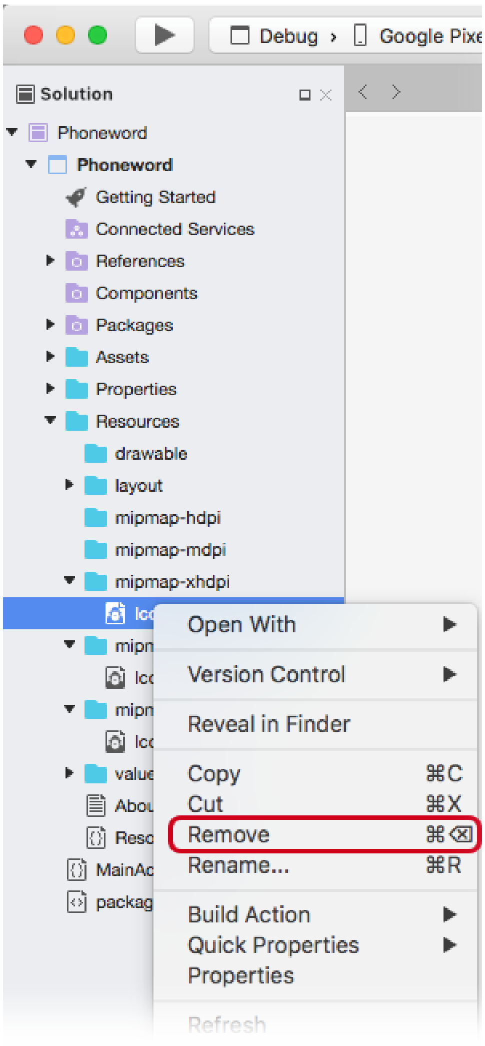
Now that you have a working application, it's time to add the finishing touches! Start by editing the `Label` for `MainActivity`. The `Label` is what Android displays at the top of the screen to let users know where they are in the application. At the top of the `MainActivity` class, change the `Label` to `Phone Word` as shown here:

```
namespace Phoneword
{
    [Activity (Label = "Phone Word", MainLauncher = true)]
    public class MainActivity : Activity
    {
        ...
    }
}
```

Now it's time to set the application icon. By default, Visual Studio for Mac will provide a default icon for the project. Delete these files from the solution, and replace them with a different icon. Expand the **Resources** folder in the **Solution Pad**. Notice that there are five folders that are prefixed with **mipmap-**, and that each of these folders contains a single **Icon.png** file:

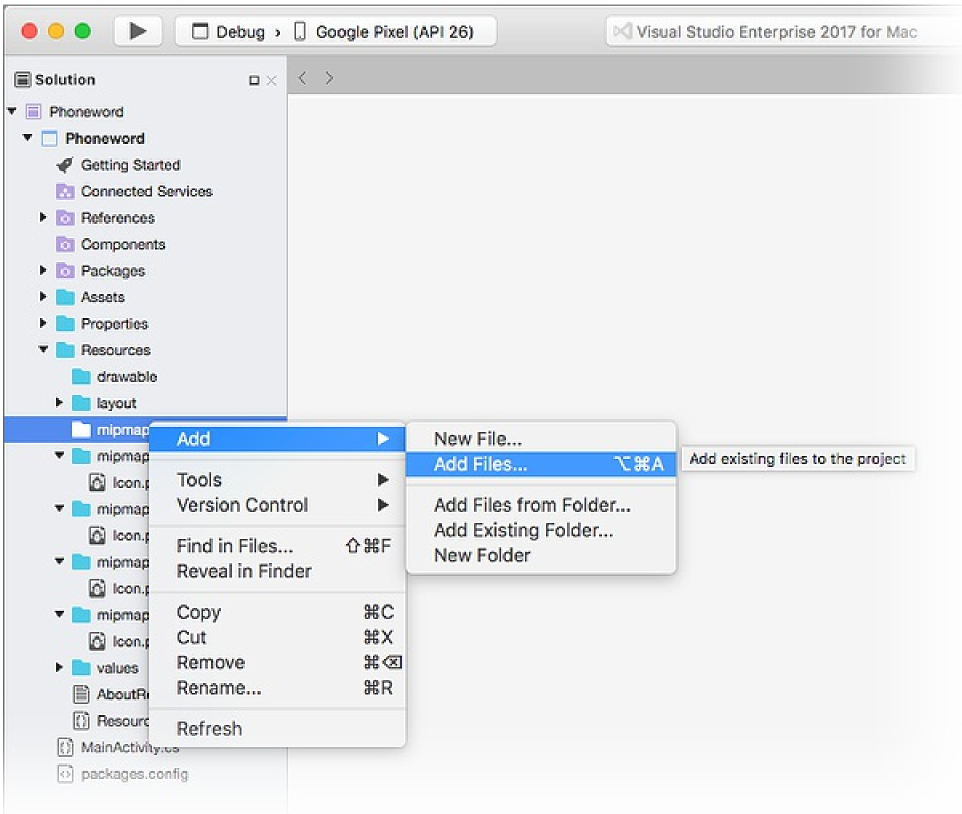


It is necessary to delete each of these icon files from the project. Right click on each of **Icon.png** files, and select **Remove** from the context menu:



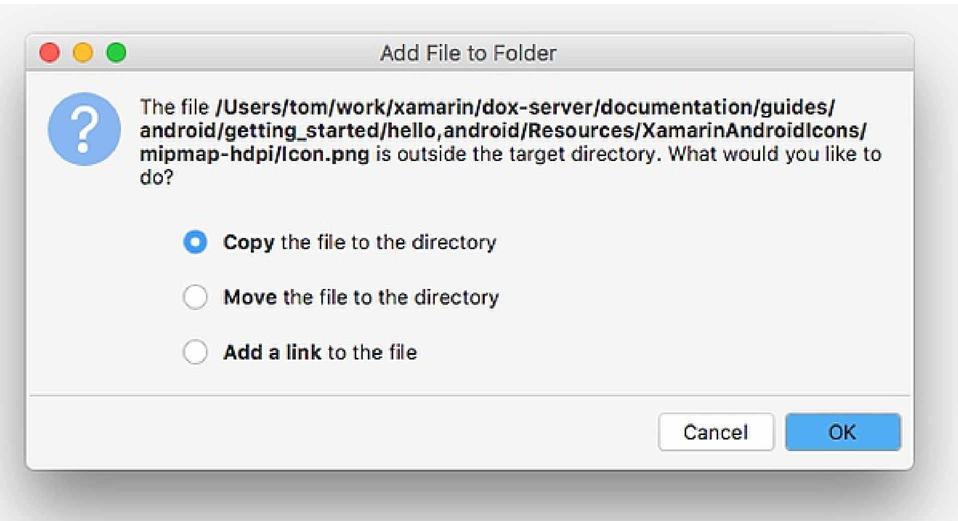
Click on the **Delete** button in the dialog.

Next, download and unzip Xamarin App Icons set. This zip file holds the icons for the application. Each icon is visually identical but at different resolutions it renders correctly on different devices with different screen densities. The set of files must be copied into the Xamarin.Android project. In Visual Studio for Mac, in the **Solution Pad**, right-click the **mipmap-hdpi** folder and select **Add > Add Files**:



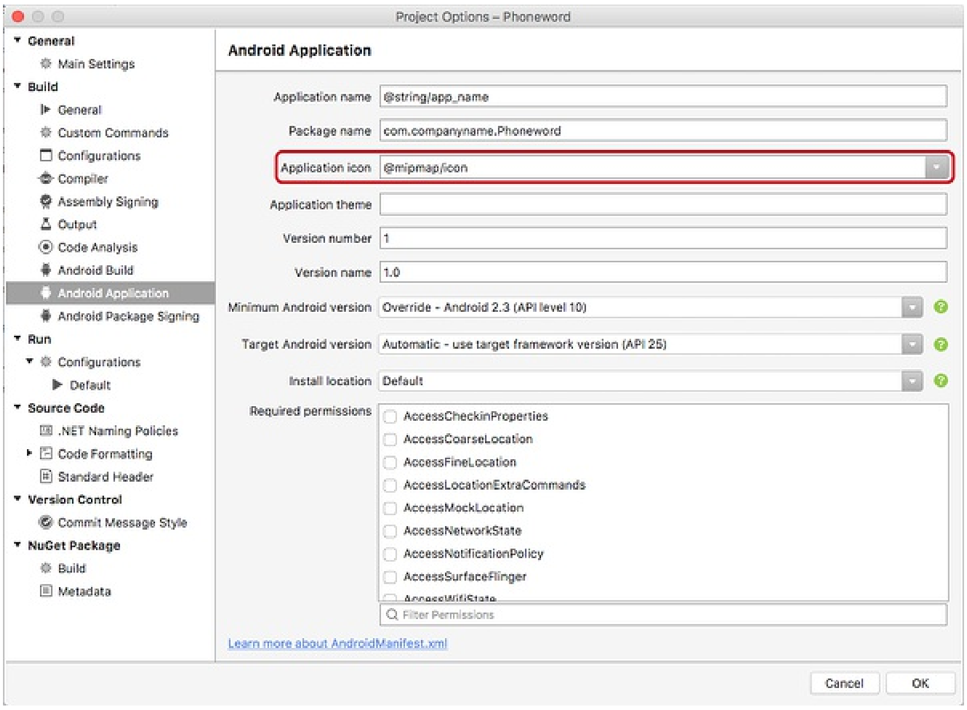
From the selection dialog, navigate to the unzipped Xamarin AdApp Icons directory and open the **mipmap-hdpi** folder. Select **Icon.png** and click **Open**.

In the **Add File to Folder** dialog box, select **Copy the file into the directory** and click **OK**:



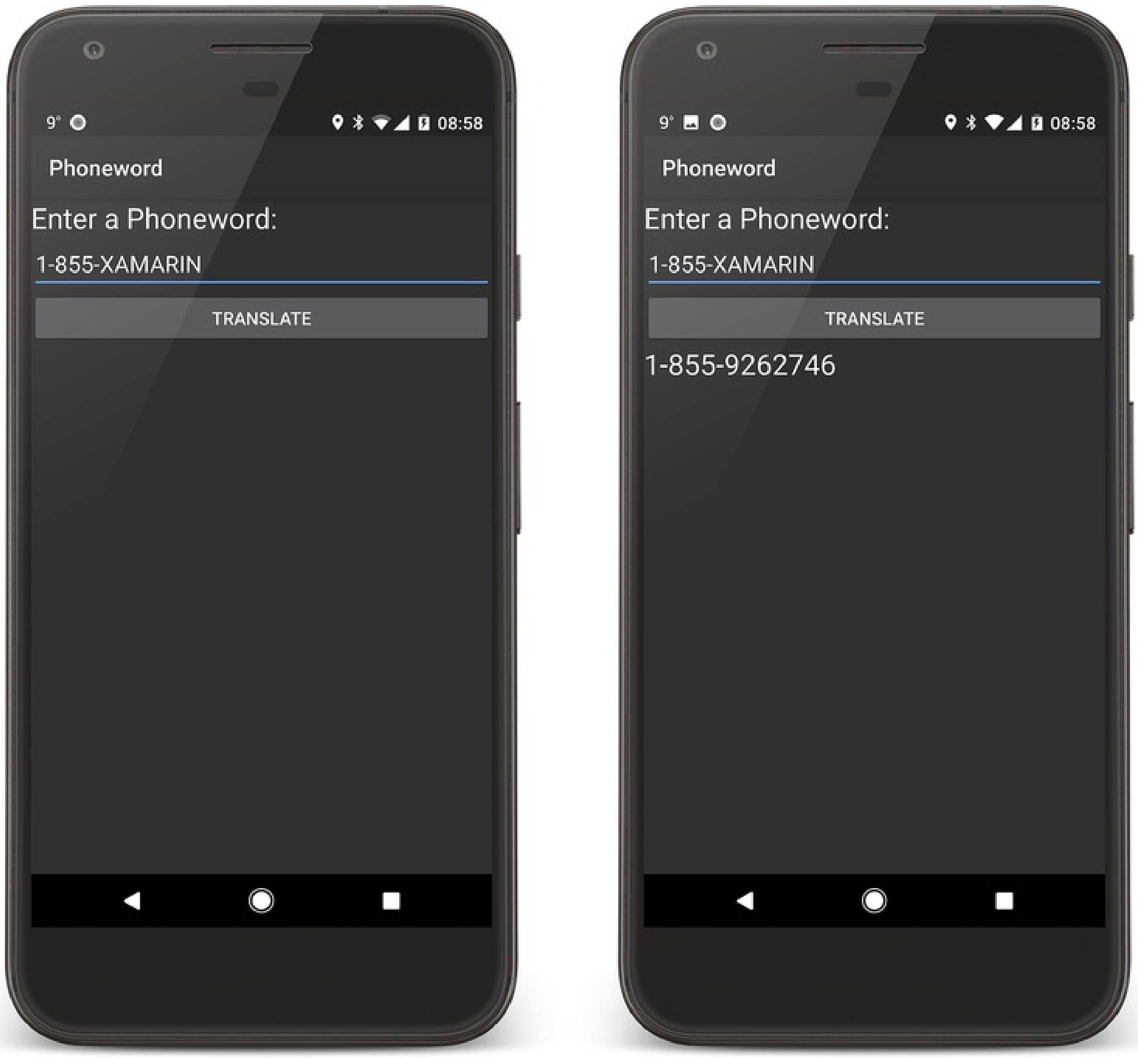
Repeat these steps for each of the **mipmap-** folders until the contents of the **mipmap-** Xamarin App Icons folders are copied to their counterpart **mipmap-** folders in the **Phoneword** project.

After all the icons are copied to the Xamarin.Android project, open the **Project Options** dialog by right clicking on the project in the **Solution Pad**. Select **Build > Android Application** and select `@mipmap/icon` from the **Application icon** combo box:



Run the app

Finally, test the application by running it on an Android device or emulator and translating a Phoneword:



To run the app on an Android device, see how to set up your device for development.

Congratulations on completing your first Xamarin.Android application! Now it's time to dissect the tools and skills you have just learned. Next up is the Hello, Android Deep Dive.

Related links

- Xamarin Android App Icons (ZIP)
- Phoneword (sample)

Hello, Android: Deep Dive

7/8/2021 • 17 minutes to read • Edit Online

In this two-part guide, you'll build your first Xamarin.Android application and develop an understanding of the fundamentals of Android application development with Xamarin. Along the way, you will be introduced to the tools, concepts, and steps required to build and deploy a Xamarin.Android application.

In the Hello, Android Quickstart, you built and ran your first Xamarin.Android application. Now it's time to develop a deeper understanding of how Android applications work so that you can build more sophisticated programs. This guide reviews the steps that you took in the Hello, Android walkthrough so that you can understand what you did and begin to develop a fundamental understanding of Android application development.

This guide will touch upon the following topics:

- **Introduction to Visual Studio** – Introduction to Visual Studio and creating a new Xamarin.Android application.
- **Anatomy of a Xamarin.Android Application** - Tour of the essential parts of a Xamarin.Android application.
- **App Fundamentals and Architecture Basics** – Introduction to Activities, the Android Manifest, and the general flavor of Android development.
- **User Interface (UI)** – Creating user interfaces with the Android Designer.
- **Activities and the Activity Lifecycle** – An introduction to the Activity Lifecycle and wiring up the user interface in code.
- **Testing, Deployment, and Finishing Touches** – Complete your application with advice on testing, deployment, generating artwork, and more.
- **Introduction to Visual Studio for Mac** – Introduction to Visual Studio for Mac and creating a new Xamarin.Android application.
- **Anatomy of a Xamarin.Android Application** – Tour of the essential parts of a Xamarin.Android application.
- **App Fundamentals and Architecture Basics** – Introduction to Activities, the Android Manifest, and the general flavor of Android development.
- **User Interface (UI)** – Creating user interfaces with the Android Designer.
- **Activities and the Activity Lifecycle** – An introduction to the Activity Lifecycle and wiring up the user interface in code.
- **Testing, Deployment, and Finishing Touches** – Complete your application with advice on testing, deployment, generating artwork, and more.

This guide helps you develop the skills and knowledge required to build a single-screen Android application. After you work through it, you should understand the different parts of a Xamarin.Android application and how they fit together.

Introduction to Visual Studio

Visual Studio is a powerful IDE from Microsoft. It features a fully integrated visual designer, a text editor that includes refactoring tools, an assembly browser, source code integration, and more. In this guide you'll learn to use some basic Visual Studio features with the Xamarin plug-in.

Visual Studio organizes code into *Solutions* and *Projects*. A Solution is a container that can hold one or more Projects. A Project can be an application (such as for iOS or Android), a supporting library, a test application, and more. In the **Phoneword** app, you added a new Android Project using the **Android Application** template to the **Phoneword** Solution created in the Hello, Android guide.

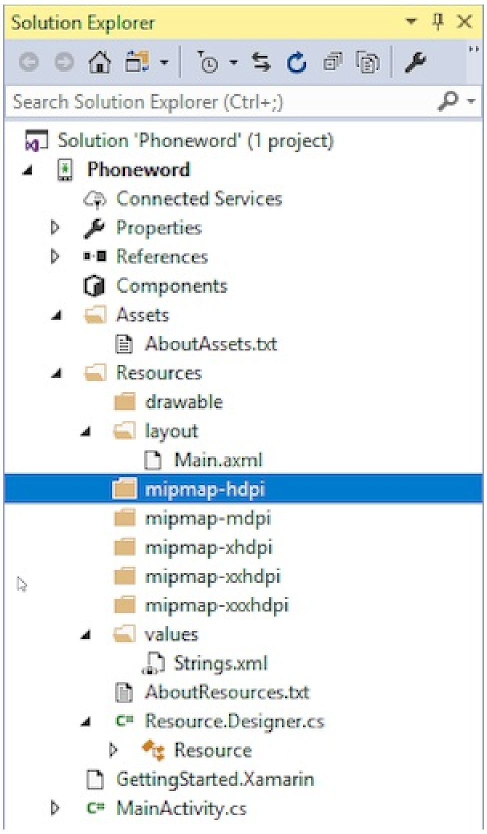
Introduction to Visual Studio for Mac

Visual Studio for Mac is a free, open-source IDE similar to Visual Studio. It features a fully integrated visual designer, a text editor complete with refactoring tools, an assembly browser, source code integration, and more. In this guide, you'll learn to use some basic Visual Studio for Mac features. If you're new to Visual Studio for Mac, you may want to check out the more in-depth Introduction to Visual Studio for Mac.

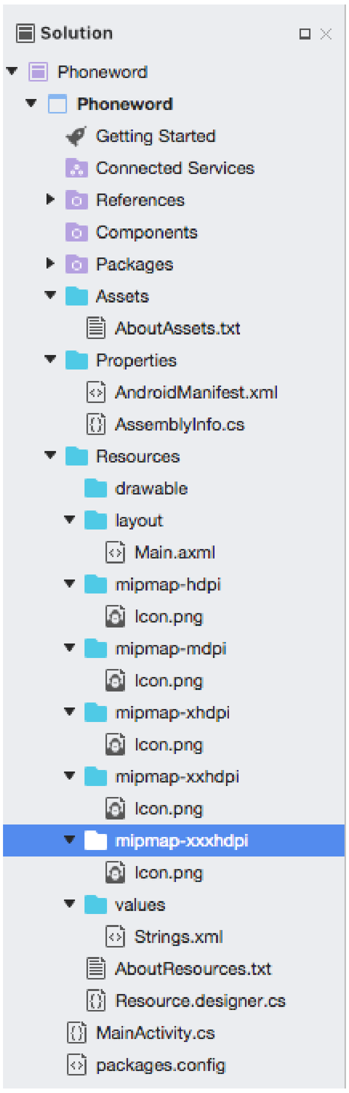
Visual Studio for Mac follows the Visual Studio practice of organizing code into *Solutions* and *Projects*. A Solution is a container that can hold one or more Projects. A Project can be an application (such as for iOS or Android), a supporting library, a test application, and more. In the **Phoneword** app, you added a new Android Project using the **Android Application** template to the **Phoneword** Solution created in the Hello, Android guide.

Anatomy of a Xamarin.Android application

The following screenshot lists the Solution's contents. This is the Solution Explorer, which contains the directory structure and all of the files associated with the Solution:



The following screenshot lists the Solution's contents. This is the Solution Pad, which contains the directory structure and all of the files associated with the Solution:



A Solution called **Phoneword** was created and the Android project **Phoneword** was placed inside of it.

Look at the items inside the Project to see each folder and its purpose:

- **Properties** – Contains the AndroidManifest.xml file that describes all of the requirements for the Xamarin.Android application, including name, version number, and permissions. The **Properties** folder also houses AssemblyInfo.cs, a .NET assembly metadata file. It is a good practice to fill this file with some basic information about your application.

- **References** – Contains the assemblies required to build and run the application. If you expand the References directory, you'll see references to .NET assemblies such as System, System.Core, and System.Xml, as well as a reference to Xamarin's Mono.Android assembly.

- **Assets** – Contains the files the application needs to run including fonts, local data files, and text files. Files included here are accessible through the generated `Assets` class. For more information on Android Assets, see the Xamarin Using Android Assets guide.

- **Resources** – Contains application resources such as strings, images, and layouts. You can access these resources in code through the generated `Resource` class. The Android Resources guide provides more details about the **Resources** directory. The application template also includes a concise guide to Resources in the **AboutResources.txt** file.

Resources

The **Resources** directory contains four folders named **drawable**, **layout**, **mipmap** and **values**, as well as a file named **Resource.designer.cs**.

The items are summarized in the table below:

- **drawable** – The drawable directories house drawable resources such as images and bitmaps.
- **mipmap** – The mipmap directory holds drawable files for different launcher icon densities. In the default template, the drawable directory houses the application icon file, **Icon.png**.
- **layout** – The layout directory contains *Android designer files* (.axml) that define the user interface for each screen or Activity. The template creates a default layout called **activity_main.axml**.
- **layout** – The layout directory contains *Android designer files* (.axml) that define the user interface for each screen or Activity. The template creates a default layout called **Main.axml**.
- **values** – This directory houses XML files that store simple values such as strings, integers, and colors. The template creates a file to store string values called **Strings.xml**.
- **Resource.designer.cs** – Also known as the `Resource` class, this file is a partial class that holds the unique IDs assigned to each resource. It is automatically created by the Xamarin.Android tools and is regenerated as necessary. This file should not be manually edited, as Xamarin.Android will overwrite any manual changes made to it.

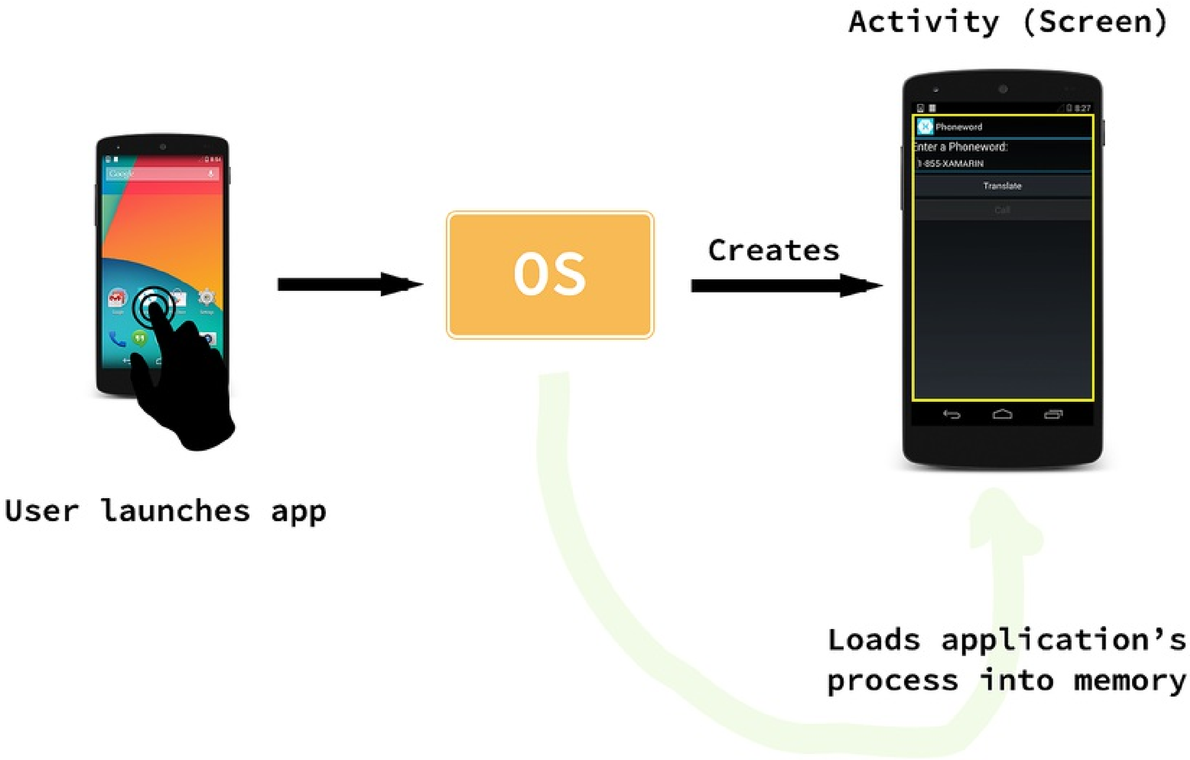
App fundamentals and architecture basics

Android applications do not have a single entry point; that is, there is no single line of code in the application that the operating system calls to start the application. Instead, an application starts when Android instantiates one of its classes, during which time Android loads the entire application's process into memory.

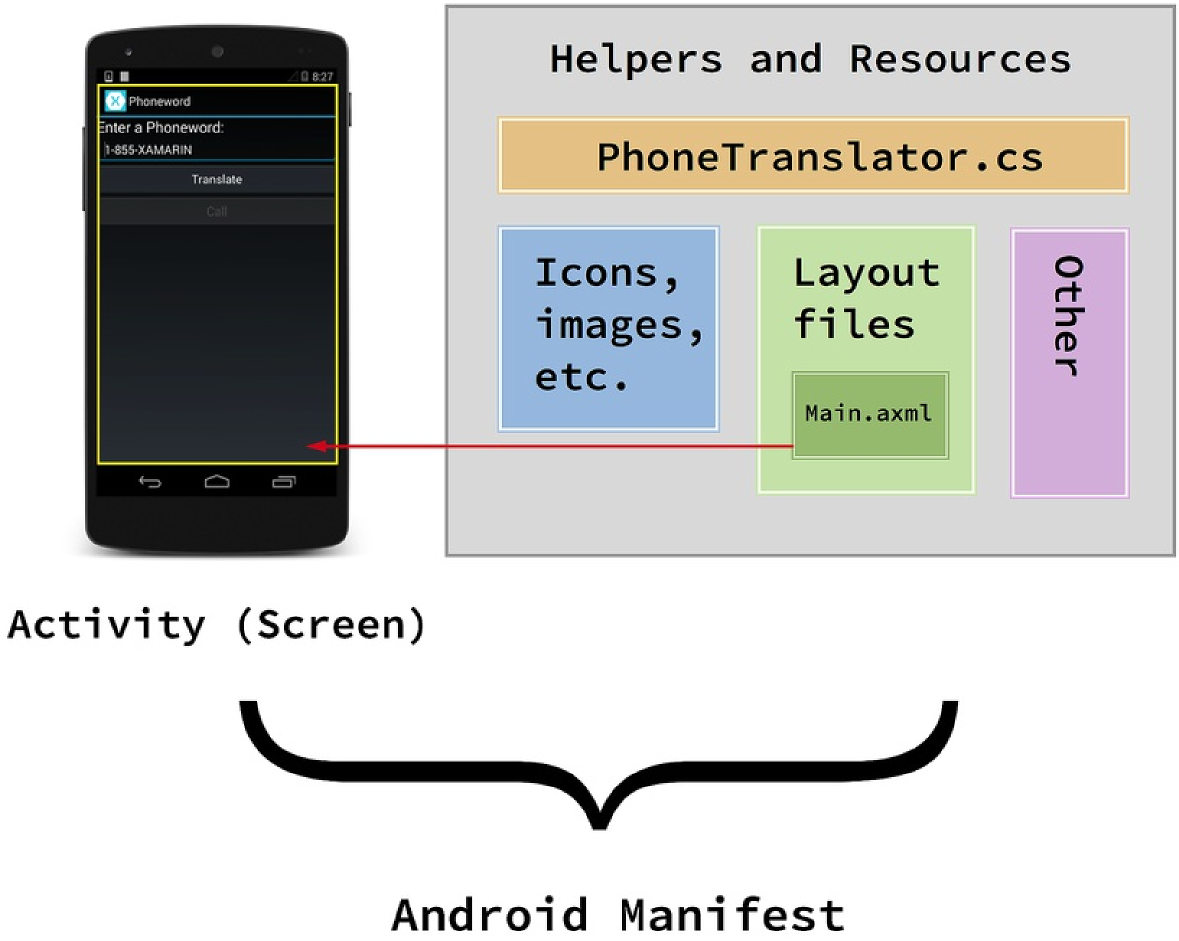
This unique feature of Android can be extremely useful when designing complicated applications or interacting with the Android operating system. However, these options also make Android complex when dealing with a basic scenario like the **Phoneword** application. For this reason, exploration of Android architecture is split in two. This guide dissects an application that uses the most common entry point for an Android app: the first screen. In Hello, Android Multiscreen, the full complexities of Android architecture are explored as different ways to launch an application are discussed.

Phoneword scenario - starting with an activity

When you open the **Phoneword** application for the first time in an emulator or device, the operating system creates the first *Activity*. An Activity is a special Android class that corresponds to a single application screen, and it is responsible for drawing and powering the user interface. When Android creates an application's first Activity, it loads the entire application:



Since there is no linear progression through an Android application (you can launch the application from several points), Android has a unique way of keeping track of what classes and files make up an application. In the **Phoneword** example, all the parts that make up the application are registered with a special XML file called the **Android Manifest**. The role of the **Android Manifest** is to keep track of an application's contents, properties, and permissions and to disclose them to the Android operating system. You can think of the **Phoneword** application as a single Activity (screen) and a collection of resource and helper files tied together by the Android Manifest file, as illustrated by the diagram below:

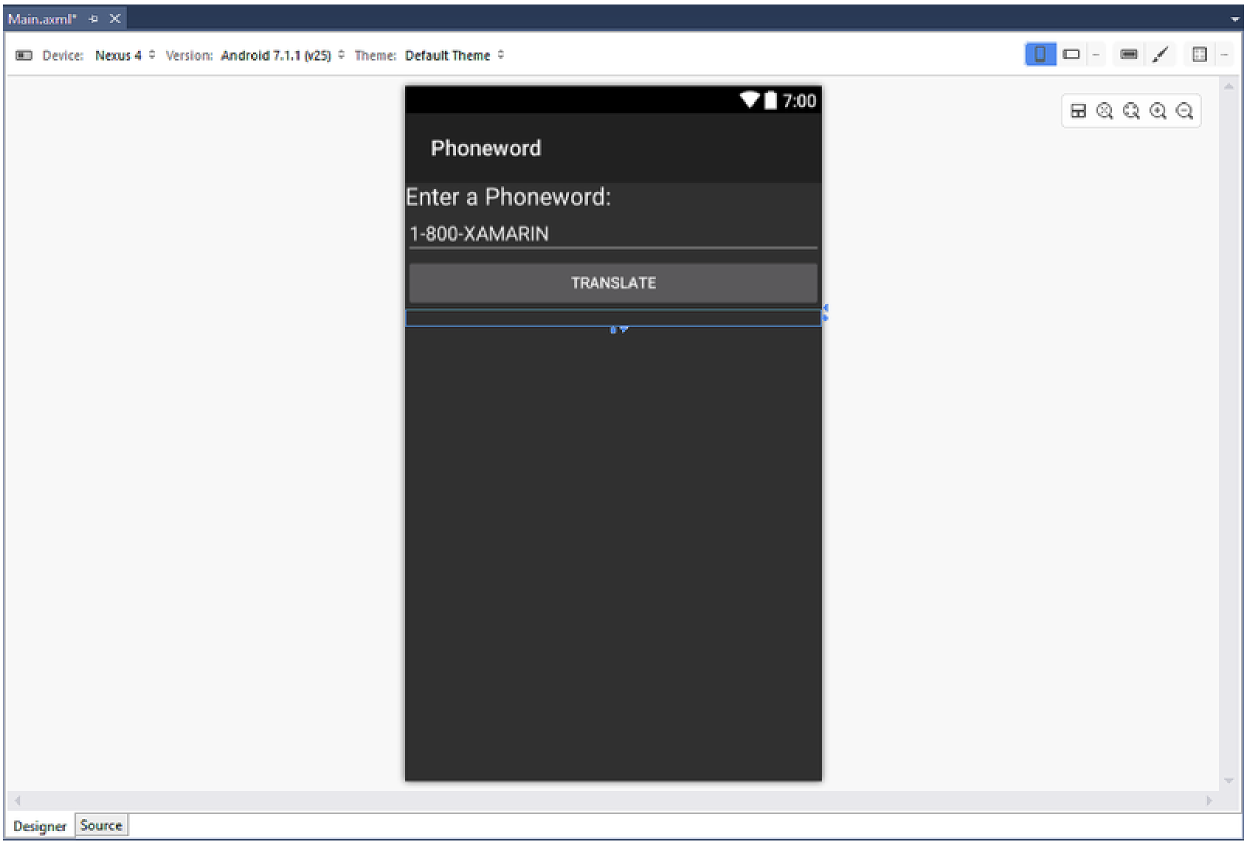


The next few sections explore the relationships between the various parts of the **Phoneword** application; this should provide you with a better understanding of the diagram above. This exploration begins with the user interface as it discusses the Android designer and layout files.

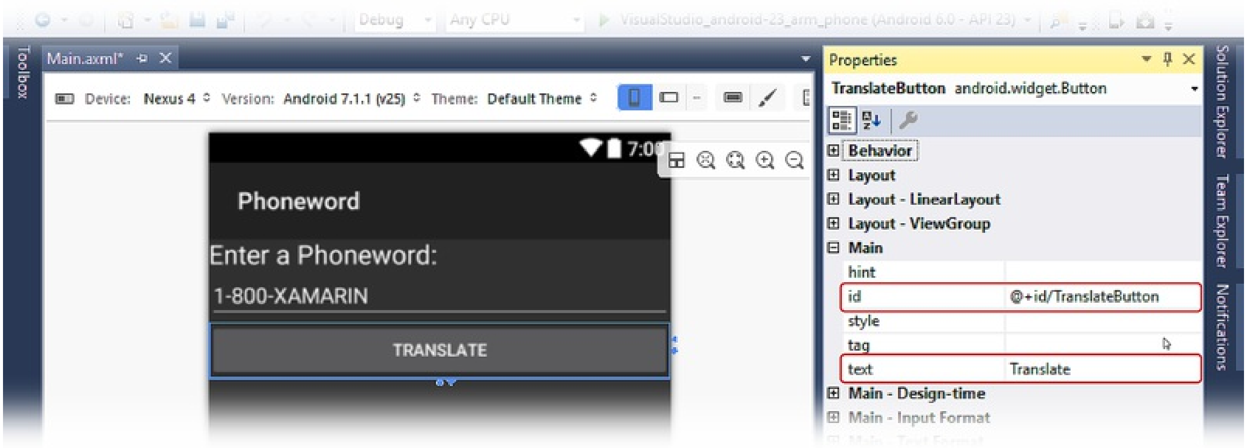
User Interface

> **TIP**
>
> Newer releases of Visual Studio support opening .xml files inside the Android Designer.
>
> Both .axml and .xml files are supported in the Android Designer.

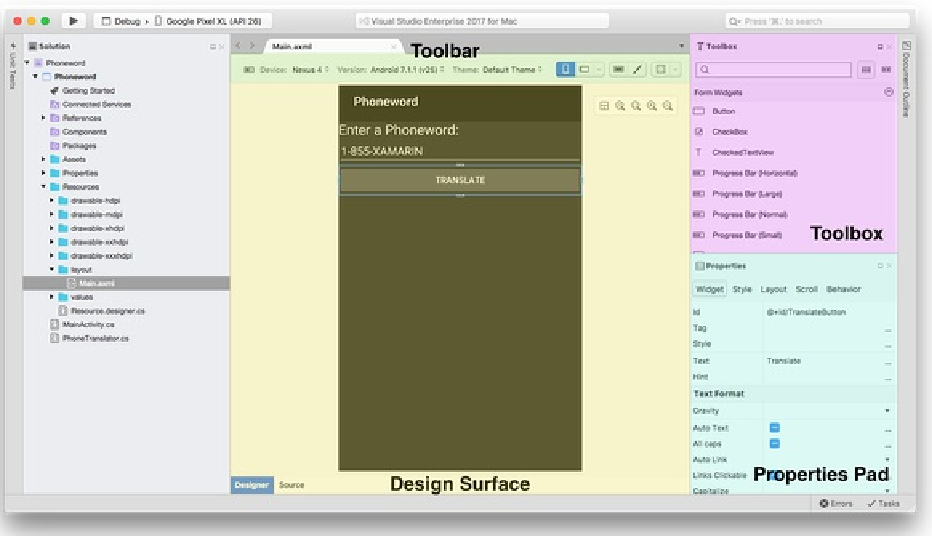
activity_main.axml is the user interface layout file for the first screen in the application. The .axml indicates that this is an Android designer file (AXML stands for *Android XML*). The name *Main* is arbitrary from Android's point of view – the layout file could have been named something else. When you open **activity_main.axml** in the IDE, it brings up the visual editor for Android layout files called the *Android Designer*.



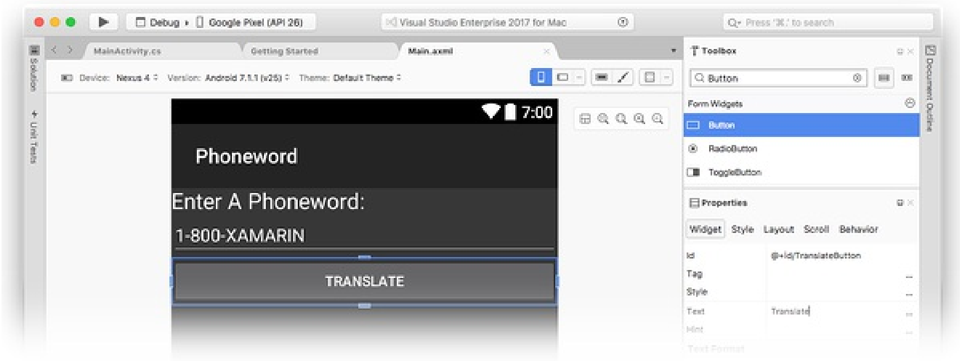
In the **Phoneword** app, the **TranslateButton**'s ID is set to `@+id/TranslateButton`:



Main.axml is the user interface layout file for the first screen in the application. The .axml indicates that this is an Android designer file (AXML stands for *Android XML*). The name *Main* is arbitrary from Android's point of view – the layout file could have been named something else. When you open **Main.axml** in the IDE, it brings up the visual editor for Android layout files called the *Android Designer*.



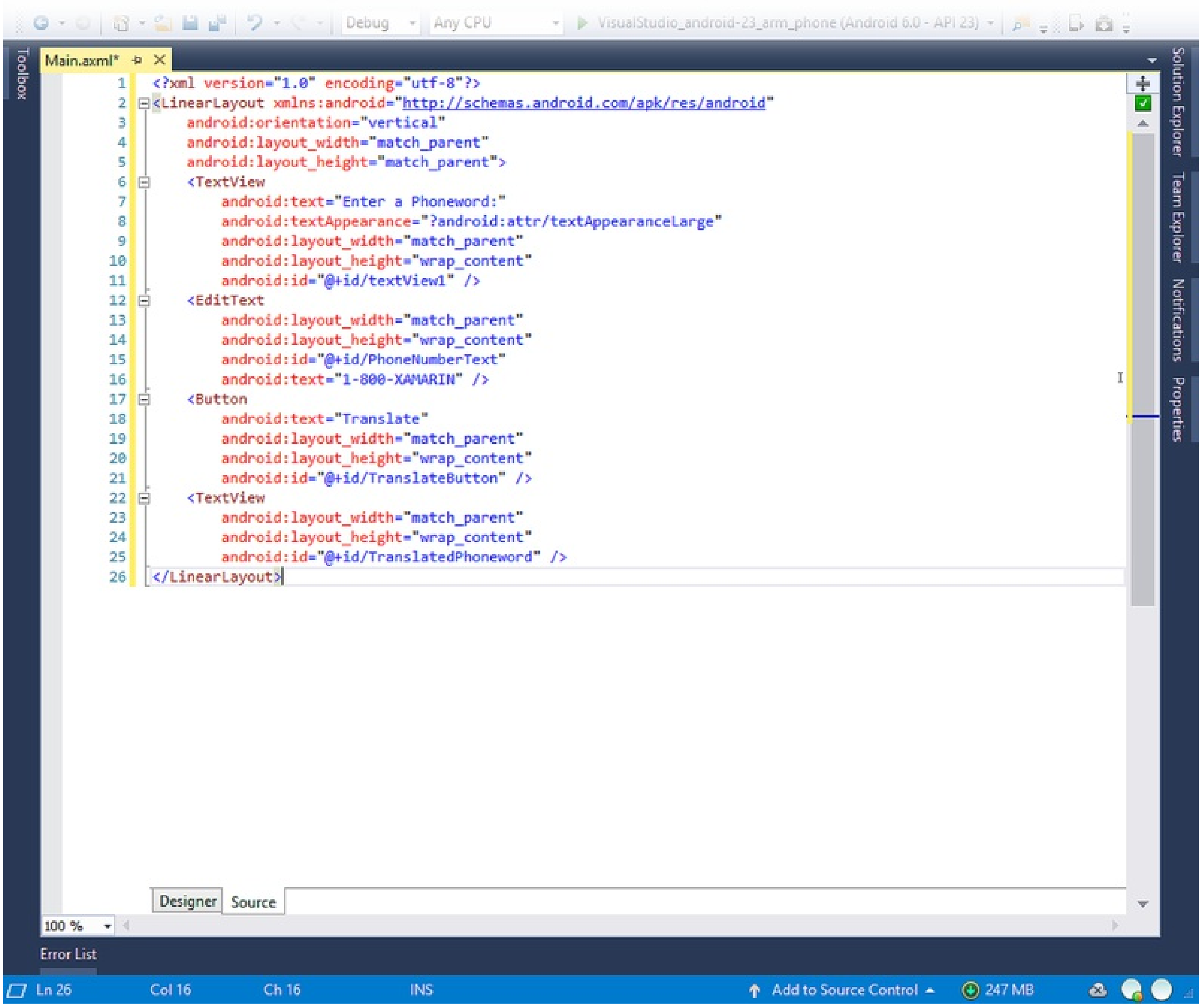
In the **Phoneword** app, the **TranslateButton**'s ID is set to `@+id/TranslateButton`:

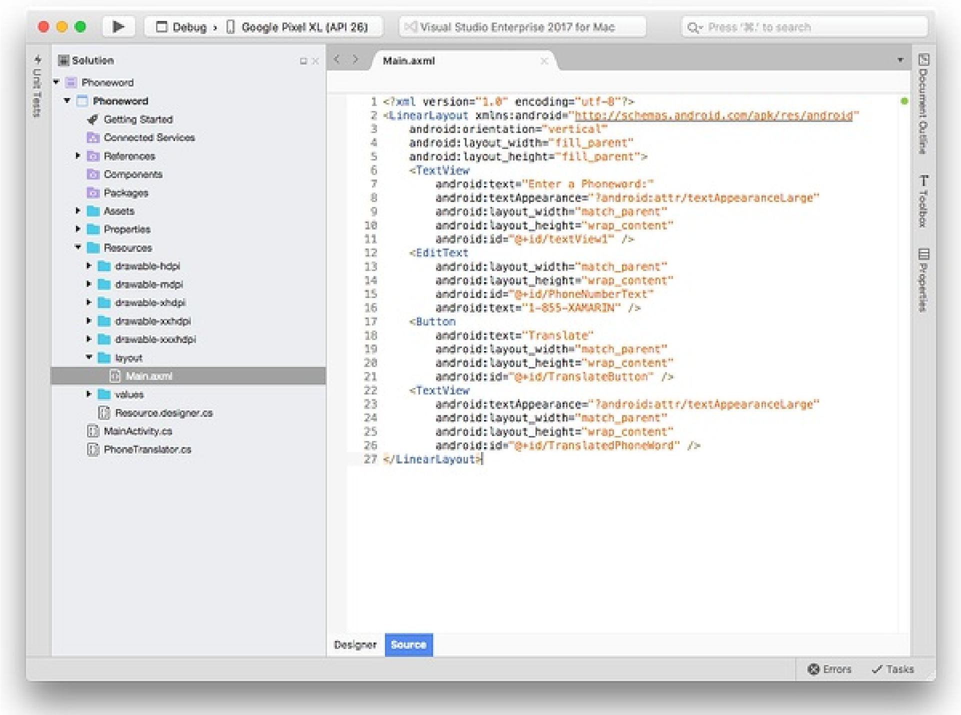


When you set the `id` property of the **TranslateButton**, the Android Designer maps the **TranslateButton** control to the `Resource` class and assigns it a *resource ID* of `TranslateButton`. This mapping of visual control to class makes it possible to locate and use the **TranslateButton** and other controls in app code. This will be covered in more detail when you break apart the code that powers the controls. All you need to know for now is that the code representation of a control is linked to the visual representation of the control in the designer via the `id` property.

Source view

Everything defined on the design surface is translated into XML for Xamarin.Android to use. The Android Designer provides a source view that contains the XML that was generated from the visual designer. You can view this XML by switching to the **Source** panel in the lower left of the designer view, as illustrated by the screenshot below:





This XML source code should contain four control elements: Two **TextView**s, one **EditText** and one **Button** element. For a more in-depth tour of the Android Designer, refer to the Xamarin Android Designer Overview guide.

The tools and concepts behind the visual part of the user interface have now been covered. Next, it's time to jump into the code that powers the user interface as Activities and the Activity Lifecycle are explored.

Activities and the Activity Lifecycle

The `Activity` class contains the code that powers the user interface. The Activity is responsible for responding to user interaction and creating a dynamic user experience. This section introduces the `Activity` class, discusses the Activity Lifecycle, and dissects the code that powers the user interface in the **Phoneword** application.

Activity class

The **Phoneword** application has only one screen (Activity). The class that powers the screen is called `MainActivity` and lives in the **MainActivity.cs** file. The name `MainActivity` has no special significance in Android – although the convention is to name the first Activity in an application `MainActivity`, Android does not care if it is named something else.

When you open **MainActivity.cs**, you can see that the `MainActivity` class is a *subclass* of the `Activity` class, and that the Activity is adorned with the Activity attribute:

```
[Activity (Label = "Phone Word", MainLauncher = true)]
public class MainActivity : Activity
{
  ...
}
```

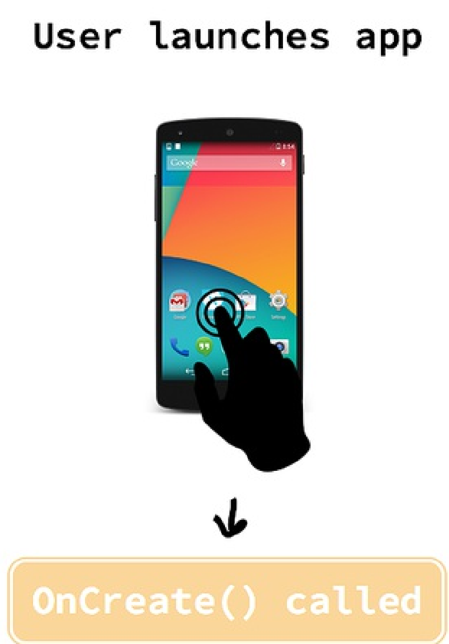
The `Activity` Attribute registers the Activity with the Android Manifest; this lets Android know that this class is part of the **Phoneword** application managed by this manifest. The `Label` property sets the text that will be displayed at the top of the screen.

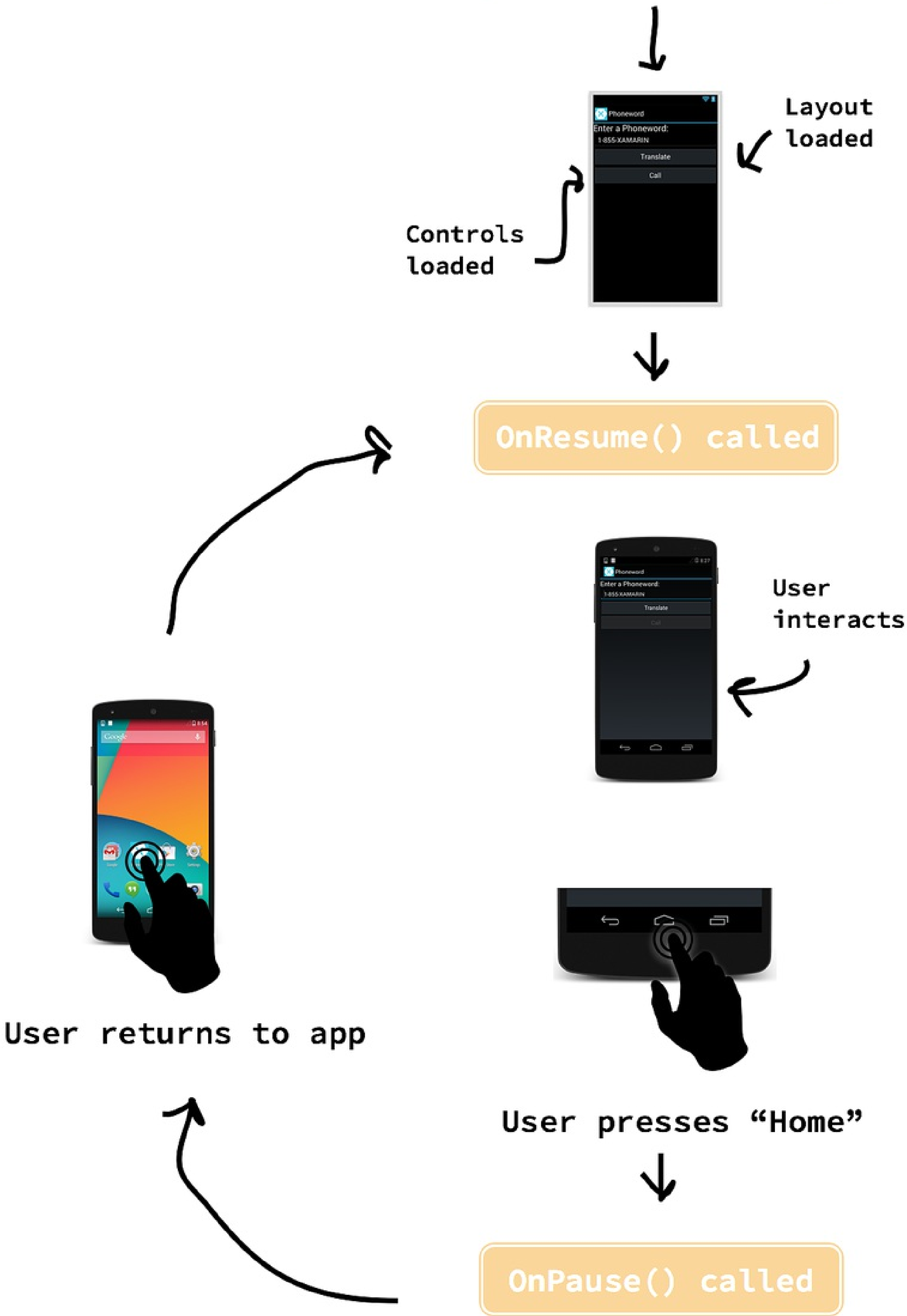
The `MainLauncher` property tells Android to display this Activity when the application starts up. This property becomes important as you add more Activities (screens) to the application as explained in the Hello, Android Multiscreen guide.

Now that the basics of `MainActivity` have been covered, it's time to dive deeper into the Activity code by introducing the *Activity Lifecycle*.

Activity lifecycle

In Android, Activities go through different stages of a lifecycle depending on their interactions with the user. Activities can be created, started and paused, resumed and destroyed, and so on. The `Activity` class contains methods that the system calls at certain points in the screen's lifecycle. The following diagram illustrates a typical life of an Activity as well as some of the corresponding lifecycle methods:





By overriding `Activity` lifecycle methods, you can control how the Activity loads, how it reacts to the user, and even what happens after it disappears from the device screen. For example, you can override the lifecycle methods in the diagram above to perform some important tasks:

- **OnCreate** – Creates views, initializes variables, and performs other prep work that must be done before the user sees the Activity. This method is called only once when the Activity is loaded into memory.
- **OnResume** – Performs any tasks that must happen every time the Activity returns to the device screen.
- **OnPause** – Performs any tasks that must happen every time the Activity leaves the device screen.

When you add custom code to a lifecycle method in the `Activity`, you *override* that lifecycle method's *base implementation*. You tap into the existing lifecycle method (which has some code already attached to it), and you extend that method with your own code. You call the base implementation from inside your method to ensure that the original code runs before your new code. An example of this is illustrated in the next section.

The Activity Lifecycle is an important and complex part of Android. If you'd like to learn more about Activities after you finish the *Getting Started* series, read the Activity Lifecycle guide. In this guide, the next focus is the first stage of the Activity Lifecycle, `OnCreate`.

OnCreate

Android calls the `Activity`'s `OnCreate` method when it creates the Activity (before the screen is presented to the user). You can override the `OnCreate` lifecycle method to create views and prepare your Activity to meet the user:

```
protected override void OnCreate (Bundle bundle)
{
    base.OnCreate (bundle);

    // Set our view from the "main" layout resource
    SetContentView (Resource.Layout.Main);
    // Additional setup code will go here
}
```

In the **Phoneword** app, the first thing to do in `OnCreate` is load the user interface created in the Android Designer. To load the UI, call `SetContentView` and pass it the *resource layout name* for the layout file: **activity_main.axml**. The layout is located at `Resource.Layout.activity_main`:

```
SetContentView (Resource.Layout.activity_main);
```

When `MainActivity` starts up, it creates a view that is based on the contents of the **activity_main.axml** file.

In the **Phoneword** app, the first thing to do in `OnCreate` is load the user interface created in the Android Designer. To load the UI, call `SetContentView` and pass it the *resource layout name* for the layout file: **Main.axml**. The layout is located at `Resource.Layout.Main`:

```
SetContentView (Resource.Layout.Main);
```

When `MainActivity` starts up, it creates a view that is based on the contents of the **Main.axml** file. Note that the layout file name is matched to the Activity name – *Main*.axml is the layout for *Main*Activity. This isn't required from Android's point of view, but as you begin to add more screens to the application, you'll find that this naming convention makes it easier to match the code file to the layout file.

After the layout file is prepared, you can start looking up controls. To look up a control, call `FindViewById` and pass in the resource ID of the control:

```
EditText phoneNumberText = FindViewById<EditText>(Resource.Id.PhoneNumberText);
Button translateButton = FindViewById<Button>(Resource.Id.TranslateButton);
TextView translatedPhoneWord = FindViewById<TextView>(Resource.Id.TranslatedPhoneWord);
```

Now that you have references to the controls in the layout file, you can start programming them to respond to user interaction.

Responding to user interaction

In Android, the `Click` event listens for the user's touch. In this app, the `Click` event is handled with a lambda, but a delegate or a named event handler could be used instead. The final **TranslateButton** code resembled the following:

```
translateButton.Click += (sender, e) =>
{
    // Translate user's alphanumeric phone number to numeric
    translatedNumber = PhonewordTranslator.ToNumber(phoneNumberText.Text);
    if (string.IsNullOrWhiteSpace(translatedNumber))
    {
        translatedPhoneWord.Text = string.Empty;
    }
    else
    {
        translatedPhoneWord.Text = translatedNumber;
    }
};
```

Testing, deployment, and finishing touches

Both Visual Studio for Mac and Visual Studio provide many options for testing and deploying an application. This section covers debugging options, demonstrates testing applications on a device, and introduces tools for creating custom app icons for different screen densities.

Debugging tools

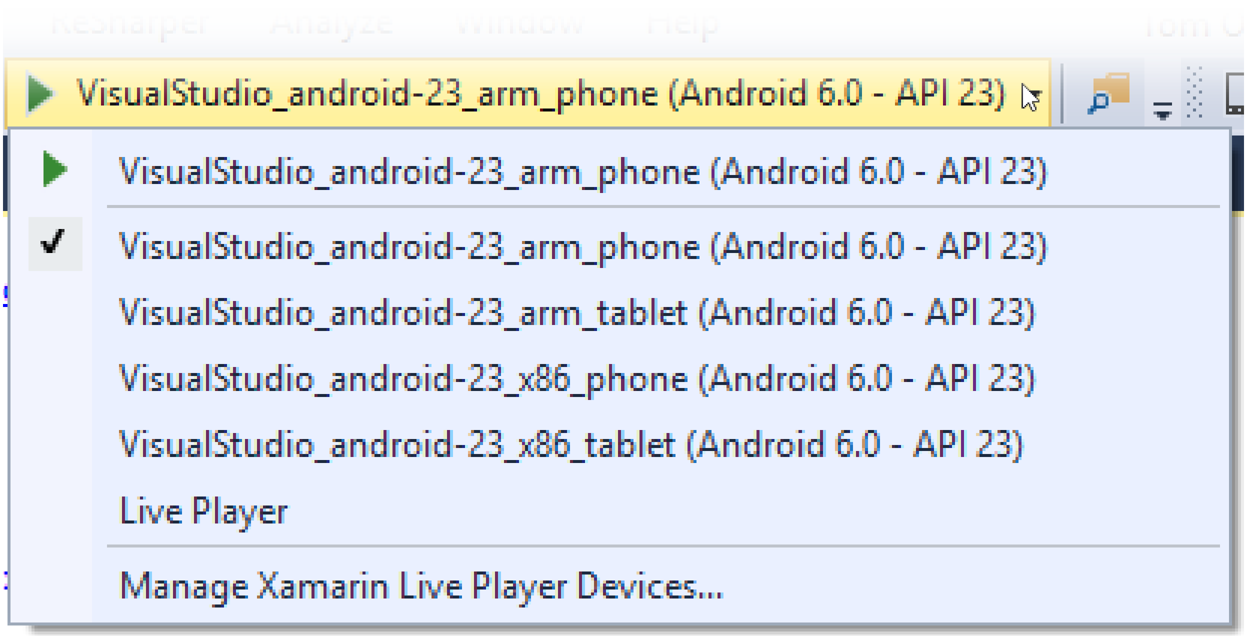
Issues in application code can be difficult to diagnose. To help diagnose complex code issues, you can Set a Breakpoint, Step Through Code, or Output Information to the Log Window.

Deploy to a device

The emulator is a good start for deploying and testing an application, but users will not consume the final app in an emulator. It's a good practice to test applications on a real device early and often.

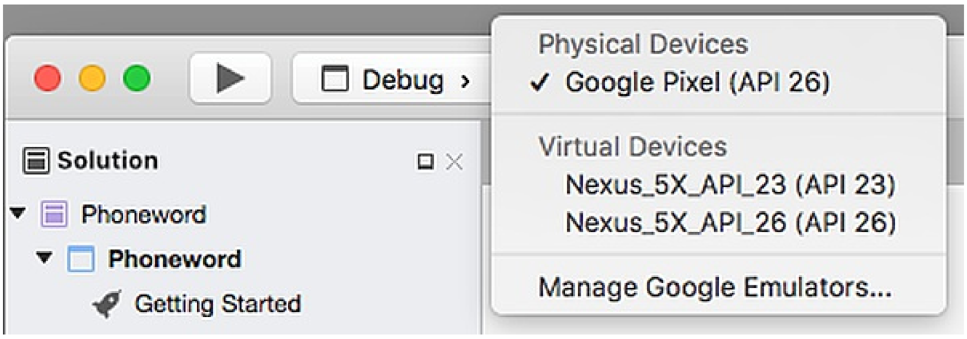
Before an Android device can be used for testing applications, it needs to be configured for development. The Set Up Device for Development guide provides thorough instructions on getting a device ready for development.

After the device is configured, you can deploy to it by plugging it in, selecting it from the **Select Device** dialog, and starting the application:



After the device is configured, you can deploy to it by plugging it in, pressing **Start (Play)**, selecting it from the

Select Device dialog, and pressing **OK**:

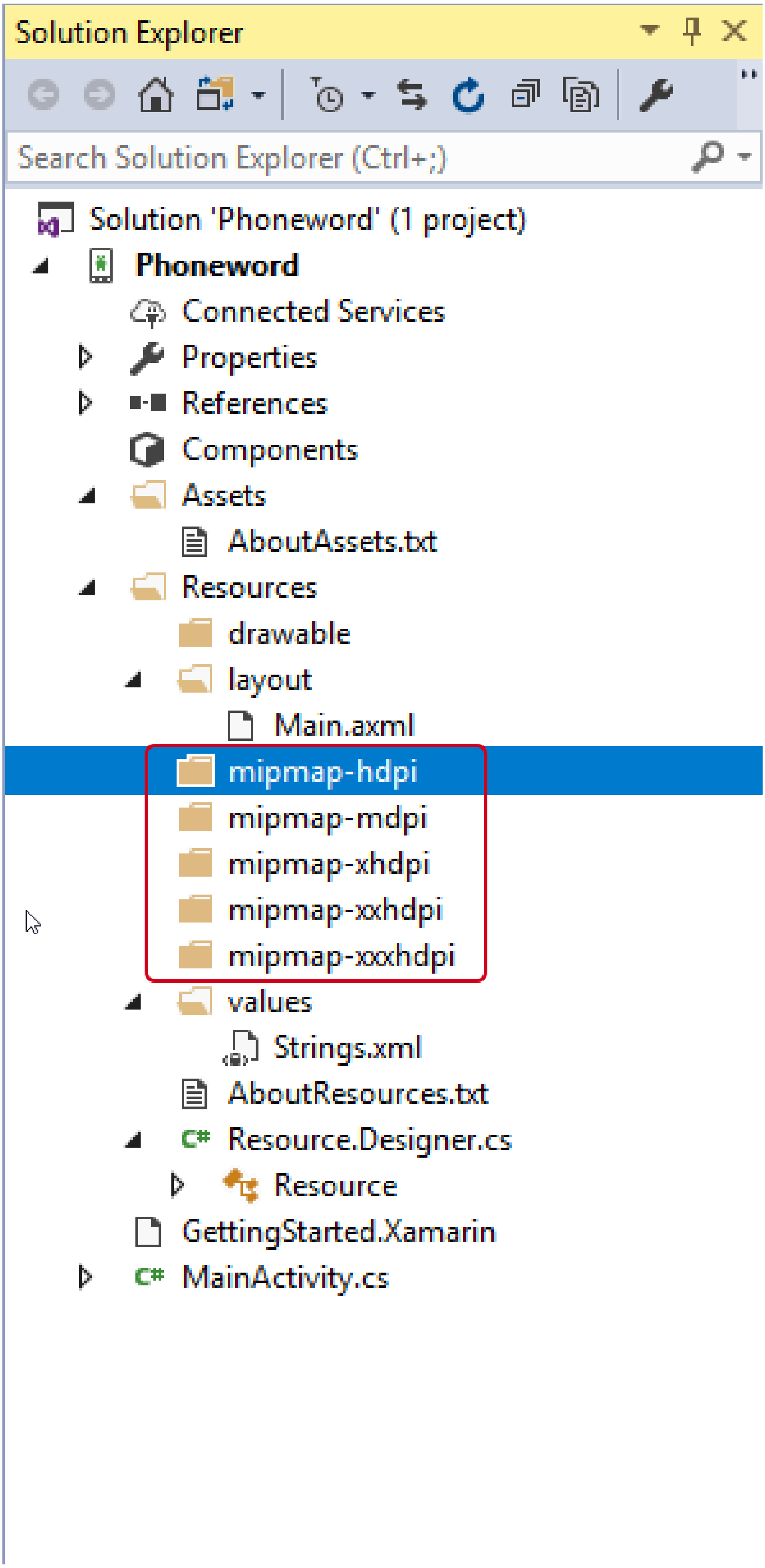


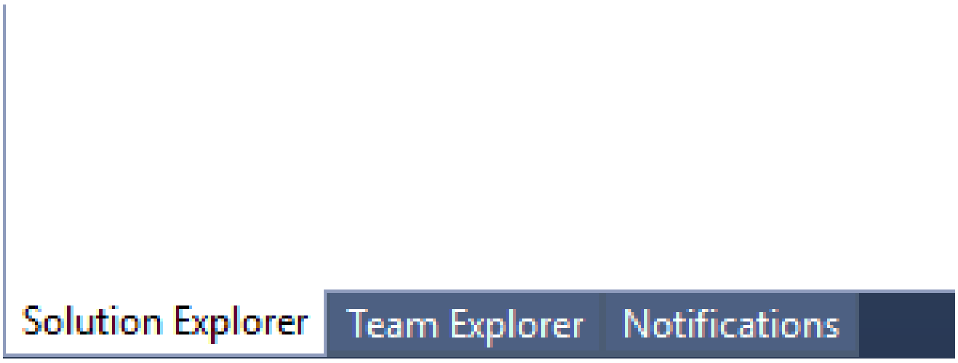
This launches the application on the device:

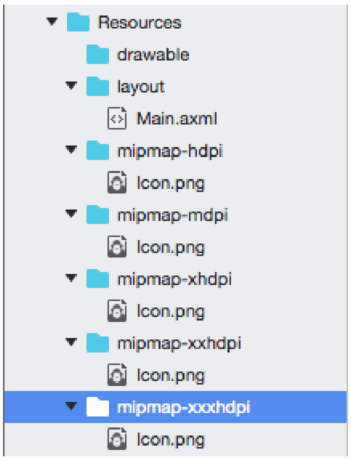
Set icons for different screen densities

Android devices come in different screen sizes and resolutions, and not all images look good on all screens. For example, here is a screenshot of a low-density icon on a high-density Nexus 5. Notice how blurry it is compared to the surrounding icons:

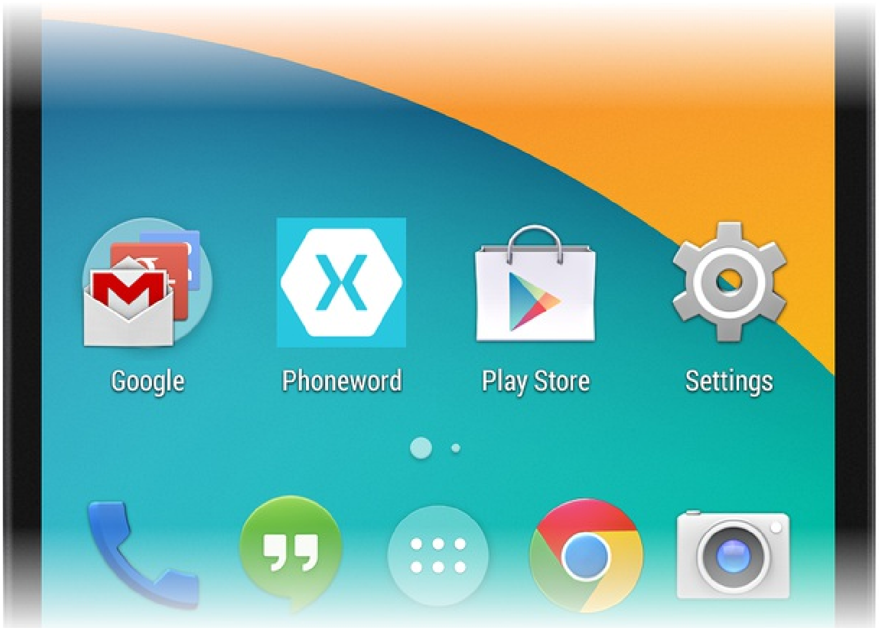
To account for this, it is good practice to add icons of different resolutions to the **Resources** folder. Android provides different versions of the **mipmap** folder to handle launcher icons of different densities, *mdpi* for medium, *hdpi* for high, and *xhdpi*, *xxhdpi*, *xxxhdpi* for very high density screens. Icons of varying sizes are stored in the appropriate **mipmap-** folders:







Android will pick the icon with the appropriate density:



Generate custom icons

Not everyone has a designer available to create the custom icons and launch images that an app needs to stand out. Here are several alternate approaches to generating custom app artwork:

- Android Asset Studio – A web-based, in-browser generator for all types of Android icons, with links to other useful community tools. It works best in Google Chrome.
- Visual Studio – You can use this to create a simple icon set for your app directly in the IDE.
- Fiverr – Choose from a variety of designers to create an icon set for you, starting at $5. Can be hit or miss but a good resource if you need icons designed on the fly.
- Android Asset Studio – A web-based, in-browser generator for all types of Android icons, with links to

other useful community tools. It works best in Google Chrome.

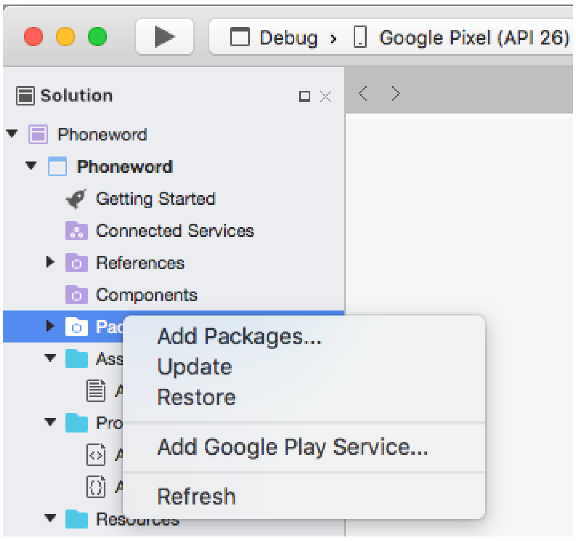
- Pixelmator – A versatile image editing app for Mac that costs about $30.
- Fiverr – Choose from a variety of designers to create an icon set for you, starting at $5. Can be hit or miss but a good resource if you need icons designed on the fly.

For more information about icon sizes and requirements, refer to the Android Resources guide.

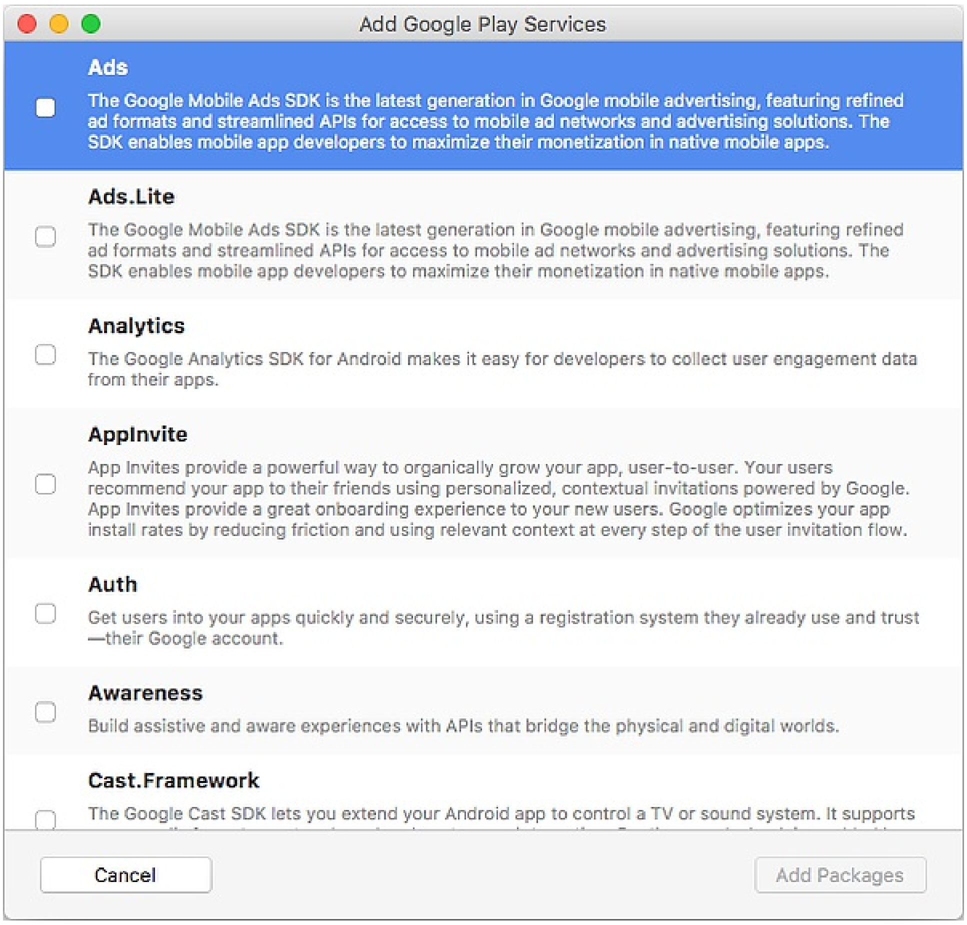
Adding Google Play Services packages

Google Play Services is a set of add-on libraries that allows Android developers to take advantage of the most recent features from Google such as Google Maps, Google Cloud Messaging, and in-app billing. Previously, bindings to all Google Play Services libraries were provided by Xamarin in the form of a single package – beginning with Visual Studio for Mac, a new project dialog is available for selecting which Google Play Services packages to include in your app.

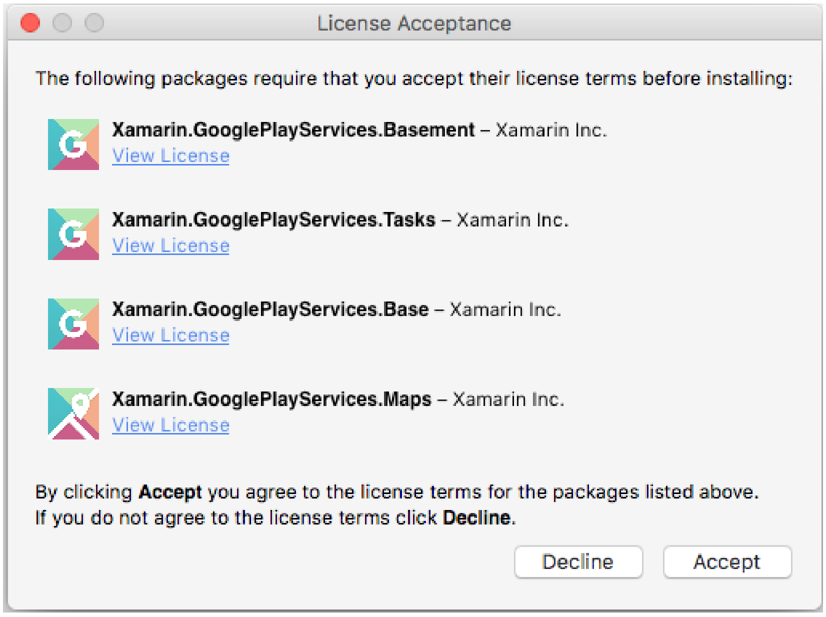
To add one or more Google Play Service libraries, right-click the **Packages** node in your project tree and click **Add Google Play Service...**:



When the **Add Google Play Services** dialog is presented, select the packages (nugets) that you want to add to your project:



When you select a service and click **Add Package**, Visual Studio for Mac downloads and installs the package you select as well as any dependent Google Play Services packages that it requires. In some cases, you may see a **License Acceptance** dialog that requires you to click **Accept** before the packages are installed:



Summary

Congratulations! You should now have a solid understanding of the components of a Xamarin.Android application as well as the tools required to create it.

In the next tutorial of the *Getting Started* series, you will extend your application to handle multiple screens as you explore more advanced Android architecture and concepts.

Hello, Android Multiscreen

11/2/2020 • 2 minutes to read • Edit Online

In this two-part guide, you expand the Phoneword application that you created in the Hello, Android guide to handle a second screen. Along the way, this guide will introduce the basic Android Application Building Blocks and dive deeper into Android architecture as you develop a better understanding of Android application structure and functionality.

Part 1: Quickstart

In the first part of this guide, you'll add a second screen to the Phoneword application to keep track of the history of numbers called from the app. The final app will display a second screen that lists the call history.

Part 2: Deep Dive

In the second part of this document, you will review what you've built and discusses architecture, navigation, and other new Android concepts that are encountered along the way.

Related Links

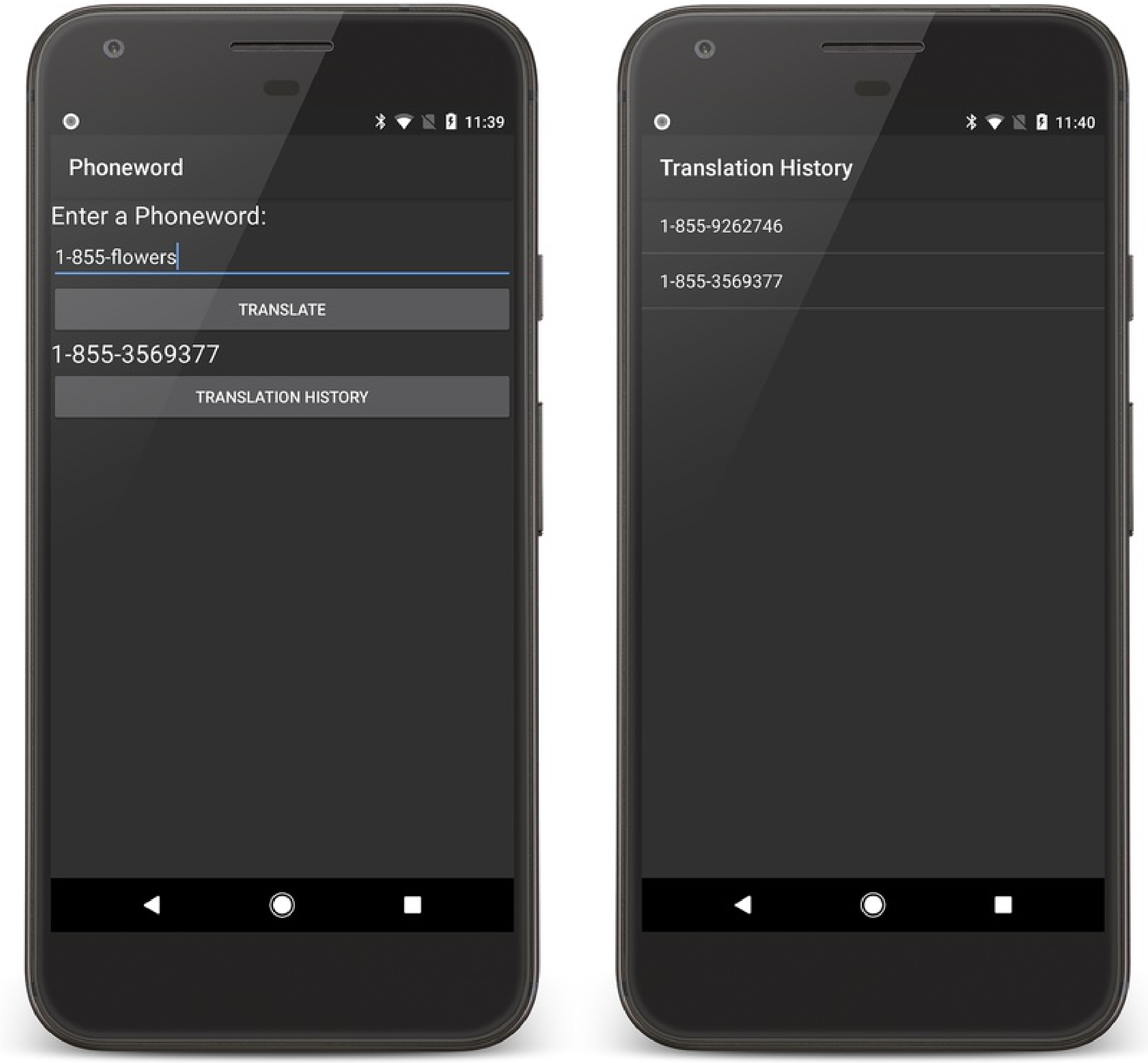
- Android Getting Started
- Debugging in Visual Studio
- Visual Studio for Mac Recipes - Debugging

Hello, Android Multiscreen: Quickstart

7/8/2021 • 6 minutes to read • Edit Online

This two-part guide expands the Phoneword application to handle a second screen. Along the way, basic Android Application Building Blocks are introduced with a deeper dive into Android architecture.

In the walkthrough portion of this guide, you'll add a second screen to the Phoneword application to keep track of the history of numbers translated using the app. The final application will have a second screen that displays the numbers that were "translated", as illustrated by the screenshot on the right:



The accompanying Deep Dive reviews what was built and discusses architecture, navigation, and other new Android concepts encountered along the way.

Requirements

Because this guide picks up where Hello, Android leaves off, it requires completion of the Hello, Android Quickstart. If you would like to jump directly into the walkthrough below, you can download the completed version of Phoneword (from the Hello, Android Quickstart) and use it to start the walkthrough.

Walkthrough

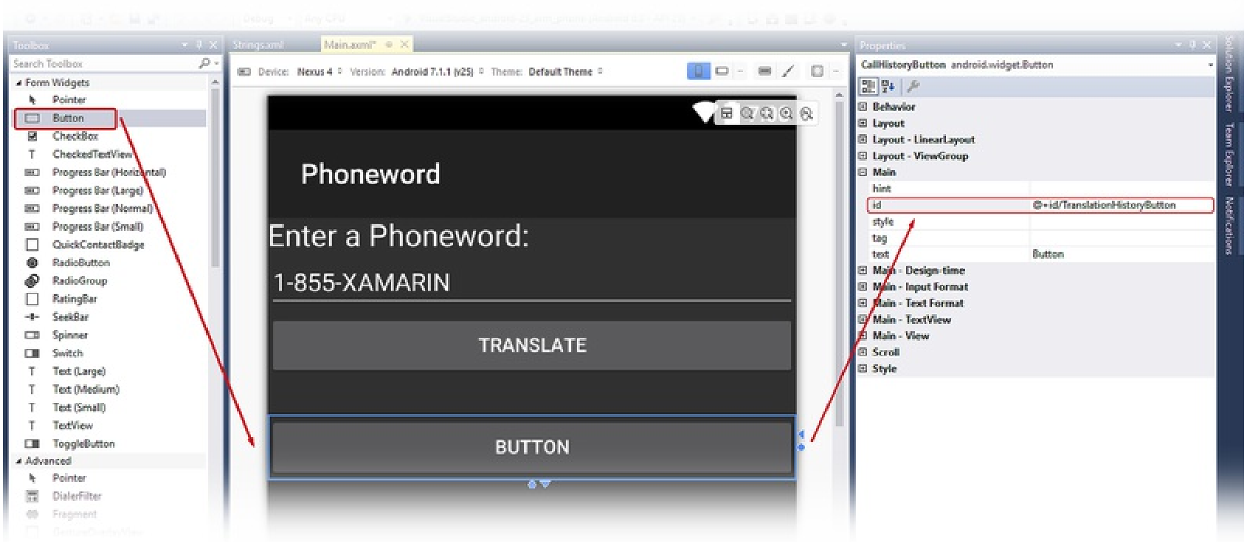
In this walkthrough you'll add a **Translation History** screen to the **Phoneword** application.

Start by opening the **Phoneword** application in Visual Studio and editing the **Main.axml** file from the **Solution Explorer**.

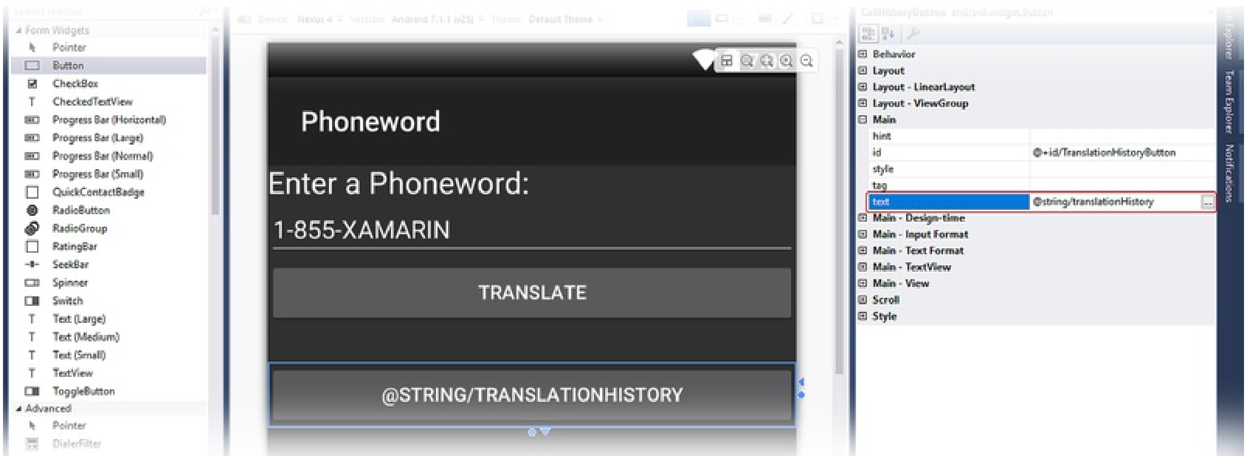
> **TIP**
> Newer releases of Visual Studio support opening .xml files inside the Android Designer.
>
> Both .axml and .xml files are supported in the Android Designer.

Updating the layout

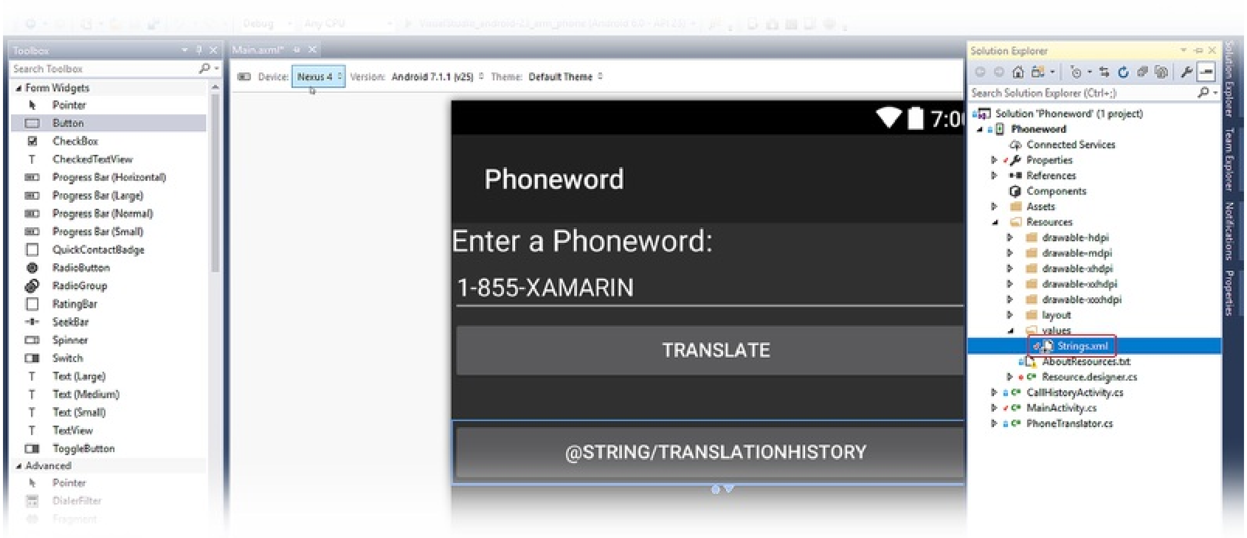
From the **Toolbox**, drag a **Button** onto the design surface and place it below the **TranslatedPhoneWord** TextView. In the **Properties** pane, change the button **Id** to `@+id/TranslationHistoryButton`



Set the **Text** property of the button to `@string/translationHistory`. The Android Designer will interpret this literally, but you're going to make a few changes so that the button's text shows up correctly:



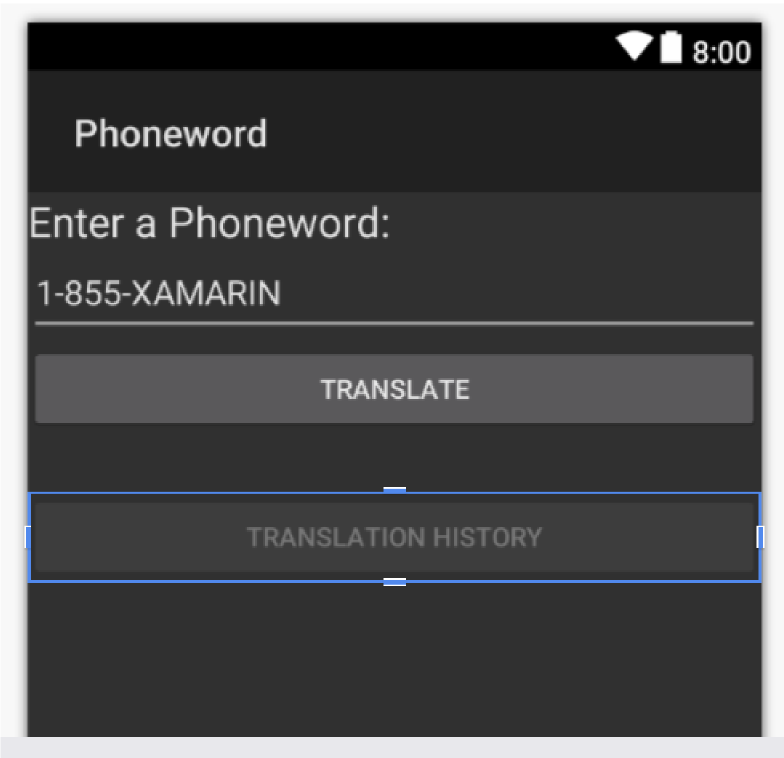
Expand the **values** node under the **Resources** folder in the **Solution Explorer** and double-click the string resources file, **Strings.xml**:



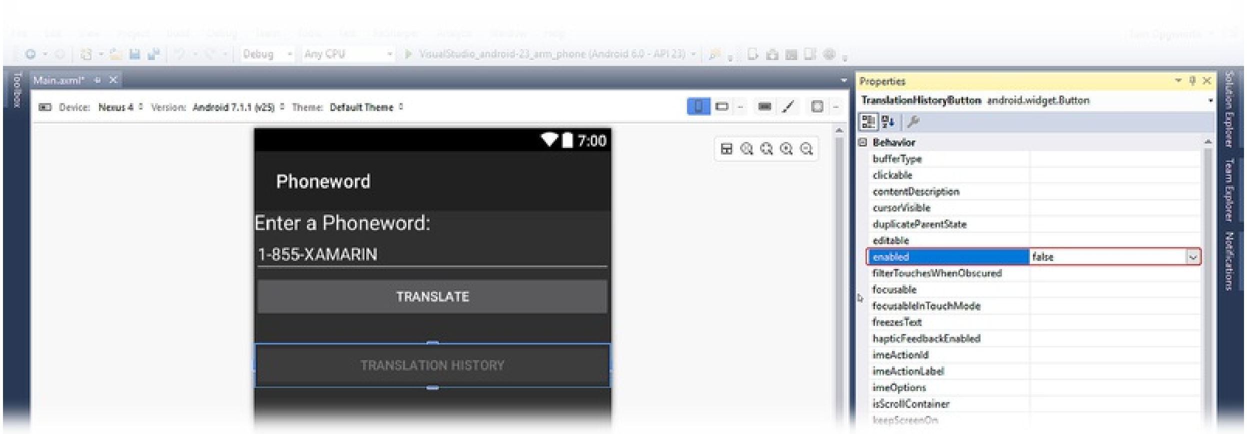
Add the `translationHistory` string name and value to the **Strings.xml** file and save it:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="translationHistory">Translation History</string>
    <string name="ApplicationName">Phoneword</string>
</resources>
```

The **Translation History** button text should update to reflect the new string value:

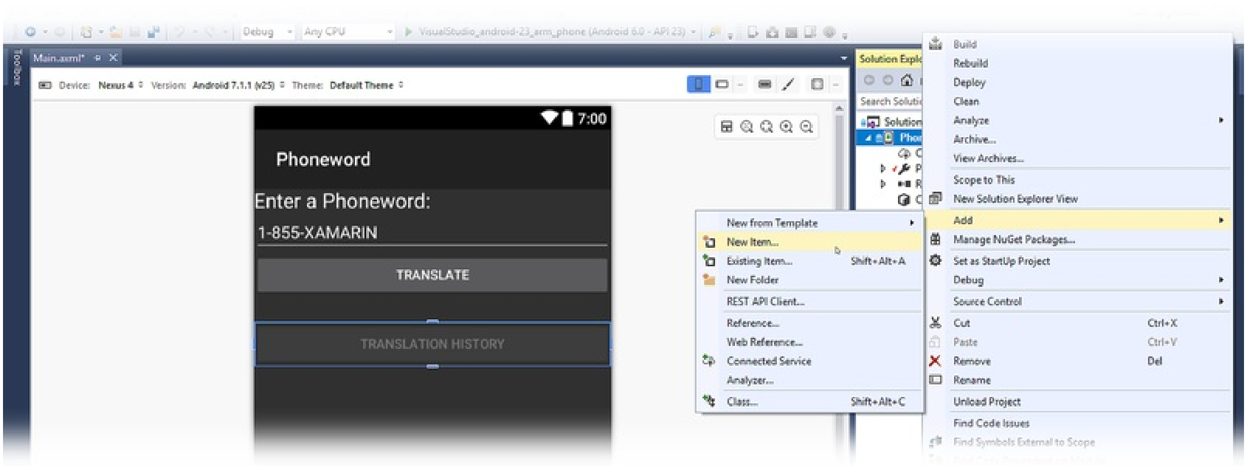


With the **Translation History** button selected on the design surface, find the `enabled` setting in the **Properties** pane and set its value to `false` to disable the button. This will cause the button to become darker on the design surface:



Creating the second activity

Create a second Activity to power the second screen. In the **Solution Explorer**, right-click the **Phoneword** project and choose **Add > New Item...**:



In the **Add New Item** dialog, choose **Visual C# > Activity** and name the Activity file **TranslationHistoryActivity.cs**.

Replace the template code in **TranslationHistoryActivity.cs** with the following:

```
using System;
using System.Collections.Generic;
using Android.App;
using Android.OS;
using Android.Widget;
namespace Phoneword
{
    [Activity(Label = "@string/translationHistory")]
    public class TranslationHistoryActivity : ListActivity
    {
        protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);
            // Create your application here
            var phoneNumbers = Intent.Extras.GetStringArrayList("phone_numbers") ?? new string[0];
            this.ListAdapter = new ArrayAdapter<string>(this, Android.Resource.Layout.SimpleListItem1,
phoneNumbers);
        }
    }
}
```

In this class, you're creating a `ListActivity` and populating it programmatically, so you don't need to create a new layout file for this Activity. This is discussed in more detail in the Hello, Android Multiscreen Deep Dive.

Adding a list

This app collects phone numbers (that the user has translated on the first screen) and passes them to the second screen. The phone numbers are stored as a list of strings. To support lists (and Intents, which are used later), add the following `using` directives to the top of **MainActivity.cs**:

```
using System.Collections.Generic;
using Android.Content;
```

Next, create an empty list that can be filled with phone numbers. The `MainActivity` class will look like this:

```
[Activity(Label = "Phoneword", MainLauncher = true)]
public class MainActivity : Activity
{
    static readonly List<string> phoneNumbers = new List<string>();
    ...// OnCreate, etc.
}
```

In the `MainActivity` class, add the following code to register the **Translation History** button (place this line after the `translateButton` declaration):

```
Button translationHistoryButton = FindViewById<Button> (Resource.Id.TranslationHistoryButton);
```

Add the following code to the end of the `OnCreate` method to wire up the **Translation History** button:

```
translationHistoryButton.Click += (sender, e) =>
{
    var intent = new Intent(this, typeof(TranslationHistoryActivity));
    intent.PutStringArrayListExtra("phone_numbers", phoneNumbers);
    StartActivity(intent);
};
```

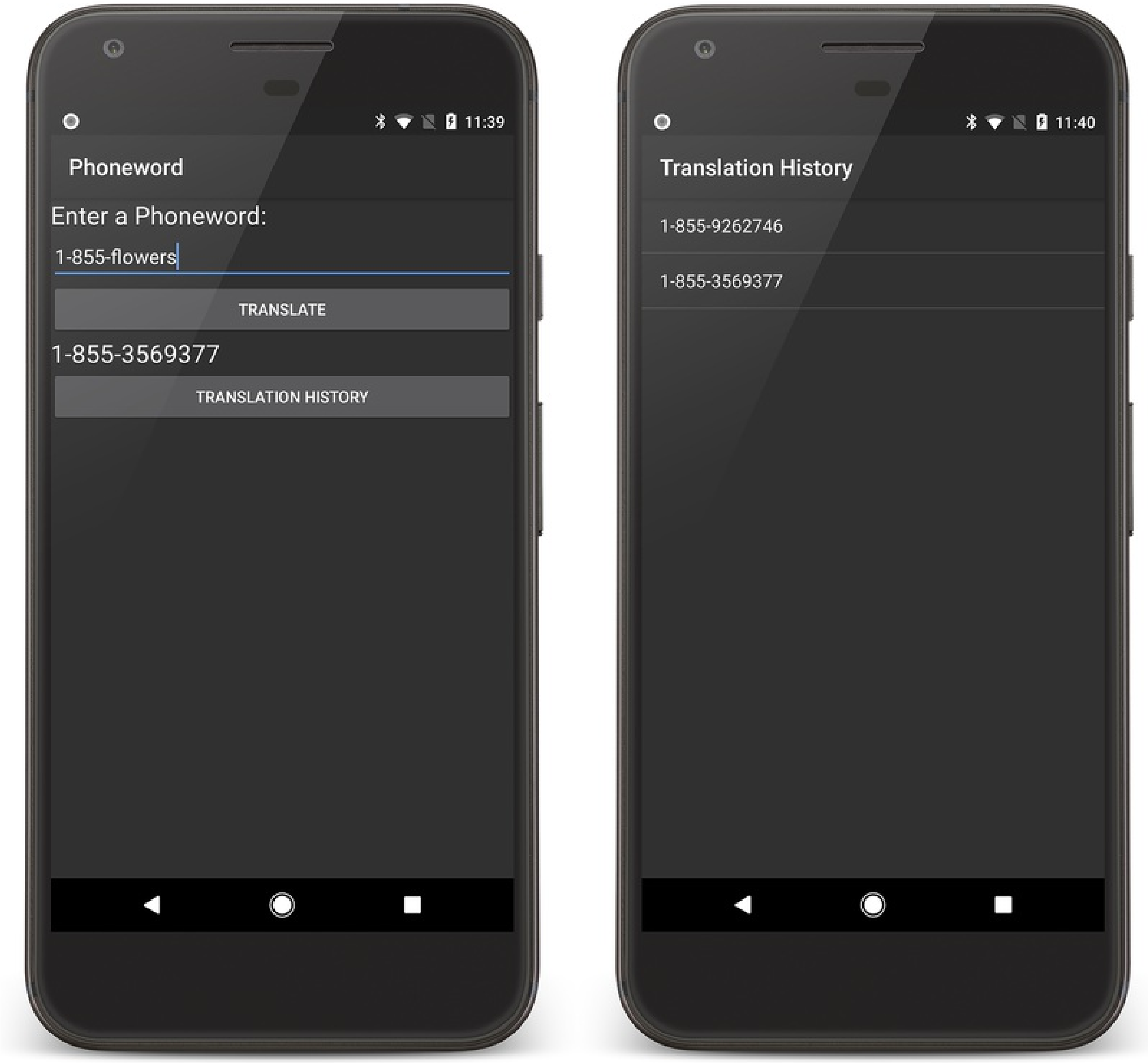
Update the **Translate** button to add the phone number to the list of `phoneNumbers`. The `Click` handler for the `translateButton` should resemble the following code:

```
// Add code to translate number
string translatedNumber = string.Empty;
translateButton.Click += (sender, e) =>
{
    // Translate user's alphanumeric phone number to numeric
    translatedNumber = Core.PhonewordTranslator.ToNumber(phoneNumberText.Text);
    if (string.IsNullOrWhiteSpace(translatedNumber))
    {
        translatedPhoneWord.Text = "";
    }
    else
    {
        translatedPhoneWord.Text = translatedNumber;
        phoneNumbers.Add(translatedNumber);
        translationHistoryButton.Enabled = true;
    }
};
```

Save and build the application to make sure there are no errors.

Running the app

Deploy the application to an emulator or device. The following screenshots illustrate the running **Phoneword** application:

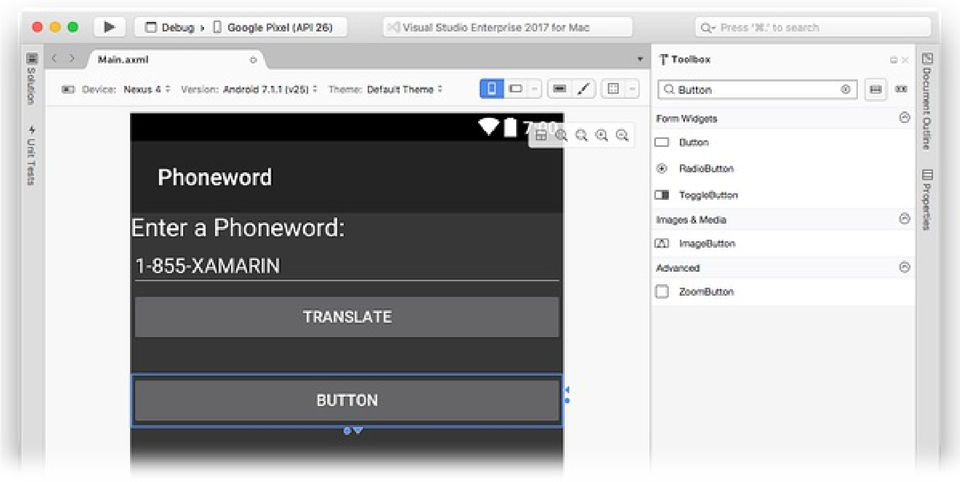


Start by opening the **Phoneword** project in Visual Studio for Mac and editing the **Main.axml** file from the **Solution Pad**.

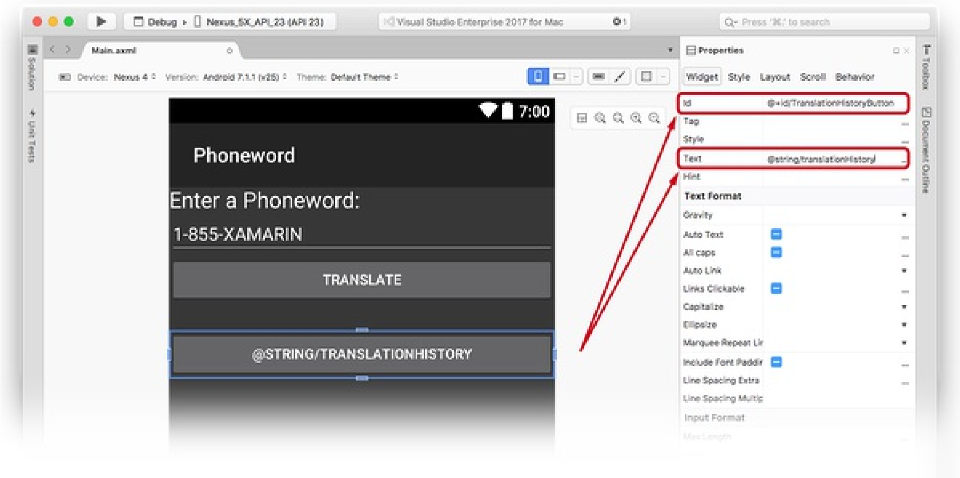
> **TIP**
>
> Newer releases of Visual Studio support opening .xml files inside the Android Designer.
>
> Both .axml and .xml files are supported in the Android Designer.

Updating the layout

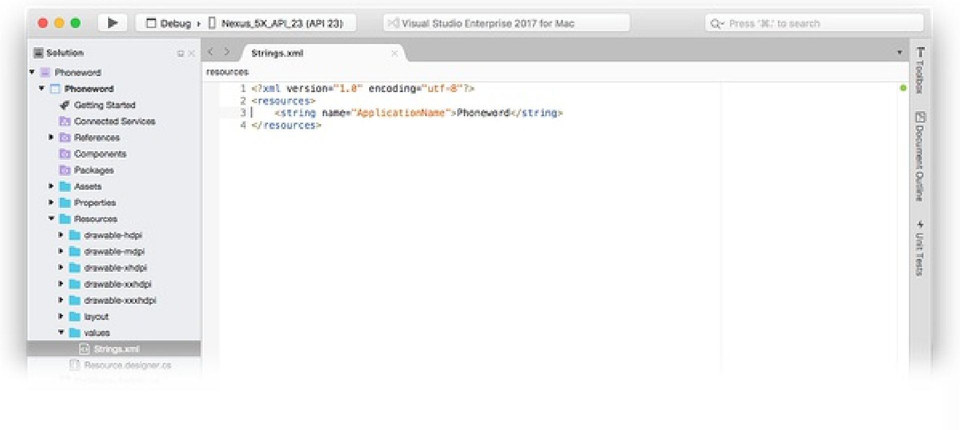
From the **Toolbox**, drag a **Button** onto the design surface and place it below the **TranslatedPhoneWord** TextView. In the **Properties** pad, change the button **Id** to `@+id/TranslationHistoryButton`:



Set the **Text** property of the button to `@string/translationHistory`. The Android Designer will interpret this literally, but you're going to make a few changes so that the button's text shows up correctly:



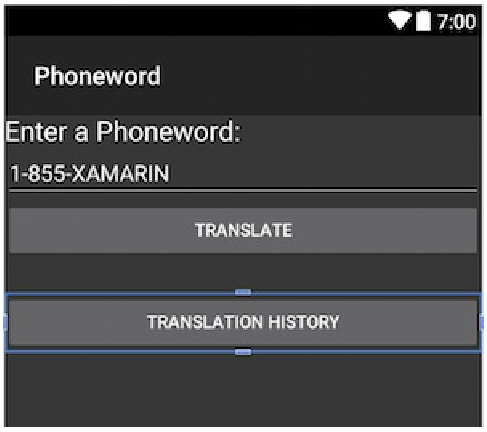
Expand the **values** node under the **Resources** folder in the **Solution Pad** and double-click the string resources file, **Strings.xml**:



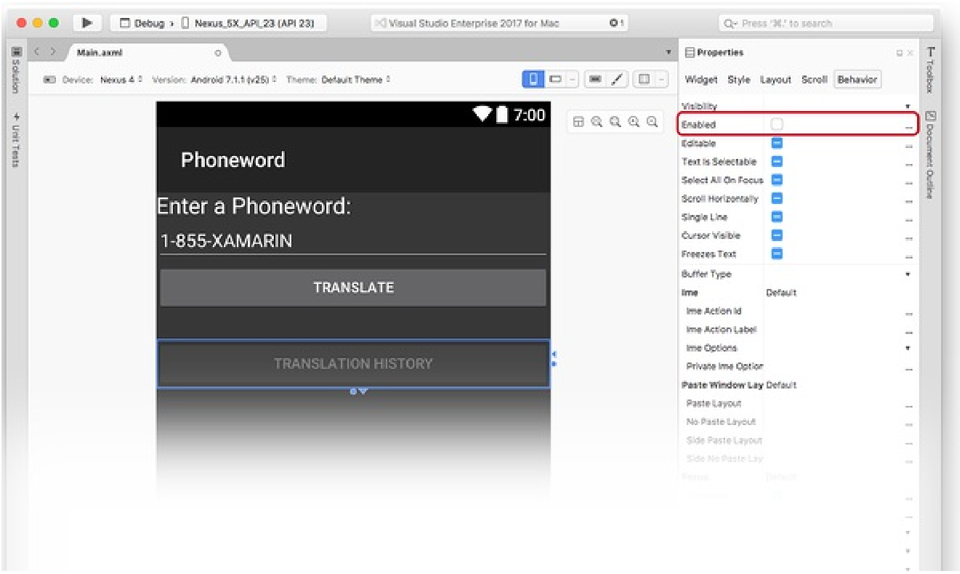
Add the `translationHistory` string name and value to the **Strings.xml** file and save it:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="translationHistory">Translation History</string>
    <string name="ApplicationName">Phoneword</string>
</resources>
```

The **Translation History** button text should update to reflect the new string value:



With the **Translation History** button selected on the design surface, open the **Behavior** tab in the **Properties Pad** and double-click the **Enabled** checkbox to disable the button. This will cause the button to become darker on the design surface:



Creating the second activity

Create a second Activity to power the second screen. In the **Solution Pad**, click the gray gear icon next to the **Phoneword** project and choose **Add > New File...**:

From the **New File** dialog, choose **Android > Activity**, name the Activity `TranslationHistoryActivity`, then click **Add**.

Replace the template code in `TranslationHistoryActivity` with the following:

```
using System;
using System.Collections.Generic;
using Android.App;
using Android.OS;
using Android.Widget;
namespace Phoneword
{
    [Activity(Label = "@string/translationHistory")]
    public class TranslationHistoryActivity : ListActivity
    {
        protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);
            // Create your application here
            var phoneNumbers = Intent.Extras.GetStringArrayList("phone_numbers") ?? new string[0];
            this.ListAdapter = new ArrayAdapter<string>(this, Android.Resource.Layout.SimpleListItem1,
phoneNumbers);
        }
    }
}
```

In this class, a `ListActivity` is created and populated programmatically, so you don't have to create a new layout file for this Activity. This is explained in more detail in the Hello, Android Multiscreen Deep Dive.

Adding a list

This app collects phone numbers (that the user has translated on the first screen) and passes them to the second screen. The phone numbers are stored as a list of strings. To support lists (and Intents, which are used later), add the following `using` directives to the top of **MainActivity.cs**:

```
using System.Collections.Generic;
using Android.Content;
```

Next, create an empty list that can be filled with phone numbers. The `MainActivity` class will look like this:

```
[Activity(Label = "Phoneword", MainLauncher = true)]
public class MainActivity : Activity
{
    static readonly List<string> phoneNumbers = new List<string>();
    ...// OnCreate, etc.
}
```

In the `MainActivity` class, add the following code to register the **TranslationHistory History** button (place this line after the `TranslationHistoryButton` declaration):

```
Button translationHistoryButton = FindViewById<Button> (Resource.Id.TranslationHistoryButton);
```

Add the following code to the end of the `OnCreate` method to wire up the **Translation History** button:

```
translationHistoryButton.Click += (sender, e) =>
{
    var intent = new Intent(this, typeof(TranslationHistoryActivity));
    intent.PutStringArrayListExtra("phone_numbers", phoneNumbers);
    StartActivity(intent);
};
```

Update the **Translate** button to add the phone number to the list of `phoneNumbers`. The `Click` handler for the `TranslateHistoryButton` should resemble the following code:

```
translateButton.Click += (sender, e) =>
{
    // Translate user's alphanumeric phone number to numeric
    translatedNumber = Core.PhonewordTranslator.ToNumber(phoneNumberText.Text);
    if (string.IsNullOrWhiteSpace(translatedNumber))
    {
        translatedPhoneWord.Text = "";
    }
    else
    {
        translatedPhoneWord.Text = translatedNumber;
        phoneNumbers.Add(translatedNumber);
        translationHistoryButton.Enabled = true;
    }
};
```

Running the app

Deploy the application to an emulator or device. The following screenshots illustrate the running **Phoneword** application:

Congratulations on completing your first multi-screen Xamarin.Android application! Now it's time to dissect the tools and skills you just learned – next up is the Hello, Android Multiscreen Deep Dive.

Related links

- Xamarin App Icons & Launch Screens (ZIP)
- Phoneword (sample)
- PhonewordMultiscreen (sample)

Hello, Android Multiscreen: Deep Dive

10/28/2019 • 6 minutes to read • Edit Online

In this two-part guide, the basic Phoneword application (created in the Hello, Android guide) is expanded to handle a second screen. Along the way, the basic Android application building blocks are introduced. A deeper dive into Android architecture is included to help you develop a better understanding of Android application structure and functionality.

In the Hello, Android Multiscreen Quickstart, you built and ran your first multi-screen Xamarin.Android application.

In this guide you will explore more advanced Android architecture. Android navigation with *Intents* is explained, and Android hardware navigation options are explored. New additions to the Phoneword app are dissected as you develop a more holistic view of the application's relationship with the operating system and other applications.

Android architecture basics

In the Hello, Android Deep Dive, you learned that Android applications are unique programs because they lack a single entry point. Instead, the operating system (or another application) starts any one of the application's registered Activities, which in turn starts the process for the application. This deep dive into Android architecture expands your understanding of how Android applications are constructed by introducing the Android Application Building Blocks and their functions.

Android application building blocks

An Android application consists of a collection of special Android classes called *Application Blocks* bundled together with any number of app resources - images, themes, helper classes, etc. – these are coordinated by an XML file called the *Android Manifest*.

Application Blocks form the backbone of Android applications because they allow you to do things you couldn't normally accomplish with a regular class. The two most important ones are *Activities* and *Services*:

- **Activity** – An Activity corresponds to a screen with a user interface, and it is conceptually similar to a web page in a web application. For example, in a newsfeed application, the login screen would be the first Activity, the scrollable list of news items would be another Activity, and the details page for each item would be a third. You can learn more about Activities in the Activity Lifecycle guide.

- **Service** – Android Services support Activities by taking over long-running tasks and running them in the background. Services don't have a user interface and are used to handle tasks that aren't tied to screens – for example, playing a song in the background or uploading photos to a server. For more information about Services, see the Creating Services and Android Services guides.

An Android application may not use all types of Blocks, and often has several Blocks of one type. For example, the Phoneword application from the Hello, Android Quickstart was composed of just one Activity (screen) and some resource files. A simple music player app might have several Activities and a Service for playing music when the app is in the background.

Intents

Another fundamental concept in Android applications is the *Intent*. Android is designed around the *principle of least privilege* – applications have access only to the Blocks they require to work, and they have limited access to the Blocks that make up the operating system or other applications. Similarly, Blocks are *loosely-coupled* – they are designed to have little knowledge of and limited access to other Blocks (even blocks that are part of the same application).

To communicate, Application Blocks send asynchronous messages called *Intents* back and forth. Intents contain information about the receiving Block and sometimes some data. An Intent sent from one App component triggers something to happen in another App component, binding the two App components and allowing them to communicate. By sending Intents back and forth, you can get Blocks to coordinate complex actions such as launching the camera app to take and save, gathering location information, or navigating from one screen to the next.

AndroidManifest.XML

When you add a Block to the application, it is registered with a special XML file called the **Android Manifest**. The Manifest keeps track of all Application Blocks in an application, as well as version requirements, permissions, and linked libraries – everything that the operating system needs to know for your application to run. The **Android Manifest** also works with Activities and Intents to control what actions are appropriate for a given Activity. These advanced features of the Android Manifest are covered in the Working with the Android Manifest guide.

In the single-screen version of the Phoneword application, only one Activity, one Intent, and the `AndroidManifest.xml` were used, alongside additional resources like icons. In the multi-screen version of Phoneword, an additional Activity was added; it was launched from the first Activity using an Intent. The next section explores how Intents help to create navigation in Android applications.

Android navigation

Intents were used to navigate between screens. It's time to dive into this code to see how Intents work and understand their role in Android navigation.

Launching a second activity with an intent

In the Phoneword application, an Intent was used to launch a second screen (Activity). Start by creating an Intent, passing in the current *Context* (`this` , referring to the current **Context**) and the type of Application Block that you're looking for (`TranslationHistoryActivity`):

```
Intent intent = new Intent(this, typeof(TranslationHistoryActivity));
```

The **Context** is an interface to global information about the application environment – it lets newly-created objects know what's going on with the application. If you think of an Intent as a message, you are providing the name of the message recipient (`TranslationHistoryActivity`) and the receiver's address (`Context`).

Android provides an option to attach simple data to an Intent (complex data is handled differently). In the Phoneword example, `PutStringArrayExtra` is used to attach a list of phone numbers to the Intent and `StartActivity` is called on the recipient of the Intent. The completed code looks like this:

```
translationHistoryButton.Click += (sender, e) =>
{
    var intent = new Intent(this, typeof(TranslationHistoryActivity));
    intent.PutStringArrayListExtra("phone_numbers", _phoneNumbers);
    StartActivity(intent);
};
```

Additional concepts introduced in phoneword

The Phoneword application introduced several concepts not covered in this guide. These concepts include:

String Resources – In the Phoneword application, the text of the `TranslationHistoryButton` was set to

`"@string/translationHistory"`. The `@string` syntax means that the string's value is stored in the *string resources file*, **Strings.xml**. The following value for the `translationHistory` string was added to **Strings.xml**:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="translationHistory">Call History</string>
</resources>
```

For more information on string resources and other Android resources, refer to the Android Resources guide.

ListView and ArrayAdapter – A *ListView* is a UI component that provides a simple way to present a scrolling list of rows. A `ListView` instance requires an *Adapter* to feed it with data contained in row views. The following line of code was used to populate the user interface of `TranslationHistoryActivity`:

```
this.ListAdapter = new ArrayAdapter<string>(this, Android.Resource.Layout.SimpleListItem1, phoneNumbers);
```

ListViews and Adapters are beyond the scope of this document, but they are covered in the very comprehensive ListViews and Adapters guide. Populating a ListView With Data deals specifically with using built-in `ListActivity` and `ArrayAdapter` classes to create and populate a `ListView` without defining a custom layout, as was done in the Phoneword example.

Summary

Congratulations, you've completed your first multi-screen Android application! This guide introduced *Android Application Building Blocks* and *Intents* and used them to build a multi-screened Android application. You now have the solid foundation you need to start developing your own Xamarin.Android applications.

Next, you'll learn to build cross-platform applications with Xamarin in the Building Cross-Platform Applications guides.

Xamarin for Java developers

11/2/2020 • 24 minutes to read • Edit Online

If you are a Java developer, you are well on your way to leveraging your skills and existing code on the Xamarin platform while reaping the code reuse benefits of C#. You will find that C# syntax is very similar to Java syntax, and that both languages provide very similar features. In addition, you'll discover features unique to C# that will make your development life easier.

Overview

This article provides an introduction to C# programming for Java developers, focusing primarily on the C# language features that you will encounter while developing Xamarin.Android applications. Also, this article explains how these features differ from their Java counterparts, and it introduces important C# features (relevant to Xamarin.Android) that are not available in Java. Links to additional reference material are included, so you can use this article as a "jumping off" point for further study of C# and .NET.

If you are familiar with Java, you will feel instantly at home with the syntax of C#. C# syntax is very similar to Java syntax – C# is a "curly brace" language like Java, C, and C++. In many ways, C# syntax reads like a superset of Java syntax, but with a few renamed and added keywords.

Many key characteristics of Java can be found in C#:

- Class-based object-oriented programming
- Strong typing
- Support for interfaces
- Generics
- Garbage collection
- Runtime compilation

Both Java and C# are compiled to an intermediate language that is run in a managed execution environment. Both C# and Java are statically-typed, and both languages treat strings as immutable types. Both languages use a single-rooted class hierarchy. Like Java, C# supports only single inheritance and does not allow for global methods. In both languages, objects are created on the heap using the `new` keyword, and objects are garbage-collected when they are no longer used. Both languages provide formal exception handling support with `try` / `catch` semantics. Both provide thread management and synchronization support.

However, there are many differences between Java and C#. For example:

- Java (as used on Android) does not support implicitly-typed local variables (C# supports the `var` keyword).
- In Java, you can pass parameters only by value, while in C# you can pass by reference as well as by value. (C# provides the `ref` and `out` keywords for passing parameters by reference; there is no equivalent to these in Java).
- Java does not support preprocessor directives like `#define`.
- Java does not support unsigned integer types, while C# provides unsigned integer types such as `ulong`, `uint`, `ushort` and `byte`.

- Java does not support operator overloading; in C# you can overload operators and conversions.
- In a Java `switch` statement, code can fall through into the next switch section, but in C# the end of every `switch` section must terminate the switch (the end of each section must close with a `break` statement).
- In Java, you specify the exceptions thrown by a method with the `throws` keyword, but C# has no concept of checked exceptions – the `throws` keyword is not supported in C#.
- C# supports Language-Integrated Query (LINQ), which lets you use the reserved words `from`, `select`, and `where` to write queries against collections in a way that is similar to database queries.

Of course, there are many more differences between C# and Java than can be covered in this article. Also, both Java and C# continue to evolve (for example, Java 8, which is not yet in the Android toolchain, supports C#-style lambda expressions) so these differences will change over time. Only the most important differences currently encountered by Java developers new to Xamarin.Android are outlined here.

- Going from Java to C# Development provides an introduction to the fundamental differences between C# and Java.
- Object-Oriented Programming Features outlines the most important object-oriented feature differences between the two languages.
- Keyword Differences provides a table of useful keyword equivalents, C#-only keywords, and links to C# keyword definitions.

C# brings many key features to Xamarin.Android that are not currently readily available to Java developers on Android. These features can help you to write better code in less time:

- Properties – With C#'s property system, you can access member variables safely and directly without having to write setter and getter methods.
- Lambda Expressions – In C# you can use anonymous methods (also called *lambdas*) to express your functionality more succinctly and more efficiently. You can avoid the overhead of having to write one-time-use objects, and you can pass local state to a method without having to add parameters.
- Event Handling – C# provides language-level support for *event-driven programming*, where an object can register to be notified when an event of interest occurs. The `event` keyword defines a multicast broadcast mechanism that a publisher class can use to notify event subscribers.
- Asynchronous Programming – The asynchronous programming features of C# (`async` / `await`) keep apps responsive. The language-level support of this feature makes async programming easy to implement and less error-prone.

Finally, Xamarin allows you to leverage existing Java assets via a technology known as *binding*. You can call your existing Java code, frameworks, and libraries from C# by making use of Xamarin's automatic binding generators. To do this, you simply create a static library in Java and expose it to C# via a binding.

> **NOTE**
>
> Android programming uses a specific version of the Java language that supports all Java 7 features and a subset of Java 8.
>
> Some features mentioned on this page (such as the `var` keyword in C#) are available in newer versions of Java (e.g. `var` in Java 10), but are still not available to Android developers.

Going from Java to C# development

The following sections outline the basic "getting started" differences between C# and Java; a later section describes the object-oriented differences between these languages.

Libraries vs. assemblies

Java typically packages related classes in **.jar** files. In C# and .NET, however, reusable bits of precompiled code are packaged into *assemblies*, which are typically packaged as *.dll* files. An assembly is a unit of deployment for C#/.NET code, and each assembly is typically associated with a C# project. Assemblies contain intermediate code (IL) that is just-in-time compiled at runtime.

For more information about assemblies, see the Assemblies and the Global Assembly Cache topic.

Packages vs. namespaces

C# uses the `namespace` keyword to group related types together; this is similar to Java's `package` keyword. Typically, a Xamarin.Android app will reside in a namespace created for that app. For example, the following C# code declares the `WeatherApp` namespace wrapper for a weather-reporting app:

```
namespace WeatherApp
{
    ...
```

Importing types

When you make use of types defined in external namespaces, you import these types with a `using` statement (which is very similar to the Java `import` statement). In Java, you might import a single type with a statement like the following:

```
import javax.swing.JButton
```

You might import an entire Java package with a statement like this:

```
import javax.swing.*
```

The C# `using` statement works in a very similar way, but it allows you to import an entire package without specifying a wildcard. For example, you will often see a series of `using` statements at the beginning of Xamarin.Android source files, as seen in this example:

```
using System;
using Android.App;
using Android.Content;
using Android.Runtime;
using Android.Views;
using Android.Widget;
using Android.OS;
using System.Net;
using System.IO;
using System.Json;
using System.Threading.Tasks;
```

These statements import functionality from the `System`, `Android.App`, `Android.Content`, etc. namespaces.

Generics

Both Java and C# support *generics*, which are placeholders that let you plug in different types at compile time. However, generics work slightly differently in C#. In Java, type erasure makes type information available only at compile time, but not at run time. By contrast, the .NET common language runtime (CLR) provides explicit support for generic types, which means that C# has access to type information at runtime. In day-to-day

Xamarin.Android development, the importance of this distinction is not often apparent, but if you are using reflection, you will depend on this feature to access type information at run time.

In Xamarin.Android, you will often see the generic method `FindViewById` used to get a reference to a layout control. This method accepts a generic type parameter that specifies the type of control to look up. For example:

```
TextView label = FindViewById<TextView> (Resource.Id.Label);
```

In this code example, `FindViewById` gets a reference to the `TextView` control that is defined in the layout as **Label**, then returns it as a `TextView` type.

For more information about generics, see the Generics topic. Note that there are some limitations in Xamarin.Android support for generic C# classes; for more information, see Limitations.

Object-oriented programming features

Both Java and C# use very similar object-oriented programming idioms:

- All classes are ultimately derived from a single root object – all Java objects derive from `java.lang.Object`, while all C# objects derive from `System.Object`.
- Instances of classes are reference types.
- When you access the properties and methods of an instance, you use the "`.`" operator.
- All class instances are created on the heap via the `new` operator.
- Because both languages use garbage collection, there is no way to explicitly release unused objects (i.e., there is not a `delete` keyword as there is in C++).
- You can extend classes through inheritance, and both languages only allow a single base class per type.
- You can define interfaces, and a class can inherit from (i.e., implement) multiple interface definitions.

However, there are also some important differences:

- Java has two powerful features that C# does not support: anonymous classes and inner classes. (However, C# does allow nesting of class definitions – C#'s nested classes are like Java's static nested classes.)
- C# supports C-style structure types (`struct`) while Java does not.
- In C#, you can implement a class definition in separate source files by using the `partial` keyword.
- C# interfaces cannot declare fields.
- C# uses C++-style destructor syntax to express finalizers. The syntax is different from Java's `finalize` method, but the semantics are nearly the same. (Note that in C#, destructors automatically call the base-class destructor – in contrast to Java where an explicit call to `super.finalize` is used.)

Class inheritance

To extend a class in Java, you use the `extends` keyword. To extend a class in C#, you use a colon (`:`) to indicate derivation. For example, in Xamarin.Android apps, you will often see class derivations that resemble the following code fragment:

```
public class MainActivity : Activity
{
    ...
```

In this example, `MainActivity` inherits from the `Activity` class.

To declare support for an interface in Java, you use the `implements` keyword. However, in C#, you simply add interface names to the list of classes to inherit from, as shown in this code fragment:

```
public class SensorsActivity : Activity, ISensorEventListener
{
    ...
```

In this example, `SensorsActivity` inherits from `Activity` and implements the functionality declared in the `ISensorEventListener` interface. Note that the list of interfaces must come after the base class (or you will get a compile-time error). By convention, C# interface names are prepended with an upper-case "I"; this makes it possible to determine which classes are interfaces without requiring an `implements` keyword.

When you want to prevent a class from being further subclassed in C#, you precede the class name with `sealed` – in Java, you precede the class name with `final`.

For more about C# class definitions, see the Classes and Inheritance topics.

Properties

In Java, mutator methods (setters) and inspector methods (getters) are often used to control how changes are made to class members while hiding and protecting these members from outside code. For example, the Android `TextView` class provides `getText` and `setText` methods. C# provides a similar but more direct mechanism known as *properties*. Users of a C# class can access a property in the same way as they would access a field, but each access actually results in a method call that is transparent to the caller. This "under the covers" method can implement side effects such as setting other values, performing conversions, or changing object state.

Properties are often used for accessing and modifying UI (user interface) object members. For example:

```
int width = rulerView.MeasuredWidth;
int height = rulerView.MeasuredHeight;
...
rulerView.DrawingCacheEnabled = true;
```

In this example, width and height values are read from the `rulerView` object by accessing its `MeasuredWidth` and `MeasuredHeight` properties. When these properties are read, values from their associated (but hidden) field values are fetched behind the scenes and returned to the caller. The `rulerView` object may store width and height values in one unit of measurement (say, pixels) and convert these values on-the-fly to a different unit of measurement (say, millimeters) when the `MeasuredWidth` and `MeasuredHeight` properties are accessed.

The `rulerView` object also has a property called `DrawingCacheEnabled` – the example code sets this property to `true` to enable the drawing cache in `rulerView`. Behind the scenes, an associated hidden field is updated with the new value, and possibly other aspects of `rulerView` state are modified. For example, when `DrawingCacheEnabled` is set to `false`, `rulerView` may also erase any drawing cache information already accumulated in the object.

Access to properties can be read/write, read-only, or write-only. Also, you can use different access modifiers for reading and writing. For example, you can define a property that has public read access but private write access.

For more information about C# properties, see the Properties topic.

Calling base class methods

To call a base-class constructor in C#, you use a colon (`:`) followed by the `base` keyword and an initializer list; this `base` constructor call is placed immediately after the derived constructor parameter list. The base-class constructor is called on entry to the derived constructor; the compiler inserts the call to the base constructor at the start of the method body. The following code fragment illustrates a base constructor called from a derived constructor in a Xamarin.Android app:

```
public class PictureLayout : ViewGroup
{
    ...
    public PictureLayout (Context context)
           : base (context)
    {
        ...
    }
    ...
}
```

In this example, the `PictureLayout` class is derived from the `ViewGroup` class. The `PictureLayout` constructor shown in this example accepts a `context` argument and passes it to the `ViewGroup` constructor via the `base(context)` call.

To call a base-class method in C#, use the `base` keyword. For example, Xamarin.Android apps often make calls to base methods as shown here:

```
public class MainActivity : Activity
{
    ...
    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);
```

In this case, the `OnCreate` method defined by the derived class (`MainActivity`) calls the `OnCreate` method of the base class (`Activity`).

Access modifiers

Java and C# both support the `public` , `private` , and `protected` access modifiers. However, C# supports two additional access modifiers:

- `internal` – The class member is accessible only within the current assembly.
- `protected internal` – The class member is accessible within the defining assembly, the defining class, and derived classes (derived classes both inside and outside the assembly have access).

For more information about C# access modifiers, see the Access Modifiers topic.

Virtual and override methods

Both Java and C# support *polymorphism*, the ability to treat related objects in the same manner. In both languages, you can use a base-class reference to refer to a derived-class object, and the methods of a derived class can override the methods of its base classes. Both languages have the concept of a *virtual* method, a method in a base class that is designed to be replaced by a method in a derived class. Like Java, C# supports `abstract` classes and methods.

However, there are some differences between Java and C# in how you declare virtual methods and override them:

- In C#, methods are non-virtual by default. Parent classes must explicitly label which methods are to be overridden by using the `virtual` keyword. By contrast, all methods in Java are virtual methods by default.

- To prevent a method from being overridden in C#, you simply leave off the `virtual` keyword. By contrast, Java uses the `final` keyword to mark a method with "override is not allowed."

- C# derived classes must use the `override` keyword to explicitly indicate that a virtual base-class method is being overridden.

For more information about C#'s support for polymorphism, see the Polymorphism topic.

Lambda expressions

C# makes it possible to create *closures*: inline, anonymous methods that can access the state of the method in which they are enclosed. Using lambda expressions, you can write fewer lines of code to implement the same functionality that you might have implemented in Java with many more lines of code.

Lambda expressions make it possible for you to skip the extra ceremony involved in creating a one-time-use class or anonymous class as you would in Java – instead, you can just write the business logic of your method code inline. Also, because lambdas have access to the variables in the surrounding method, you don't have to create a long parameter list to pass state to your method code.

In C#, lambda expressions are created with the `=>` operator as shown here:

```
(arg1, arg2, ...) => {
    // implementation code
};
```

In Xamarin.Android, lambda expressions are often used for defining event handlers. For example:

```
button.Click += (sender, args) => {
    clickCount += 1;    // access variable in surrounding code
    button.Text = string.Format ("Clicked {0} times.", clickCount);
};
```

In this example, the lambda expression code (the code within the curly braces) increments a click count and updates the `button` text to display the click count. This lambda expression is registered with the `button` object as a click event handler to be called whenever the button is tapped. (Event handlers are explained in more detail below.) In this simple example, the `sender` and `args` parameters are not used by the lambda expression code, but they are required in the lambda expression to meet the method signature requirements for event registration. Under the hood, the C# compiler translates the lambda expression into an anonymous method that is called whenever button click events take place.

For more information about C# and lambda expressions, see the Lambda Expressions topic.

Event handling

An *event* is a way for an object to notify registered subscribers when something interesting happens to that object. Unlike in Java, where a subscriber typically implements a `Listener` interface that contains a callback method, C# provides language-level support for event handling through *delegates*. A *delegate* is like an object-oriented type-safe function pointer – it encapsulates an object reference and a method token. If a client object wants to subscribe to an event, it creates a delegate and passes the delegate to the notifying object. When the event occurs, the notifying object invokes the method represented by the delegate object, notifying the subscribing client object of the event. In C#, event handlers are essentially nothing more than methods that are invoked through delegates.

For more information about delegates, see the Delegates topic.

In C#, events are *multicast*; that is, more than one listener can be notified when an event takes place. This difference is observed when you consider the syntactical differences between Java and C# event registration. In Java you call `SetXXXListener` to register for event notifications; in C# you use the `+=` operator to register for event notifications by "adding" your delegate to the list of event listeners. In Java, you call `SetXXXListener` to unregister, while in C# you use the `-=` to "subtract" your delegate from the list of listeners.

In Xamarin.Android, events are often used for notifying objects when a user does something to a UI control. Normally, a UI control will have members that are defined using the `event` keyword; you attach your delegates to these members to subscribe to events from that UI control.

To subscribe to an event:

1. Create a delegate object that refers to the method that you want to be invoked when the event occurs.
2. Use the `+=` operator to attach your delegate to the event you are subscribing to.

The following example defines a delegate (with an explicit use of the `delegate` keyword) to subscribe to button clicks. This button-click handler starts a new activity:

```
startActivityButton.Click += delegate {
    Intent intent = new Intent (this, typeof (MyActivity));
    StartActivity (intent);
};
```

However, you also can use a lambda expression to register for events, skipping the `delegate` keyword altogether. For example:

```
startActivityButton.Click += (sender, e) => {
    Intent intent = new Intent (this, typeof (MyActivity));
    StartActivity (intent);
};
```

In this example, the `startActivityButton` object has an event that expects a delegate with a certain method signature: one that accepts sender and event arguments and returns void. However, because we don't want to go to the trouble to explicitly define such a delegate or its method, we declare the signature of the method with `(sender, e)` and use a lambda expression to implement the body of the event handler. Note that we have to declare this parameter list even though we aren't using the `sender` and `e` parameters.

It is important to remember that you can unsubscribe a delegate (via the `-=` operator), but you cannot unsubscribe a lambda expression – attempting to do so can cause memory leaks. Use the lambda form of event registration only when your handler will not unsubscribe from the event.

Typically, lambda expressions are used to declare event handlers in Xamarin.Android code. This shorthand way to declare event handlers may seem cryptic at first, but it saves an enormous amount of time when you are writing and reading code. With increasing familiarity, you become accustomed to recognizing this pattern (which occurs frequently in Xamarin.Android code), and you spend more time thinking about the business logic of your application and less time wading through syntactical overhead.

Asynchronous programming

Asynchronous programming is a way to improve the overall responsiveness of your application. Asynchronous programming features make it possible for the rest of your app code to continue running while some part of your app is blocked by a lengthy operation. Accessing the web, processing images, and reading/writing files are examples of operations that can cause an entire app to appear to freeze up if it is not written asynchronously.

C# includes language-level support for asynchronous programming via the `async` and `await` keywords. These language features make it very easy to write code that performs long-running tasks without blocking the main thread of your application. Briefly, you use the `async` keyword on a method to indicate that the code in the method is to run asynchronously and not block the caller's thread. You use the `await` keyword when you call methods that are marked with `async`. The compiler interprets the `await` as the point where your method execution is to be moved to a background thread (a task is returned to the caller). When this task completes, execution of the code resumes on the caller's thread at the `await` point in your code, returning the results of the `async` call. By convention, methods that run asynchronously have `Async` suffixed to their names.

In Xamarin.Android applications, `async` and `await` are typically used to free up the UI thread so that it can respond to user input (such as the tapping of a **Cancel** button) while a long-running operation takes place in a background task.

In the following example, a button click event handler causes an asynchronous operation to download an image from the web:

```
downloadButton.Click += downloadAsync;
...
async void downloadAsync(object sender, System.EventArgs e)
{
    webClient = new WebClient ();
    var url = new Uri ("http://photojournal.jpl.nasa.gov/jpeg/PIA15416.jpg");
    byte[] bytes = null;

    bytes = await webClient.DownloadDataTaskAsync(url);

    // display the downloaded image ...
```

In this example, when the user clicks the `downloadButton` control, the `downloadAsync` event handler creates a `WebClient` object and a `Uri` object to fetch an image from the specifed URL. Next, it calls the `WebClient` object's `DownloadDataTaskAsync` method with this URL to retrieve the image.

Notice that the method declaration of `downloadAsync` is prefaced by the `async` keyword to indicate that it will run asynchronously and return a task. Also note that the call to `DownloadDataTaskAsync` is preceded by the `await` keyword. The app moves the execution of the event handler (starting at the point where `await` appears) to a background thread until `DownloadDataTaskAsync` completes and returns. Meanwhile, the app's UI thread can still respond to user input and fire event handlers for the other controls. When `DownloadDataTaskAsync` completes (which may take several seconds), execution resumes where the `bytes` variable is set to the result of the call to `DownloadDataTaskAsync`, and the remainder of the event handler code displays the downloaded image on the caller's (UI) thread.

For an introduction to `async`/`await` in C#, see the Asynchronous Programming with Async and Await topic. For more information about Xamarin support of asynchronous programming features, see Async Support Overview.

Keyword differences

Many language keywords used in Java are also used in C#. There are also a number of Java keywords that have an equivalent but differently-named counterpart in C#, as listed in this table:

JAVA	C#	DESCRIPTION
`boolean`	bool	Used for declaring the boolean values true and false.
`extends`	`:`	Precedes the class and interfaces to inherit from.
`implements`	`:`	Precedes the class and interfaces to inherit from.
`import`	using	Imports types from a namespace, also used for creating a namespace alias.
`final`	sealed	Prevents class derivation; prevents methods and properties from being overridden in derived classes.
`instanceof`	is	Evaluates whether an object is compatible with a given type.
`native`	extern	Declares a method that is implemented externally.
`package`	namespace	Declares a scope for a related set of objects.
`T...`	params T	Specifies a method parameter that takes a variable number of arguments.
`super`	base	Used to access members of the parent class from within a derived class.
`synchronized`	lock	Wraps a critical section of code with lock acquisition and release.

Also, there are many keywords that are unique to C# and have no counterpart in the Java used on Android. Xamarin.Android code often makes use of the following C# keywords (this table is useful to refer to when you are reading through Xamarin.Android sample code):

C#	DESCRIPTION
as	Performs conversions between compatible reference types or nullable types.
async	Specifies that a method or lambda expression is asynchronous.
await	Suspends the execution of a method until a task completes.
byte	Unsigned 8-bit integer type.
delegate	Used to encapsulate a method or anonymous method.
enum	Declares an enumeration, a set of named constants.

C#	DESCRIPTION
event	Declares an event in a publisher class.
fixed	Prevents a variable from being relocated.
`get`	Defines an accessor method that retrieves the value of a property.
in	Enables a parameter to accept a less derived type in a generic interface.
object	An alias for the Object type in the .NET framework.
out	Parameter modifier or generic type parameter declaration.
override	Extends or modifies the implementation of an inherited member.
partial	Declares a definition to be split into multiple files, or splits a method definition from its implementation.
readonly	Declares that a class member can be assigned only at declaration time or by the class constructor.
ref	Causes an argument to be passed by reference rather than by value.
set	Defines an accessor method that sets the value of a property.
string	Alias for the String type in the .NET framework.
struct	A value type that encapsulates a group of related variables.
typeof	Obtains the type of an object.
var	Declares an implicitly-typed local variable.
value	References the value that client code wants to assign to a property.
virtual	Allows a method to be overridden in a derived class.

Interoperating with existing java code

If you have existing Java functionality that you do not want to convert to C#, you can reuse your existing Java libraries in Xamarin.Android applications via two techniques:

- **Create a Java Bindings Library** – Using this approach, you use Xamarin tools to generate C# wrappers around Java types. These wrappers are called *bindings*. As a result, your Xamarin.Android application can use your *.jar* file by calling into these wrappers.
- **Java Native Interface** – The *Java Native Interface* (JNI) is a framework that makes it possible for C#

apps to call or be called by Java code.

For more information about these techniques, see Java Integration Overview.

Further reading

The MSDN C# Programming Guide is a great way to get started in learning the C# programming language, and you can use the C# Reference to look up particular C# language features.

In the same way that Java knowledge is at least as much about familiarity with the Java class libraries as knowing the Java language, practical knowledge of C# requires some familiarity with the .NET framework. Microsoft's Moving to C# and the .NET Framework, for Java Developers learning packet is a good way to learn more about the .NET framework from a Java perspective (while gaining a deeper understanding of C#).

When you are ready to tackle your first Xamarin.Android project in C#, our Hello, Android series can help you build your first Xamarin.Android application and further advance your understanding of the fundamentals of Android application development with Xamarin.

Summary

This article provided an introduction to the Xamarin.Android C# programming environment from a Java developer's perspective. It pointed out the similarities between C# and Java while explaining their practical differences. It introduced assemblies and namespaces, explained how to import external types, and provided an overview of the differences in access modifiers, generics, class derivation, calling base-class methods, method overriding, and event handling. It introduced C# features that are not available in Java, such as properties, `async` / `await` asynchronous programming, lambdas, C# delegates, and the C# event handling system. It included tables of important C# keywords, explained how to interoperate with existing Java libraries, and provided links to related documentation for further study.

Related links

- Java Integration Overview
- C# Programming Guide
- C# Reference
- Moving to C# and the .NET Framework, for Java Developers

Xamarin.Android Application Fundamentals

10/28/2019 • 3 minutes to read • Edit Online

This section provides a guide on some of the more common things tasks or concepts that developers need to be aware of when developing Android applications.

Accessibility

This page describes how to use the Android Accessibility APIs to build apps according to the accessibility checklist.

Understanding Android API Levels

This guide describes how Android uses API levels to manage app compatibility across different versions of Android, and it explains how to configure Xamarin.Android project settings to deploy these API levels in your app. In addition, this guide explains how to write runtime code that deals with different API levels, and it provides a reference list of all Android API levels, version numbers (such as Android 8.0), Android code names (such as Oreo), and build version codes.

Resources in Android

This article introduces the concept of Android resources in Xamarin.Android and documents how to use them. It covers how to use resources in your Android application to support application localization, and multiple devices including varying screen sizes and densities.

Activity Lifecycle

Activities are a fundamental building block of Android Applications and they can exist in a number of different states. The activity lifecycle begins with instantiation and ends with destruction, and includes many states in between. When an activity changes state, the appropriate lifecycle event method is called, notifying the activity of the impending state change and allowing it to execute code to adapt to that change. This article examines the lifecycle of activities and explains the responsibility that an activity has during each of these state changes to be part of a well-behaved, reliable application.

Localization

This article explains how to localize a Xamarin.Android into other languages by translating strings and providing alternate images.

Services

This article covers Android services, which are Android components that allow work to be done in the background. It explains the different scenarios that services are suited for and shows how to implement them both for performing long-running background tasks as well as to provide an interface for remote procedure calls.

Broadcast Receivers

This guide covers how to create and use broadcast receivers, an Android component that responds to system-wide broadcasts, in Xamarin.Android.

Permissions

You can use the tooling support built into Visual Studio for Mac or Visual Studio to create and add permissions to the Android Manifest. This document describes how to add permissions in Visual Studio and Xamarin Studio.

Graphics and Animation

Android provides a very rich and diverse framework for supporting 2D graphics and animations. This document introduces these frameworks and discusses how to create custom graphics and animations and use them in a Xamarin.Android application.

CPU Architectures

Xamarin.Android supports several CPU architectures, including 32-bit and 64-bit devices. This article explains how to target an app to one or more Android-supported CPU architectures.

Handling Rotation

This article describes how to handle device orientation changes in Xamarin.Android. It covers how to work with the Android resource system to automatically load resources for a particular device orientation as well as how to programmatically handle orientation changes. Then it describes techniques for maintaining state when a device is rotated.

Android Audio

The Android OS provides extensive support for multimedia, encompassing both audio and video. This guide focuses on audio in Android and covers playing and recording audio using the built-in audio player and recorder classes, as well as the low-level audio API. It also covers working with Audio events broadcast by other applications, so that developers can build well-behaved applications.

Notifications

This section explains how to implement local and remote notifications in Xamarin.Android. It describes the various UI elements of an Android notification and discusses the API's involved with creating and displaying a notification. For remote notifications, both Google Cloud Messaging and Firebase Cloud Messaging are explained. Step-by-step walkthroughs and code samples are included.

Touch

This section explains the concepts and details of implementing touch gestures on Android. Touch APIs are introduced and explained followed by an exploration of gesture recognizers.

HttpClient Stack and SSL/TLS

This section explains the HttpClient Stack and SSL/TLS Implementation selectors for Android. These settings determine the HttpClient and SSL/TLS implementation that will be used by your Xamarin.Android apps.

Writing Responsive Applications

This article discusses how to use threading to keep a Xamarin.Android application responsive by moving long-running tasks on to a background thread.

Accessibility on Android

10/28/2019 • 3 minutes to read • Edit Online

This page describes how to use the Android Accessibility APIs to build apps according to the accessibility checklist. Refer to the iOS accessibility and OS X accessibility pages for other platform APIs.

Describing UI Elements

Android provides a `ContentDescription` property that is used by screen reading APIs to provide an accessible description of the control's purpose.

The content description can be set in either C# or in the AXML layout file.

C#

The description can be set in code to any string (or a string resource):

```
saveButton.ContentDescription = "Save data";
```

AXML layout

In XML layouts use the `android:contentDescription` attribute:

```
<ImageButton
    android:id=@+id/saveButton"
    android:src="@drawable/save_image"
    android:contentDescription="Save data" />
```

Use Hint for TextView

For `EditText` and `TextView` controls for data input, use the `Hint` property to provide a description of what input is expected (instead of `ContentDescription`). When some text has been entered, the text itself will be "read" instead of the hint.

C#

Set the `Hint` property in code:

```
someText.Hint = "Enter some text"; // displays (and is "read") when control is empty
```

AXML layout

In XML layout files use the `android:hint` attribute:

```
<EditText
    android:id="@+id/someText"
    android:hint="Enter some text" />
```

LabelFor links input fields with labels

To associate a label with a data input control, use the `LabelFor` property to

C#

In C#, set the `LabelFor` property to the resource ID of the control that this content describes (typically this property is set on a label and references some other input control):

```
EditText edit = FindViewById<EditText> (Resource.Id.editFirstName);
TextView tv = FindViewById<TextView> (Resource.Id.labelFirstName);
tv.LabelFor = Resource.Id.editFirstName;
```

AXML layout

In layout XML use the `android:labelFor` property to reference another control's identifier:

```
<TextView
    android:id="@+id/labelFirstName"
    android:hint="Enter some text"
    android:labelFor="@+id/editFirstName" />
<EditText
    android:id="@+id/editFirstName"
    android:hint="Enter some text" />
```

Announce for Accessibility

Use the `AnnounceForAccessibility` method on any view control to communicate an event or status change to users when accessibility is enabled. This method isn't required for most operations where the built-in narration provides sufficient feedback, but should be used where additional information would be helpful for the user.

The code below shows a simple example calling `AnnounceForAccessibility`:

```
button.Click += delegate {
  button.Text = string.Format ("{0} clicks!", count++);
  button.AnnounceForAccessibility (button.Text);
};
```

Changing Focus Settings

Accessible navigation relies on controls having focus to aid the user in understanding what operations are available. Android provides a `Focusable` property which can tag controls as specifically able to receive focus during navigation.

C#

To prevent a control from gaining focus with C#, set the `Focusable` property to `false`:

```
label.Focusable = false;
```

AXML layout

In layout XML files set the `android:focusable` attribute:

```
<android:focusable="false" />
```

You can also control focus order with the `nextFocusDown`, `nextFocusLeft`, `nextFocusRight`, `nextFocusUp` attributes, typically set in the layout AXML. Use these attributes to ensure the user can navigate easily through the controls on the screen.

Accessibility and Localization

In the examples above the hint and content description are set directly to the display value. It is preferable to use values in a **Strings.xml** file, such as this:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="enter_info">Enter some text</string>
    <string name="save_info">Save data</string>
</resources>
```

Using text from a strings file is shown below in C# and AXML layout files:

C#

Instead of using string literals in code, look up translated values from strings files with `Resources.GetText`:

```
someText.Hint = Resources.GetText (Resource.String.enter_info);
saveButton.ContentDescription = Resources.GetText (Resource.String.save_info);
```

AXML

In layout XML accessibility attributes like `hint` and `contentDescription` can be set to a string identifier:

```
<TextView
    android:id="@+id/someText"
    android:hint="@string/enter_info" />
<ImageButton
    android:id=@+id/saveButton"
    android:src="@drawable/save_image"
    android:contentDescription="@string/save_info" />
```

The benefit of storing text in a separate file is multiple language translations of the file can be provided in your app. See the Android localization guide to learn how add localized string files to an application project.

Testing Accessibility

Follow these steps to enable TalkBack and Explore by Touch to test accessibility on Android devices.

You may need to install TalkBack from Google Play if it does not appear in **Settings > Accessibility**.

Related Links

- Cross-platform Accessibility
- Android Accessibility APIs

Understanding Android API levels

7/8/2021 • 18 minutes to read • Edit Online

Xamarin.Android has several Android API level settings that determine your app's compatibility with multiple versions of Android. This guide explains what these settings mean, how to configure them, and what effect they have on your app at run time.

Quick start

Xamarin.Android exposes three Android API level project settings:

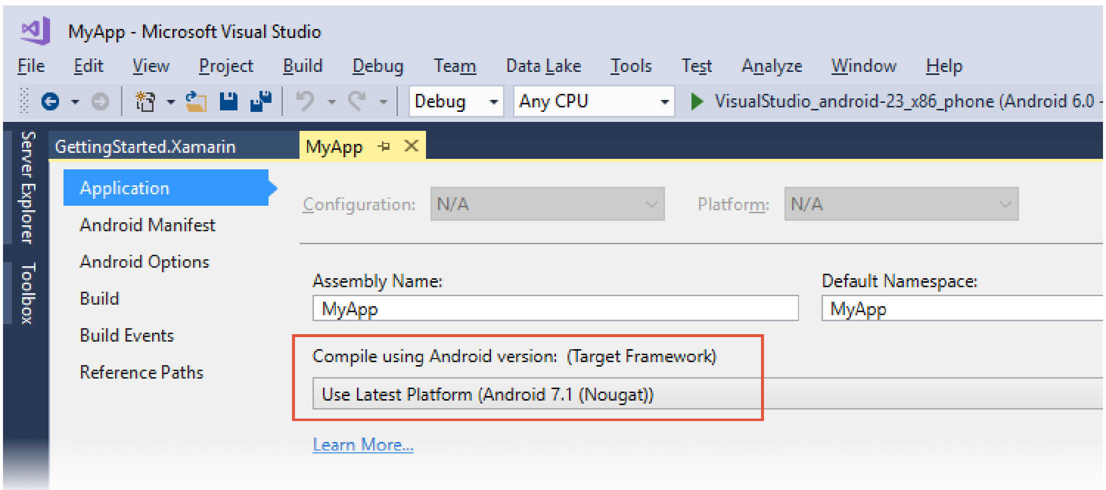
- Target Framework – Specifies which framework to use in building your application. This API level is used at *compile* time by Xamarin.Android.
- Minimum Android Version – Specifies the oldest Android version that you want your app to support. This API level is used at *run* time by Android.
- Target Android Version – Specifies the version of Android that your app is intended to run on. This API level is used at *run* time by Android.

Before you can configure an API level for your project, you must install the SDK platform components for that API level. For more information about downloading and installing Android SDK components, see Android SDK Setup.

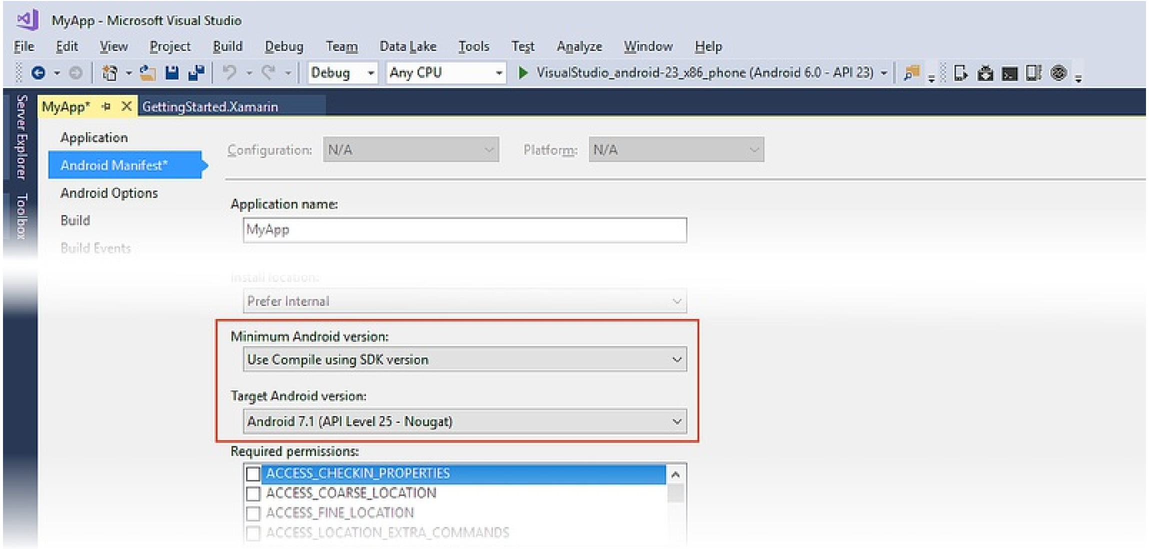
> **NOTE**
>
> Beginning in August 2020, the Google Play Console requires that new apps target API level 29 (Android 10.0) or higher. Existing apps are required to target API level 29 or higher beginning in November 2020. For more information, see Target API level requirements for the Play Console in "Create and set up your app" in the Play Console documentation.

- Visual Studio
- Visual Studio for Mac

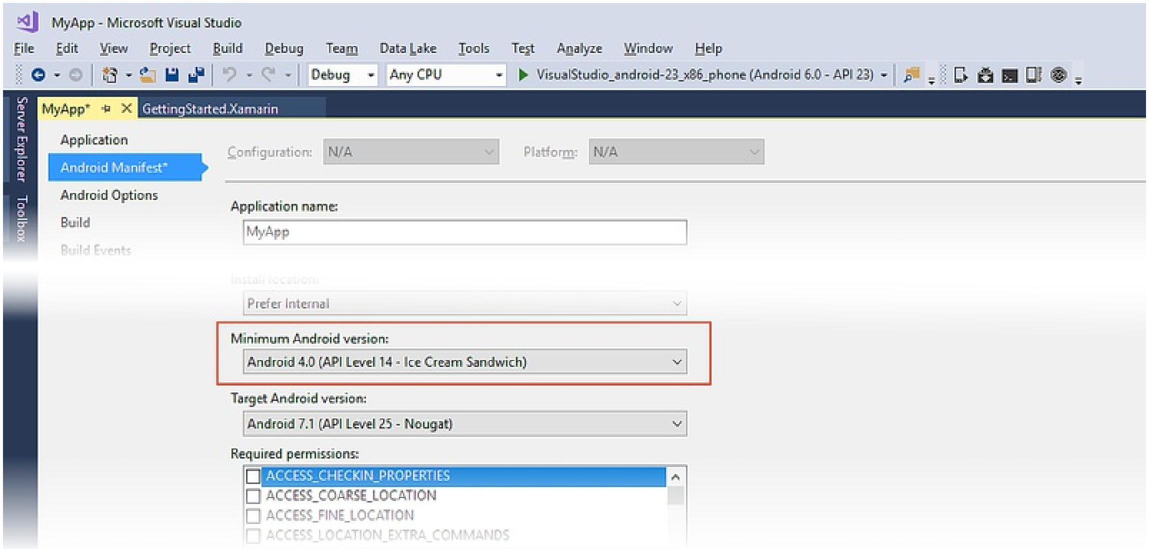
Normally, all three Xamarin.Android API levels are set to the same value. On the **Application** page, set **Compile using Android version (Target Framework)** to the latest stable API version (or, at a minimum, to the Android version that has all of the features you need). In the following screenshot, the Target Framework is set to **Android 7.1 (API Level 25 - Nougat)**:



On the **Android Manifest** page, set the Minimum Android version to **Use Compile using SDK version** and set the Target Android version to the same value as the Target Framework version (in the following screenshot, the Target Android Framework is set to **Android 7.1 (Nougat)**):



If you want to maintain backward compatibility with an earlier version of Android, set **Minimum Android version to target** to the oldest version of Android that you want your app to support. (Note that API Level 14 is the minimum API level required for Google Play services and Firebase support.) The following example configuration supports Android versions from API Level 14 through API level 25:



If your app supports multiple Android versions, your code must include runtime checks to ensure that your app works with the Minimum Android version setting (see Runtime Checks for Android Versions below for details). If you are consuming or creating a library, see API Levels and Libraries below for best practices in configuring API level settings for libraries.

Android versions and API levels

As the Android platform evolves and new Android versions are released, each Android version is assigned a unique integer identifier, called the *API Level*. Therefore, each Android version corresponds to a single Android API Level. Because users install apps on older as well as the most recent versions of Android, real-world Android apps must be designed to work with multiple Android API levels.

Android versions

Each release of Android goes by multiple names:

- The Android version, such as **Android 9.0**
- A code (or dessert) name, such as *Pie*
- A corresponding API level, such as **API level 28**

An Android code name may correspond to multiple versions and API levels (as seen in the table below), but each

Android version corresponds to exactly one API level.

In addition, Xamarin.Android defines *build version codes* that map to the currently known Android API levels. The following table can help you translate between API level, Android version, code name, and Xamarin.Android build version code (build version codes are defined in the `Android.OS` namespace):

NAME	VERSION	API LEVEL	RELEASED	BUILD VERSION CODE
Q	10.0	29	Aug 2020	`BuildVersionCodes.Q`
Pie	9.0	28	Aug 2018	`BuildVersionCodes.P`
Oreo	8.1	27	Dec 2017	`BuildVersionCodes.OMr1`
Oreo	8.0	26	Aug 2017	`BuildVersionCodes.O`
Nougat	7.1	25	Dec 2016	`BuildVersionCodes.NMr1`
Nougat	7.0	24	Aug 2016	`BuildVersionCodes.N`
Marshmallow	6.0	23	Aug 2015	`BuildVersionCodes.M`
Lollipop	5.1	22	Mar 2015	`BuildVersionCodes.LollipopMr1`
Lollipop	5.0	21	Nov 2014	`BuildVersionCodes.Lollipop`
Kitkat Watch	4.4W	20	Jun 2014	`BuildVersionCodes.KitKatWatch`
Kitkat	4.4	19	Oct 2013	`BuildVersionCodes.KitKat`
Jelly Bean	4.3	18	Jul 2013	`BuildVersionCodes.JellyBeanMr2`
Jelly Bean	4.2-4.2.2	17	Nov 2012	`BuildVersionCodes.JellyBeanMr1`
Jelly Bean	4.1-4.1.1	16	Jun 2012	`BuildVersionCodes.JellyBean`
Ice Cream Sandwich	4.0.3-4.0.4	15	Dec 2011	`BuildVersionCodes.IceCreamSandwi`
Ice Cream Sandwich	4.0-4.0.2	14	Oct 2011	`BuildVersionCodes.IceCreamSandwi`
Honeycomb	3.2	13	Jun 2011	`BuildVersionCodes.HoneyCombMr2`
Honeycomb	3.1.x	12	May 2011	`BuildVersionCodes.HoneyCombMr1`
Honeycomb	3.0.x	11	Feb 2011	`BuildVersionCodes.HoneyComb`
Gingerbread	2.3.3-2.3.4	10	Feb 2011	`BuildVersionCodes.GingerBreadMr1`
Gingerbread	2.3-2.3.2	9	Nov 2010	`BuildVersionCodes.GingerBread`
Froyo	2.2.x	8	Jun 2010	`BuildVersionCodes.Froyo`
Eclair	2.1.x	7	Jan 2010	`BuildVersionCodes.EclairMr1`
Eclair	2.0.1	6	Dec 2009	`BuildVersionCodes.Eclair01`

NAME	VERSION	API LEVEL	RELEASED	BUILD VERSION CODE
Eclair	2.0	5	Nov 2009	`BuildVersionCodes.Eclair`
Donut	1.6	4	Sep 2009	`BuildVersionCodes.Donut`
Cupcake	1.5	3	May 2009	`BuildVersionCodes.Cupcake`
Base	1.1	2	Feb 2009	`BuildVersionCodes.Base11`
Base	1.0	1	Oct 2008	`BuildVersionCodes.Base`

As this table indicates, new Android versions are released frequently – sometimes more than one release per year. As a result, the universe of Android devices that might run your app includes of a wide variety of older and newer Android versions. How can you guarantee that your app will run consistently and reliably on so many different versions of Android? Android's API levels can help you manage this problem.

Android API levels

Each Android device runs at exactly *one* API level – this API level is guaranteed to be unique per Android platform version. The API level precisely identifies the version of the API set that your app can call into; it identifies the combination of manifest elements, permissions, etc. that you code against as a developer. Android's system of API levels helps Android determine whether an application is compatible with an Android system image prior to installing the application on a device.

When an application is built, it contains the following API level information:

- The *target* API level of Android that the app is built to run on.
- The *minimum* Android API level that an Android device must have to run your app.

These settings are used to ensure that the functionality needed to run the app correctly is available on the Android device at installation time. If not, the app is blocked from running on that device. For example, if the API level of an Android device is lower than the minimum API level that you specify for your app, the Android device will prevent the user from installing your app.

Project API level settings

The following sections explain how to use the SDK Manager to prepare your development environment for the API levels you want to target, followed by detailed explanations of how to configure *Target Framework*, *Minimum Android version*, and *Target Android version* settings in Xamarin.Android.

Android SDK platforms

Before you can select a Target or Minimum API level in Xamarin.Android, you must install the Android SDK platform version that corresponds to that API level. The range of available choices for Target Framework, Minimum Android version, and Target Android version is limited to the range of Android SDK versions that you have installed. You can use the SDK Manager to verify that the required Android SDK versions are installed, and you can use it to add any new API levels that you need for your app. If you are not familiar with how to install API levels, see Android SDK Setup.

Target Framework

The *Target Framework* (also known as `compileSdkVersion`) is the specific Android framework version (API level) that your app is compiled for at build time. This setting specifies what APIs your app *expects* to use when it runs, but it has no effect on which APIs are actually available to your app when it is installed. As a result, changing the Target Framework setting does not change runtime behavior.

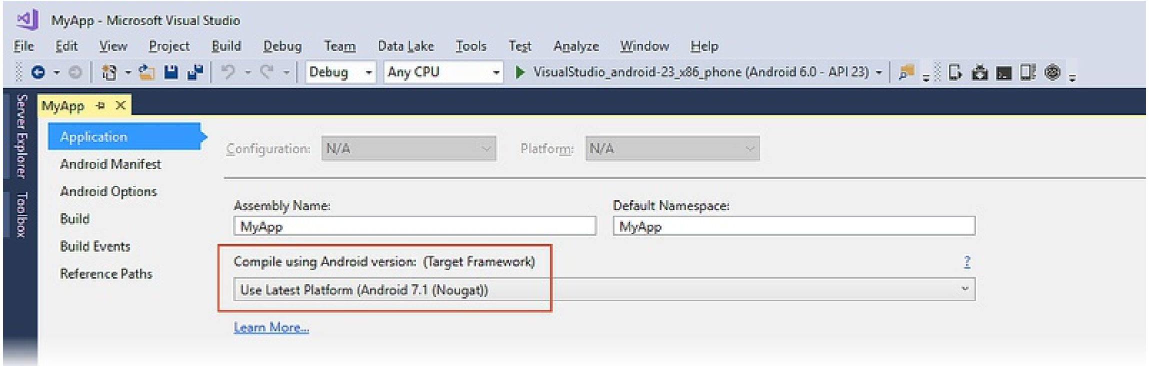
The Target Framework identifies which library versions your application is linked against – this setting determines which APIs you can use in your app. For example, if you want to use the NotificationBuilder.SetCategory method that was introduced in Android 5.0 Lollipop, you must set the Target

Framework to **API Level 21 (Lollipop)** or later. If you set your project's Target Framework to an API level such as **API Level 19 (KitKat)** and try to call the `SetCategory` method in your code, you will get a compile error.

We recommend that you always compile with the *latest* available Target Framework version. Doing so provides you with helpful warning messages for any deprecated APIs that might be called by your code. Using the latest Target Framework version is especially important when you use the latest support library releases – each library expects your app to be compiled at that support library's minimum API level or greater.

- Visual Studio
- Visual Studio for Mac

To access the Target Framework setting in Visual Studio, open the project properties in **Solution Explorer** and select the **Application** page:



Set the Target Framework by selecting an API level in the drop-down menu under **Compile using Android version** as shown above.

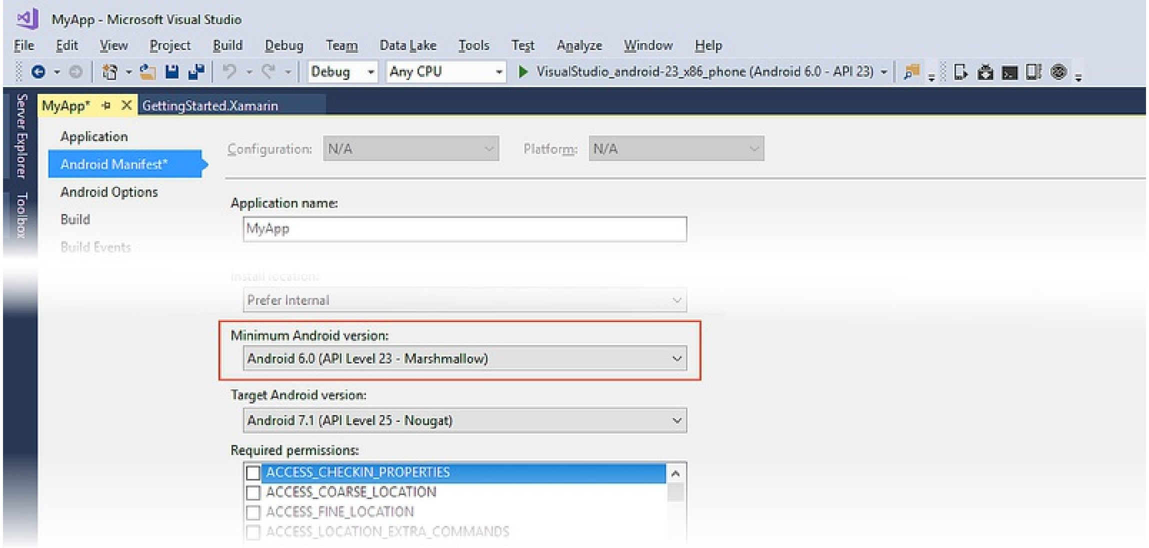
Minimum Android Version

The *Minimum Android version* (also known as `minSdkVersion`) is the oldest version of the Android OS (i.e., the lowest API level) that can install and run your application. By default, an app can only be installed on devices matching the Target Framework setting or higher; if the Minimum Android version setting is *lower* than the Target Framework setting, your app can also run on earlier versions of Android. For example, if you set the Target Framework to **Android 7.1 (Nougat)** and set the Minimum Android version to **Android 4.0.3 (Ice Cream Sandwich)**, your app can be installed on any platform from API level 15 to API level 25, inclusive.

Although your app may successfully build and install on this range of platforms, this does not guarantee that it will successfully *run* on all of these platforms. For example, if your app is installed on **Android 5.0 (Lollipop)** and your code calls an API that is available only in **Android 7.1 (Nougat)** and newer, your app will get a runtime error and possibly crash. Therefore, your code must ensure – at runtime – that it calls only those APIs that are supported by the Android device that it is running on. In other words, your code must include explicit runtime checks to ensure that your app uses newer APIs only on devices that are recent enough to support them. Runtime Checks for Android Versions, later in this guide, explains how to add these runtime checks to your code.

- Visual Studio
- Visual Studio for Mac

To access the Minimum Android version setting in Visual Studio, open the project properties in **Solution Explorer** and select the **Android Manifest** page. In the drop-down menu under **Minimum Android version** you can select the Minimum Android version for your application:



If you select **Use Compile using SDK version**, the Minimum Android version will be the same as the Target Framework setting.

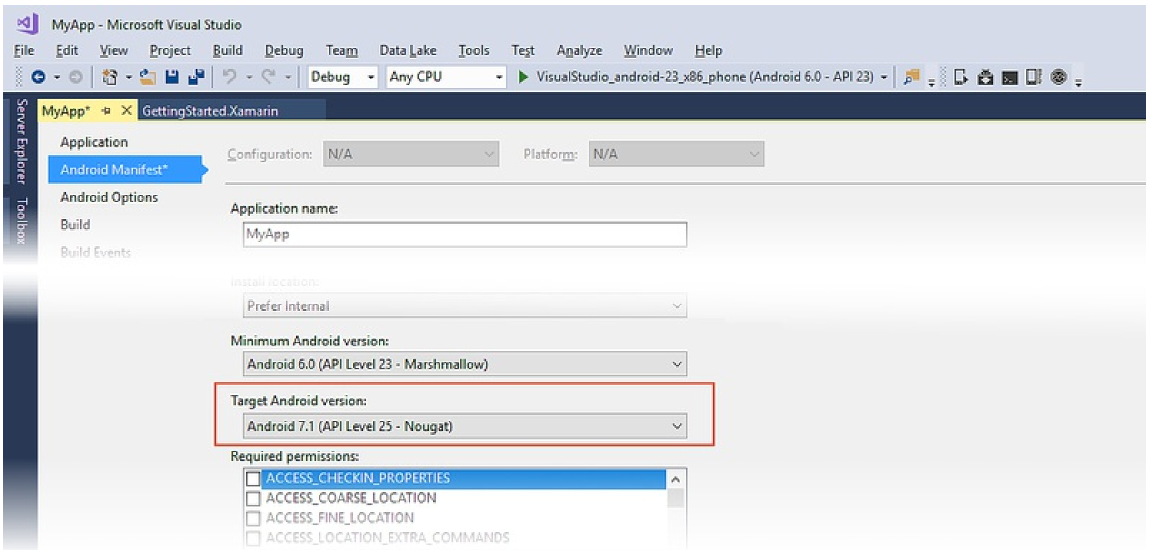
Target Android Version

The *Target Android Version* (also known as `targetSdkVersion`) is the API level of the Android device where the app expects to run. Android uses this setting to determine whether to enable any compatibility behaviors – this ensures that your app continues to work the way you expect. Android uses the Target Android version setting of your app to figure out which behavior changes can be applied to your app without breaking it (this is how Android provides forward compatibility).

The Target Framework and the Target Android version, while having very similar names, are not the same thing. The Target Framework setting communicates target API level information to Xamarin.Android for use at *compile time*, while the Target Android version communicates target API level information to Android for use at *run time* (when the app is installed and running on a device).

- Visual Studio
- Visual Studio for Mac

To access this setting in Visual Studio, open the project properties in **Solution Explorer** and select the **Android Manifest** page. In the drop-down menu under **Target Android version** you can select the Target Android version for your application:



We recommend that you explicitly set the Target Android version to the latest version of Android that you use to test your app. Ideally, it should be set to the latest Android SDK version – this allows you to use new APIs prior to working through the behavior changes. For most developers, we *do not* recommend setting the Target Android version to **Use Compile using SDK version**.

In general, the Target Android Version should be bounded by the Minimum Android Version and the Target Framework. That is:

Minimum Android Version <= Target Android Version <= Target Framework

For more information about SDK levels, see the Android Developer uses-sdk documentation.

Runtime checks for Android versions

As each new version of Android is released, the framework API is updated to provide new or replacement functionality. With few exceptions, API functionality from earlier Android versions is carried forward into newer Android versions without modifications. As a result, if your app runs on a particular Android API level, it will typically be able to run on a later Android API level without modifications. But what if you also want to run your app on earlier versions of Android?

If you select a Minimum Android version that is *lower* than your Target Framework setting, some APIs may not be available to your app at runtime. However, your app can still run on an earlier device, but with reduced functionality. For each API that is not available on Android platforms corresponding to your Minimum Android version setting, your code must explicitly check the value of the `Android.OS.Build.VERSION.SdkInt` property to determine the API level of the platform the app is running on. If the API level is *lower* than the Minimum Android version that supports the API you want to call, then your code has to find a way to function properly without making this API call.

For example, let's suppose that we want to use the NotificationBuilder.SetCategory method to categorize a notification when running on **Android 5.0 Lollipop** (and later), but we still want our app to run on earlier versions of Android such as **Android 4.1 Jelly Bean** (where `SetCategory` is not available). Referring to the Android version table at the beginning of this guide, we see that the build version code for **Android 5.0 Lollipop** is `Android.OS.BuildVersionCodes.Lollipop`. To support older versions of Android where `SetCategory` is not available, our code can detect the API level at runtime and conditionally call `SetCategory` only when the API level is greater than or equal to the Lollipop build version code:

```
if (Android.OS.Build.VERSION.SdkInt >= Android.OS.BuildVersionCodes.Lollipop)
{
    builder.SetCategory(Notification.CategoryEmail);
}
```

In this example, our app's Target Framework is set to **Android 5.0 (API Level 21)** and its Minimum Android version is set to **Android 4.1 (API Level 16)**. Because `SetCategory` is available in API level `Android.OS.BuildVersionCodes.Lollipop` and later, this example code will call `SetCategory` only when it is actually available – it will *not* attempt to call `SetCategory` when the API level is 16, 17, 18, 19, or 20. The functionality is reduced on these earlier Android versions only to the extent that notifications are not sorted properly (because they are not categorized by type), yet the notifications are still published to alert the user. Our app still works, but its functionality is slightly diminished.

In general, the build version check helps your code decide at runtime between doing something the new way versus the old way. For example:

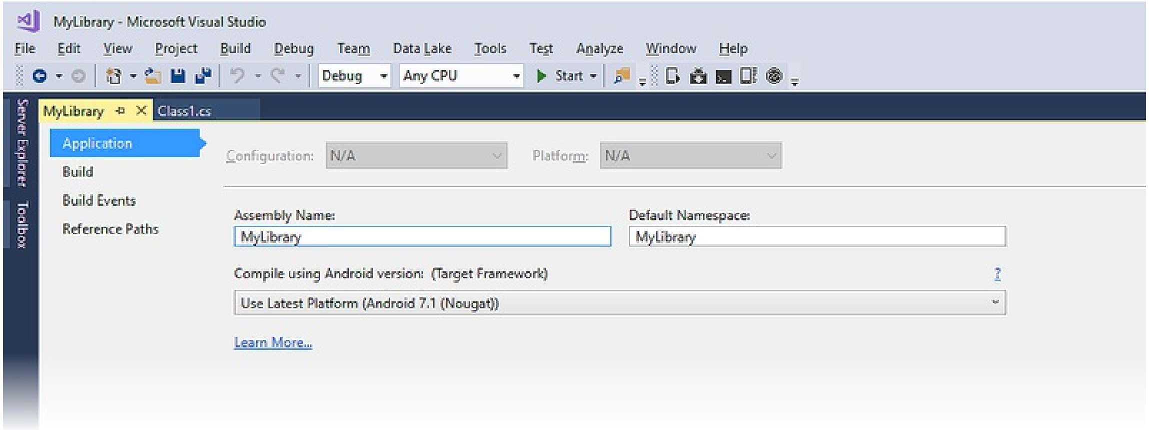
```
if (Android.OS.Build.VERSION.SdkInt >= Android.OS.BuildVersionCodes.Lollipop)
{
    // Do things the Lollipop way
}
else
{
    // Do things the pre-Lollipop way
}
```

There's no fast and simple rule that explains how to reduce or modify your app's functionality when it runs on older Android versions that are lacking one or more APIs. In some cases (such as in the `SetCategory` example above), it's sufficient to omit the API call when it's not available. However, in other cases, you may need to implement alternate functionality for when `Android.OS.Build.VERSION.SdkInt` is detected to be less than the API level that your app needs to present its optimum experience.

API levels and libraries

- Visual Studio
- Visual Studio for Mac

When you create a Xamarin.Android library project (such as a class library or a bindings library), you can configure only the Target Framework setting – the Minimum Android version and the Target Android version settings are not available. That is because there is no **Android Manifest** page:



The Minimum Android version and Target Android version settings are not available because the resulting library is not a stand-alone app – the library could be run on any Android version, depending on the app that it is packaged with. You can specify how the library is to be *compiled*, but you can't predict which platform API level the library will be run on. With this in mind, the following best practices should be observed when consuming or creating libraries:

- **When consuming an Android library** – If you are consuming an Android library in your application, be sure to set your app's Target Framework setting to an API level that is *at least as high as* the Target Framework setting of the library.
- **When creating an Android library** – If you are creating an Android library for use by other applications, be sure to set its Target Framework setting to the minimum API level that it needs in order to compile.

These best practices are recommended to help prevent the situation where a library attempts to call an API that is not available at runtime (which can cause the app to crash). If you are a library developer, you should strive to restrict your usage of API calls to a small and well-established subset of the total API surface area. Doing so helps to ensure that your library can be used safely across a wider range of Android versions.

Summary

This guide explained how Android API levels are used to manage app compatibility across different versions of Android. It provided detailed steps for configuring the Xamarin.Android *Target Framework*, *Minimum Android version*, and *Target Android version* project settings. It provided instructions for using the Android SDK Manager to install SDK packages, included examples of how to write code to deal with different API levels at runtime, and explained how to manage API levels when creating or consuming Android libraries. It also provided a comprehensive list that relates API levels to Android version numbers (such as Android 4.4), Android version names (such as Kitkat), and Xamarin.Android build version codes.

Related Links

- Android SDK Setup

- SDK CLI Tooling Changes
- Picking your compileSdkVersion, minSdkVersion, and targetSdkVersion
- What is API Level?
- Codenames, Tags, and Build Numbers

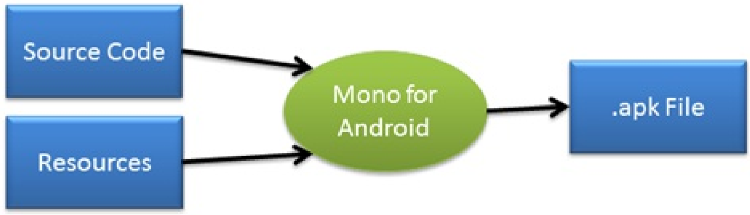
Android Resources

7/8/2021 • 2 minutes to read • Edit Online

This article introduces the concept of Android resources in Xamarin.Android and will document how to use them. It covers how to use resources in your Android application to support application localization, and multiple devices including varying screen sizes and densities.

Overview

An Android application is seldom just source code. There are often many other files that make up an application: video, images, fonts, and audio files just to name a few. Collectively, these non-source code files are referred to as resources and are compiled (along with the source code) during the build process and packaged as an APK for distribution and installation onto devices:

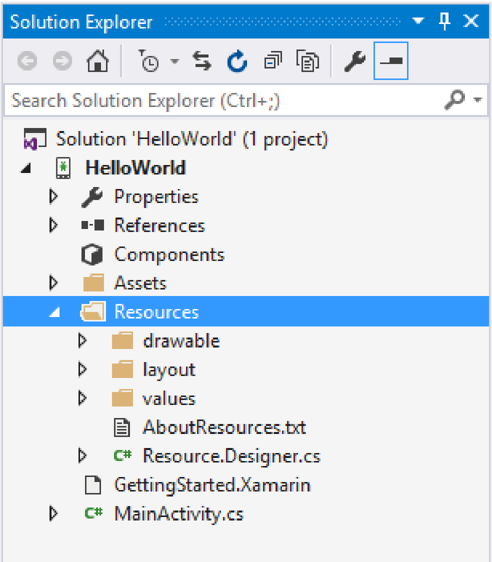


Resources offer several advantages to an Android application:

- **Code-Separation** – Separates source code from images, strings, menus, animations, colors, etc. As such resources can help considerably when localizing.
- **Target multiple devices** – Provides simpler support of different device configurations without code changes.
- **Compile-time Checking** – Resources are static and compiled into the application. This allows the usage of the resources to be checked at compile time, when it will be easy to catch and correct the mistakes, as opposed to run-time when it is more difficult to locate and costly to correct.

When a new Xamarin.Android project is started, a special directory called Resources is created, along with some subdirectories:

- Visual Studio
- Visual Studio for Mac



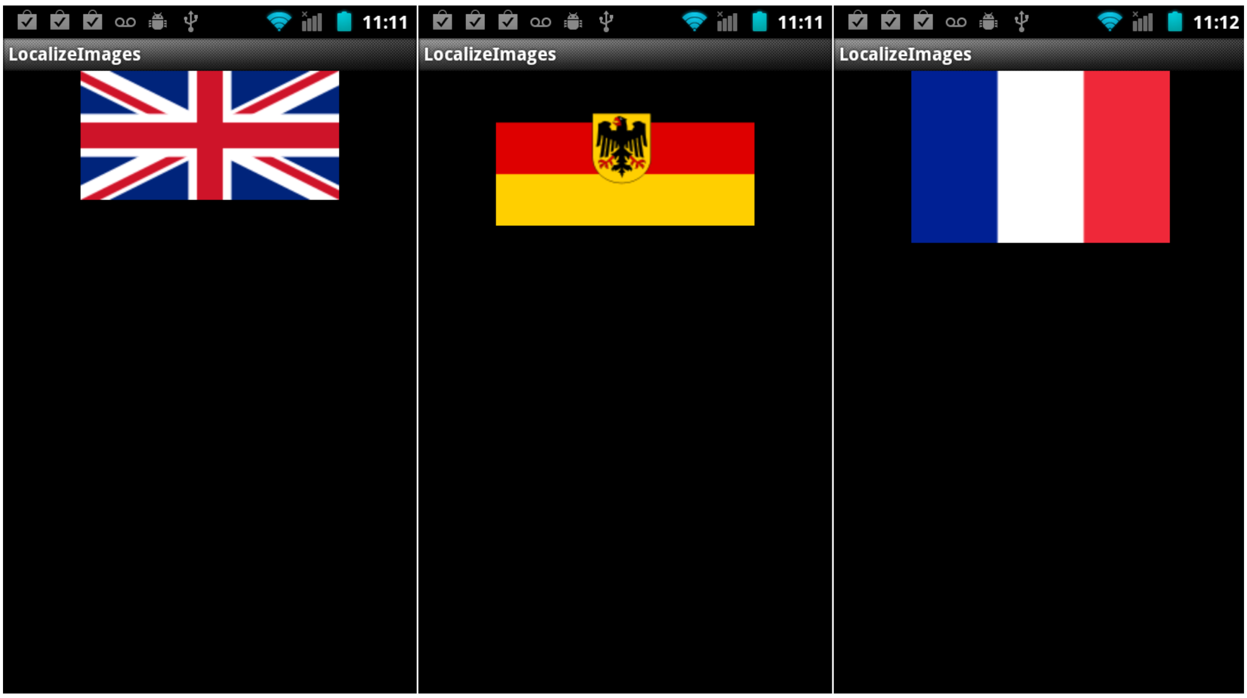
In the image above, the application resources are organized according to their type into these subdirectories: images will go in the **drawable** directory; views go in the **layout** subdirectory, etc.

There are two ways to access these resources in a Xamarin.Android application: *programmatically* in code and *declaratively* in XML using a special XML syntax.

These resources are called *Default Resources* and are used by all devices unless a more specific match is specified. Additionally, every type of resource may optionally have *Alternate Resources* that Android may use to target specific devices. For example, resources may be provided to target the user's locale, the screen size, or if the device is rotated 90 degrees from portrait to landscape, etc. In each of these cases, Android will load the resources for use by the application without any extra coding effort by the developer.

Alternate resources are specified by adding a short string, called a *qualifier*, to the end of the directory holding a given type of resources.

For example, **resources/drawable-de** will specify the images for devices that are set to a German locale, while **resources/drawable-fr** would hold images for devices set to a French locale. An example of providing alternate resources can be seen in the image below where the same application is being run with just the locale of the device changing:



This article will take a comprehensive look at using resources and cover the following topics:

- **Android Resource Basics** – Using default resources programmatically and declaratively, adding resource types such as images and fonts to an application.
- **Device Specific Configurations** – Supporting the different screen resolutions and densities in an application.
- **Localization** – Using resources to support the different regions an application may be used.

Related Links

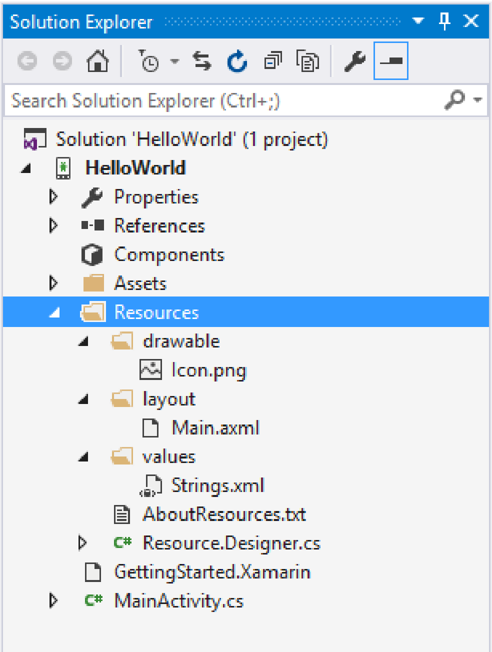
- Using Android Assets
- Application Fundamentals
- Application Resources
- Supporting Multiple Screens

Android Resource Basics

7/8/2021 • 4 minutes to read • Edit Online

Almost all Android applications will have some sort of resources in them; at a minimum they often have the user interface layouts in the form of XML files. When a Xamarin.Android application is first created, default resources are setup by the Xamarin.Android project template:

- Visual Studio
- Visual Studio for Mac



The five files that make up the default resources were created in the Resources folder:

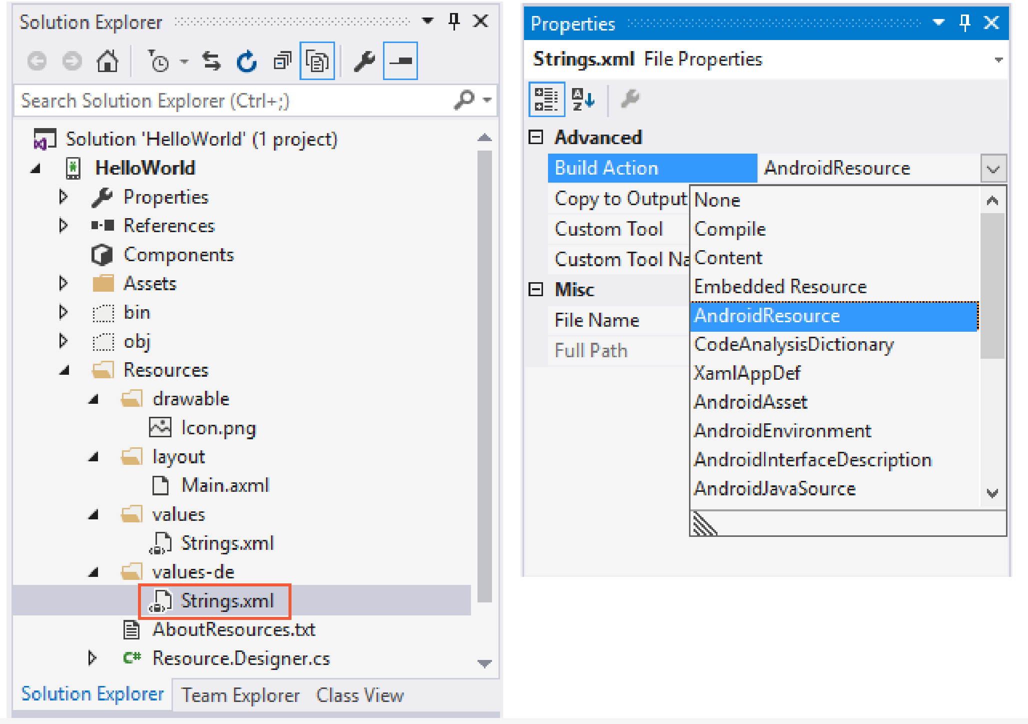
- **Icon.png** – The default icon for the application
- **Main.axml** – The default user interface layout file for an application. Note that while Android uses the **.xml** file extension, Xamarin.Android uses the **.axml** file extension.
- **Strings.xml** – A string table to help with localization of the application
- **AboutResources.txt** – This is not necessary and may safely be deleted. It just provides a high level overview of the Resources folder and the files in it.
- **Resource.designer.cs** – This file is automatically generated and maintained by Xamarin.Android and holds the unique ID's assigned to each resource. This is very similar and identical in purpose to the R.java file that an Android application written in Java would have. It is automatically created by the Xamarin.Android tools and will be regenerated from time to time.

Creating and Accessing Resources

Creating resources is as simple as adding files to the directory for the resource type in question. The screen shot below shows string resources for German locales were added to a project. When **Strings.xml** was added to the file, the **Build Action** was automatically set to **AndroidResource** by the Xamarin.Android tools:

- Visual Studio

- Visual Studio for Mac



This allows the Xamarin.Android tools to properly compile and embed the resources in to the APK file. If for some reason the **Build Action** is not set to **Android Resource**, then the files will be excluded from the APK, and any attempt to load or access the resources will result in a run-time error and the application will crash.

Also, it's important to note that while Android only supports lowercase filenames for resource items, Xamarin.Android is a bit more forgiving; it will support both uppercase and lowercase filenames. The convention for image names is to use lowercase with underscores as separators (for example, **my_image_name.png**). Note that resource names cannot be processed if dashes or spaces are used as separators.

Once resources have been added to a project, there are two ways to use them in an application – programmatically (inside code) or from XML files.

Referencing Resources Programmatically

To access these files programmatically, they are assigned a unique resource ID. This resource ID is an integer defined in a special class called `Resource`, which is found in the file **Resource.designer.cs**, and looks something like this:

```
public partial class Resource
{
    public partial class Attribute
    {
    }
    public partial class Drawable {
        public const int Icon=0x7f020000;
    }
    public partial class Id
    {
        public const int Textview=0x7f050000;
    }
    public partial class Layout
    {
        public const int Main=0x7f030000;
    }
    public partial class String
    {
        public const int App_Name=0x7f040001;
        public const int Hello=0x7f040000;
    }
}
```

Each resource ID is contained inside a nested class that corresponds to the resource type. For example, when the file **Icon.png** was added to the project, Xamarin.Android updated the `Resource` class, creating a nested class called `Drawable` with a constant inside named `Icon`. This allows the file **Icon.png** to be referred to in code as `Resource.Drawable.Icon`. The `Resource` class should not be manually edited, as any changes that are made to it will be overwritten by Xamarin.Android.

When referencing resources programmatically (in code), they can be accessed via the Resources class hierarchy which uses the following syntax:

```
[<PackageName>.]Resource.<ResourceType>.<ResourceName>
```

- **PackageName** – The package which is providing the resource and is only required when resources from other packages are being used.
- **ResourceType** – This is the nested resource type that is within the Resource class described above.
- **Resource Name** – this is the filename of the resource (without the extension) or the value of the android:name attribute for resources that are in an XML element.

Referencing Resources from XML

Resources in an XML file are accessed by a following a special syntax:

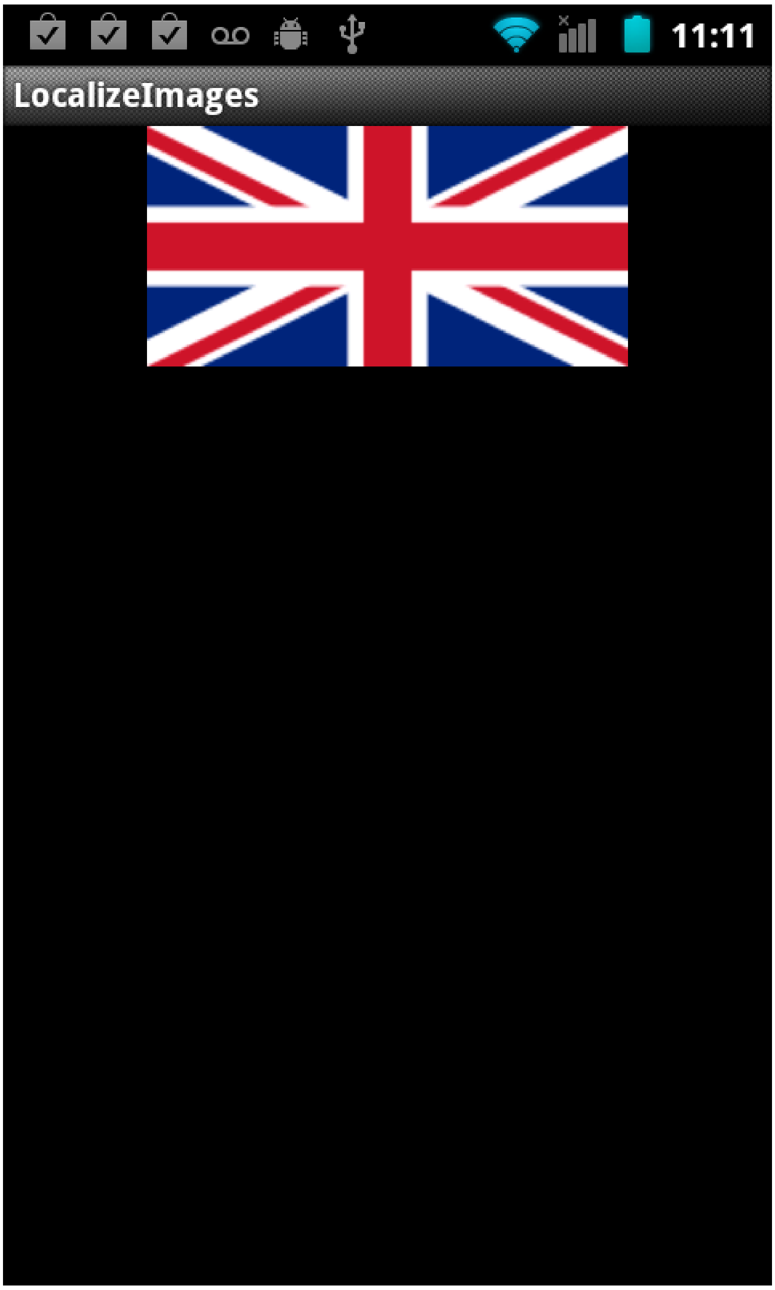
```
@[<PackageName>:]<ResourceType>/<ResourceName>
```

- **PackageName** – the package which is providing the resource and is only required when resources from other packages are being used.
- **ResourceType** – This is the nested resource type that is within the Resource class.
- **Resource Name** – this is the filename of the resource (*without* the file type extension) or the value of the `android:name` attribute for resources that are in an XML element.

For example the contents of a layout file, **Main.axml**, are as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
    <ImageView android:id="@+id/myImage"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/flag" />
</LinearLayout>
```

This example has an `ImageView` that requires a drawable resource named **flag**. The `ImageView` has its `src` attribute set to `@drawable/flag`. When the activity starts, Android will look inside the directory **Resource/Drawable** for a file named **flag.png** (the file extension could be another image format, like **flag.jpg**) and load that file and display it in the `ImageView`. When this application is run, it would look something like the following image:

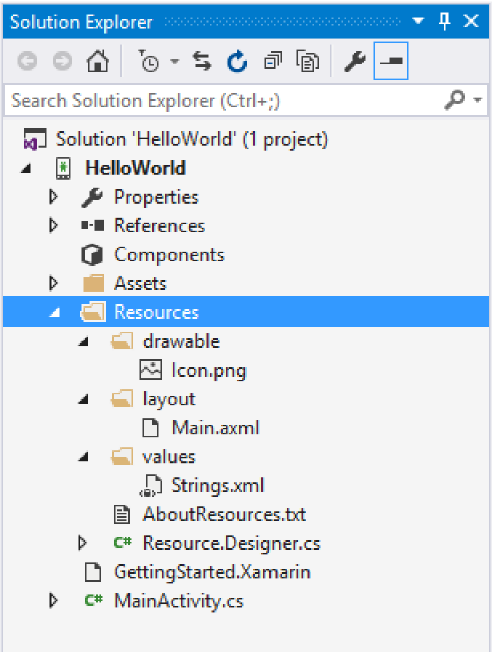


Default Resources

7/8/2021 • 4 minutes to read • Edit Online

Default resources are items that are not specific to any particular device or form factor, and therefore are the default choice by the Android OS if no more specific resources can be found. As such, they're the most common type of resource to create. They're organized into sub-directories of the **Resources** directory according to their resource type:

- Visual Studio
- Visual Studio for Mac



In the image above, the project has default values for drawable resources, layouts, and values (XML files that contain simple values).

A complete list of resource types is provided below:

- **animator** – XML files that describe property animations. Property animations were introduced in API level 11 (Android 3.0) and provides for the animation of properties on an object. Property animations are a more flexible and powerful way to describe animations on any type of object.
- **anim** – XML files that describe *tween* animations. Tween animations are a series of animation instructions to perform transformations on the contents of a View object, or example, rotation an image or growing the size of text. Tween animations are limited to only View objects.
- **color** – XML files that describe a state list of colors. To understand color state lists, consider a UI widget such as a Button. It may have different states such as pressed or disabled, and the button may change color with each change in state. The list is expressed in a state list.
- **drawable** – Drawable resources are a general concept for graphics that can be compiled into the application and then accessed by API calls or referenced by other XML resources. Some examples of drawables are bitmap files (.png, .gif, .jpg), special resizable bitmaps known as Nine-Patches, state lists, generic shapes defined in XML, etc.
- **layout** – XML files that describe a user interface layout, such as an activity or a row in a list.

- **menu** – XML files that describe application menus such as *Options Menus*, *Context Menus*, and *submenus*. For an example of menus, see the Popup Menu Demo or the Standard Controls sample.
- **raw** – Arbitrary files that are saved in their raw, binary form. These files are compiled into an Android application in a binary format.
- **values** – XML files that contain simple values. An XML file in the values directory does not define a single resource, but instead can define multiple resources. For example one XML file may hold a list of string values, while another XML file may hold a list of color values.
- **xml** – XML files that are similar in function to the .NET configuration files. These are arbitrary XML that can be read at run time by the application.

Alternate Resources

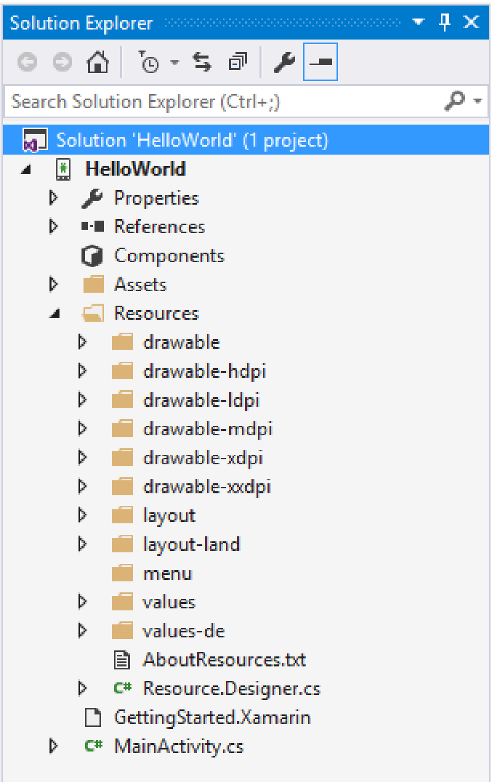
7/8/2021 • 8 minutes to read • Edit Online

Alternate resources are those resources that target a specific device or run-time configuration such as the current language, particular screen size, or pixel density. If Android can match a resource that is more specific for a particular device or configuration than the default resource, then that resource will be used instead. If it does not find an alternate resource that matches the current configuration, then the default resources will be loaded. How Android decides what resources will be used by an application will be covered in more detail below, in the section Resource Location

Alternate resources are organized as a sub-directory inside the Resources folder according to the resource type, just like default resources. The name of the alternate resource subdirectory is in the form:
ResourceType-Qualifier

Qualifier is a name that identifies a specific device configuration. There may be more than one qualifier in a name, each of them separated by a dash. For example, the screenshot below shows a simple project that has alternate resources for various configurations such as locale, screen density, screen size, and orientation:

- Visual Studio
- Visual Studio for Mac



The following rules apply when adding qualifiers to a resource type:

1. There may be more than one qualifier, with each qualifier separated by a dash.

2. The qualifiers maybe specified only once.

3. Qualifiers must be in the order they appear in the table below.

The possible qualifiers are listed below for reference:

- **MCC and MNC** – The mobile country code (MCC) and optionally the mobile network code (MNC). The SIM card will provide the MCC, while the network the device is connected to will provide the MNC. Although it is possible to target locales using the mobile country code, the recommend approach is to use the Language qualifier specified below. For example, to target resources to Germany, the qualifier would be `mcc262`. To target resources for T-Mobile in the U.S., the qualifier is `mcc310-mnc026`. For a complete list of mobile country codes and mobile network codes see http://mcc-mnc.com/.

- **Language** – The two-letter ISO 639-1 language code and optionally followed by the two-letter ISO-3166-alpha-2 region code. If both qualifiers are provided, then they are separated by an `-r`. For example, to target French-speaking locales then the qualifier of `fr` is used. To target French-Canadian locales, the `fr-rCA` would be used. For a complete list of language codes and region codes, see Codes for the Representation of Names Of Languages and Country names and code elements.

- **Smallest Width** – Specifies the smallest screen width the application is meant to execute on. Covered in more detail in Creating Resources for Varying Screens. Available in API level 13 (Android 3.2) and above. For example, the qualifier `sw320dp` is used to target devices whose height and width is at least 320dp.

- **Available Width** – The minimum width of the screen in the format w*N*dp, where *N* is the width in density independent pixels. This value may change as the user rotates the device. Covered in more detail in Creating Resources for Varying Screens. Available in API level 13 (Android 3.2) and above. Example: the qualifier w720dp is used to target devices that have a width of least 720dp.

- **Available Height** – The minimum height of the screen in the format h*N*dp, where *N* is the height in dp. This value may change as the user rotates the device. Covered in more detail in Creating Resources for Varying Screens. Available in API level 13 (Android 3.2) and above. For example, the qualifier h720dp is used to target devices that have a height of least 720dp

- **Screen Size** – This qualifier is a generalization of the screen size that these resources are for. It is covered in more detail in Creating Resources for Varying Screens. Possible values are `small`, `normal`, `large`, and `xlarge`. Added in API level 9 (Android 2.3/Android 2.3.1/Android 2.3.2)

- **Screen Aspect** – This is based on the aspect ratio, not the screen orientation. A long screen is wider. Added in API level 4 (Android 1.6). Possible values are long and notlong.

- **Screen Orientation** – Portrait or landscape screen orientation. This can change during the lifetime of an application. Possible values are `port` and `land`.

- **Dock Mode** – For devices in a car dock or a desk dock. Added in API level 8 (Android 2.2.x). Possible values are `car` and `desk`.

- **Night Mode** – Whether or not the application is running at night or in the day. This may change during the lifetime of an application and is meant to give developers an opportunity to use darker versions of an interface at night. Added in API level 8 (Android 2.2.x). Possible values are `night` and `notnight`.

- **Screen Pixel Density (dpi)** – The number of pixels in a given area on the physical screen. Typically expressed as dots per inch (dpi). Possible values are:

  - `ldpi` – Low density screens.

  - `mdpi` – Medium density screens

  - `hdpi` – High density screens

  - `xhdpi` – Extra high density screens

  - `nodpi` – Resources that are not to be scaled

  - `tvdpi` – Introduced in API level 13 (Android 3.2) for screens between mdpi and hdpi.

- **Touchscreen Type** – Specifies the type of touchscreen a device may have. Possible values are `notouch` (no touch screen), `stylus` (a resistive touchscreen suitable for a stylus), and `finger` (a touchscreen).
- **Keyboard Availability** – Specifies what kind of keyboard is available. This may change during the lifetime of an application – for example when a user opens a hardware keyboard. Possible values are:
    - `keysexposed` – The device has a keyboard available. If there is no software keyboard enabled, then this is only used when the hardware keyboard is opened.
    - `keyshidden` – The device does have a hardware keyboard but it is hidden and no software keyboard is enabled.
    - `keyssoft` – the device has a software keyboard enabled.
- **Primary Text Input Method** – Use to specify what kinds of hardware keys are available for input. Possible values are:
    - `nokeys` – There are no hardware keys for input.
    - `qwerty` – There is a qwerty keyboard available.
    - `12key` – There is a 12-key hardware keyboard
- **Navigation Key Availability** – For when 5-way or d-pad (directional-pad) navigation is available. This can change during the lifetime of your application. Possible values are:
    - `navexposed` – the navigational keys are available to the user
    - `navhidden` – the navigational keys are not available.
- **Primary Non-Touch Navigation Method** – The kind of navigation available on the device. Possible values are:
    - `nonav` – the only navigation facility available is the touch screen
    - `dpad` – a d-pad (directional-pad) is available for navigation
    - `trackball` – the device has a trackball for navigation
    - `wheel` – the uncommon scenario where there are one or more directional wheels available
- **Platform Version (API level)** – The API level supported by the device in the format v*N*, where *N* is the API level that is being targeted. For example, v11 will target an API level 11 (Android 3.0) device.

For more complete information about resource qualifiers see Providing Resources on the Android Developers website.

How Android Determines What Resources to Use

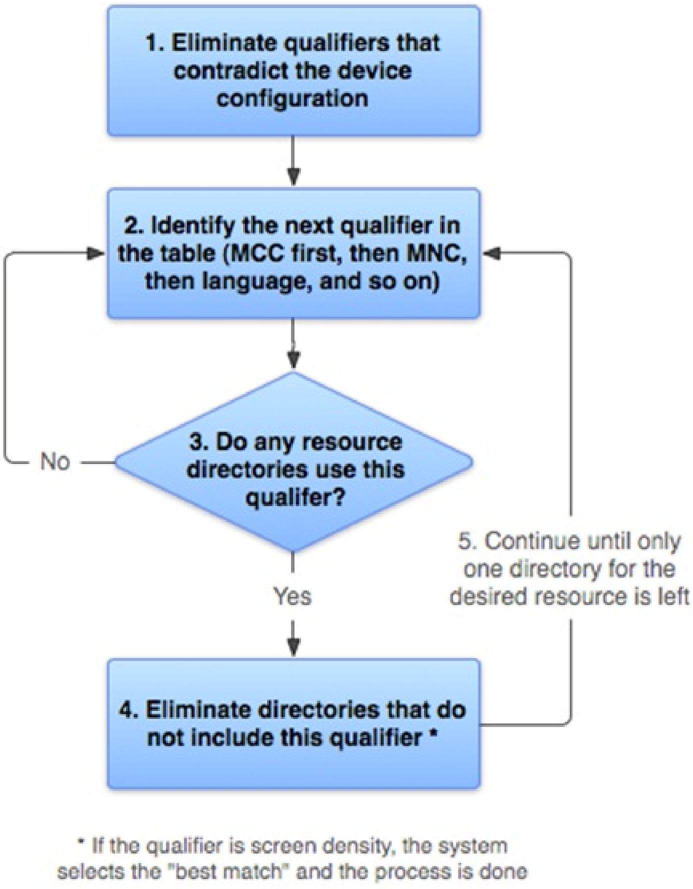
It is very possible and likely that an Android application will contain many resources. It is important to understand how Android will select the resources for an application when it runs on a device.

Android determines the resources base by iterating over the following test of rules:

- **Eliminate contradictory qualifiers** – for example, if the device orientation is portrait, then all landscape resource directories will be rejected.
- **Ignore qualifiers not supported** – Not all qualifiers are available to all API levels. If a resource directory contains a qualifier that is not supported by the device, then that resource directory will be ignored.

- **Identify the next highest priority qualifier** – referring to the table above select the next highest priority qualifier (from top to bottom).
- **Keep any resource directories for qualifier** – if there are any resource directories that match the qualifier to the table above select the next highest priority qualifier (from top to bottom).

These rules are also illustrated in the following flowchart:



When the system is looking for density-specific resources and cannot find them, it will attempt to locate other density specific resources and scale them. Android may not necessarily use the default resources. For example, when looking for a low-density resource and it is not available, Android may select high-density version of the resource over the default or medium-density resources. It does this because the high-density resource can be scaled down by a factor of 0.5, which will result in fewer visibility issues than scaling down a medium-density resource which would require a factor of 0.75.

As an example, consider an application that has the following drawable resource directories:

```
drawable
drawable-en
drawable-fr-rCA
drawable-en-port
drawable-en-notouch-12key
drawable-en-port-ldpi
drawable-port-ldpi
drawable-port-notouch-12key
```

And now the application is run on a device with the following configuration:

- **Locale** – en-GB
- **Orientation** – port
- **Screen density** – hdpi

- **Touchscreen type** – notouch
- **Primary input method** – 12key

To begin with, the French resources are eliminated as they conflict with the locale of `en-GB`, leaving us with:

```
drawable
drawable-en
drawable-en-port
drawable-en-notouch-12key
drawable-en-port-ldpi
drawable-port-ldpi
drawable-port-notouch-12key
```

Next, the first qualifier is selected from the qualifiers table above: MCC and MNC. There are no resource directories that contain this qualifier so the MCC/MNC code is ignored.

The next qualifier is selected, which is Language. There are resources that match the language code. All resource directories that do not match the language code of `en` are rejected, so that the list of resources is now:

```
drawable-en-port
drawable-en-notouch-12key
drawable-en-port-ldpi
```

The next qualifier that is present is for screen orientation, so all resource directories that do not match the screen orientation of `port` are eliminated:

```
drawable-en-port
drawable-en-port-ldpi
```

Next is the qualifier for screen density, `ldpi`, which results in the exclusion of one more resource directory:

```
drawable-en-port-ldpi
```

As a result of this process, Android will use the drawable resources in the resource directory `drawable-en-port-ldpi` for the device.

> **NOTE**
> The screen size qualifiers provide one exception to this selection process. It is possible for Android to select resources that are designed for a smaller screen than what the current device provides. For example, a large screen device may use the resources provide for a normal sized screen. However the reverse of this is not true: the same large screen device will not use the resources provided for an xlarge screen. If Android cannot find a resource set that matches a given screen size, the application will crash.

Creating resources for varying screens

7/8/2021 • 7 minutes to read • Edit Online

Android itself runs on many different devices, each having a wide variety of resolutions, screen sizes, and screen densities. Android will perform scaling and resizing to make your application work on these devices, but this may result in a sub-optimal user experience. For example, images could appear blurry, or they may be positioned as expected on a view.

Concepts

A few terms and concepts are important to understand to support multiple screens.

- **Screen Size** – The amount of physical space for displaying your application
- **Screen Density** – The number of pixels in any given area on the screen. The typical unit of measure is dots per inch (dpi).
- **Resolution** – The total number of pixels on the screen. When developing applications, resolution is not as important as screen size and density.
- **Density-independent pixel (dp)** – A virtual unit of measure to allow layouts to be designed independent of density. This formula is used to convert dp into screen pixels:

  $px = dp \times dpi \div 160$

- **Orientation** – The screen's orientation is considered to be landscape when it is wider than it is tall. In contrast, portrait orientation is when the screen is taller than it is wide. The orientation can change during the lifetime of an application as the user rotates the device.

Notice that the first three of these concepts are inter-related – increasing the resolution without increasing the density will increase the screen size. However if both the density and resolution are increased, then the screen size can remain unchanged. This relationship between screen size, density, and resolution complicate screen support quickly.

To help deal with this complexity, the Android framework prefers to use *density-independent pixels (dp)* for screen layouts. By using density independent pixels, UI elements will appear to the user to have the same physical size on screens with different densities.

Supporting various screen sizes and densities

Android handles most of the work to render the layouts properly for each screen configuration. However, there are some actions that can be taken to help the system out.

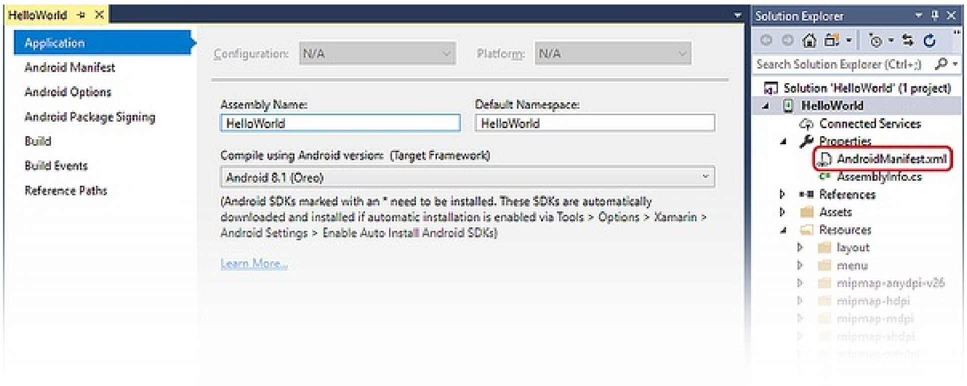
The use of density-independent pixels instead of actual pixels in layouts is sufficient in most cases to ensure density independence. Android will scale the drawables at runtime to the appropriate size. However, it is possible that scaling will cause bitmaps to appear blurry. To work around this problem, supply alternate resources for the different densities. When designing devices for multiple resolutions and screen densities, it will prove easier to start with the higher resolution or density images and then scale down.

Declare the supported screen size

Declaring the screen size ensures that only supported devices can download the application. This is accomplished by setting the supports-screens element in the **AndroidManifest.xml** file. This element is used to specify what screen sizes are supported by the application. A given screen is considered to be supported if the application can properly place its layouts to fill screen. By using this manifest element, the application will not show up in *Google Play* for devices that do not meet the screen specifications. However, the application will still run on devices with unsupported screens, but the layouts may appear blurry and pixelated.

Supported screen sixes are declared in the **Properites/AndroidManifest.xml** file of the solution:

- Visual Studio
- Visual Studio for Mac



Edit **AndroidManifest.xml** to include supports-screens:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          android:versionCode="1"
          android:versionName="1.0"
          package="HelloWorld.HelloWorld">
      <uses-sdk android:minSdkVersion="21" android:targetSdkVersion="27" />
      <supports-screens android:resizable="true"
                        android:smallScreens="true"
                        android:normalScreens="true"
                        android:largeScreens="true" />
      <application android:allowBackup="true"
                   android:icon="@mipmap/ic_launcher"
                   android:label="@string/app_name"
                   android:roundIcon="@mipmap/ic_launcher_round"
                   android:supportsRtl="true" android:theme="@style/AppTheme">
  </application>
</manifest>
```

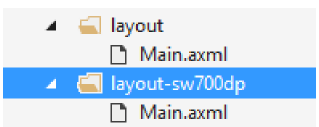
Provide alternate layouts for different screen sizes

Alternate layouts make it possible to customize a view for a specifc screen size, changing the positioning or size of the component UI elements.

Starting with API Level 13 (Android 3.2), the screen sizes are deprecated in favor of using the sw*N*dp qualifier. This new qualifier declares the amount of space a given layout needs. It is recommended that applications that are meant to run on Android 3.2 or higher should be using these newer qualifiers.

For example, if a layout required a minimum 700 dp of screen width, the alternate layout would go in a folder **layout-sw700dp**:

- Visual Studio
- Visual Studio for Mac

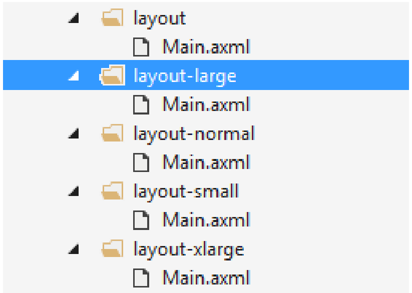


As a guideline, here are some numbers for various devices:

- **Typical phone** – 320 dp: a typical phone
- **A 5" tablet / "tweener" device** – 480 dp: such as the Samsung Note
- **A 7" tablet** – 600 dp: such as the Barnes & Noble Nook
- **A 10" tablet** – 720 dp: such as the Motorola Xoom

For applications that target API levels up to 12 (Android 3.1), the layouts should go in directories that use the qualifiers **small/normal/large/xlarge** as generalizations of the various screen sizes that are available in most devices. For example, in the image below, there are alternate resources for the four different screen sizes:

- Visual Studio
- Visual Studio for Mac

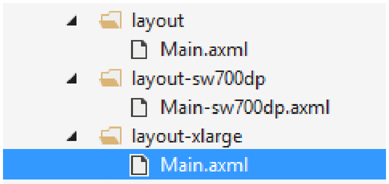


The following is a comparison of how the older pre-API Level 13 screen size qualifiers compare to density-independent pixels:

- 426 dp x 320 dp is **small**
- 470 dp x 320 dp is **normal**
- 640 dp x 480 dp is **large**
- 960 dp x 720 dp is **xlarge**

The newer screen size qualifiers in API level 13 and up have a higher precedence than the older screen qualifiers of API levels 12 and lower. For applications that will span the old and the new API levels, it may be necessary to create alternate resources using both sets of qualifiers as shown in the following screenshot:

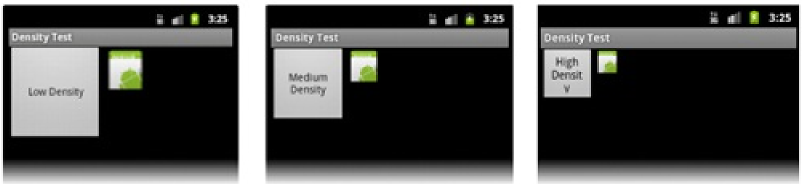
- Visual Studio
- Visual Studio for Mac



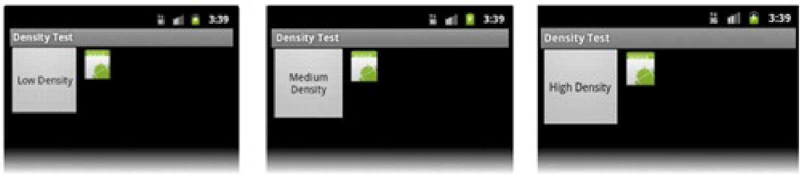
Provide different bitmaps for different screen densities

Although Android will scale bitmaps as necessary for a device, the bitmaps themselves may not elegantly scale up or down: they may become fuzzy or blurry. Providing bitmaps appropriate for the screen density will mitigate this problem.

For example, the image below is an example of layout and appearance problems that may occur when density-specify resources are not provided.

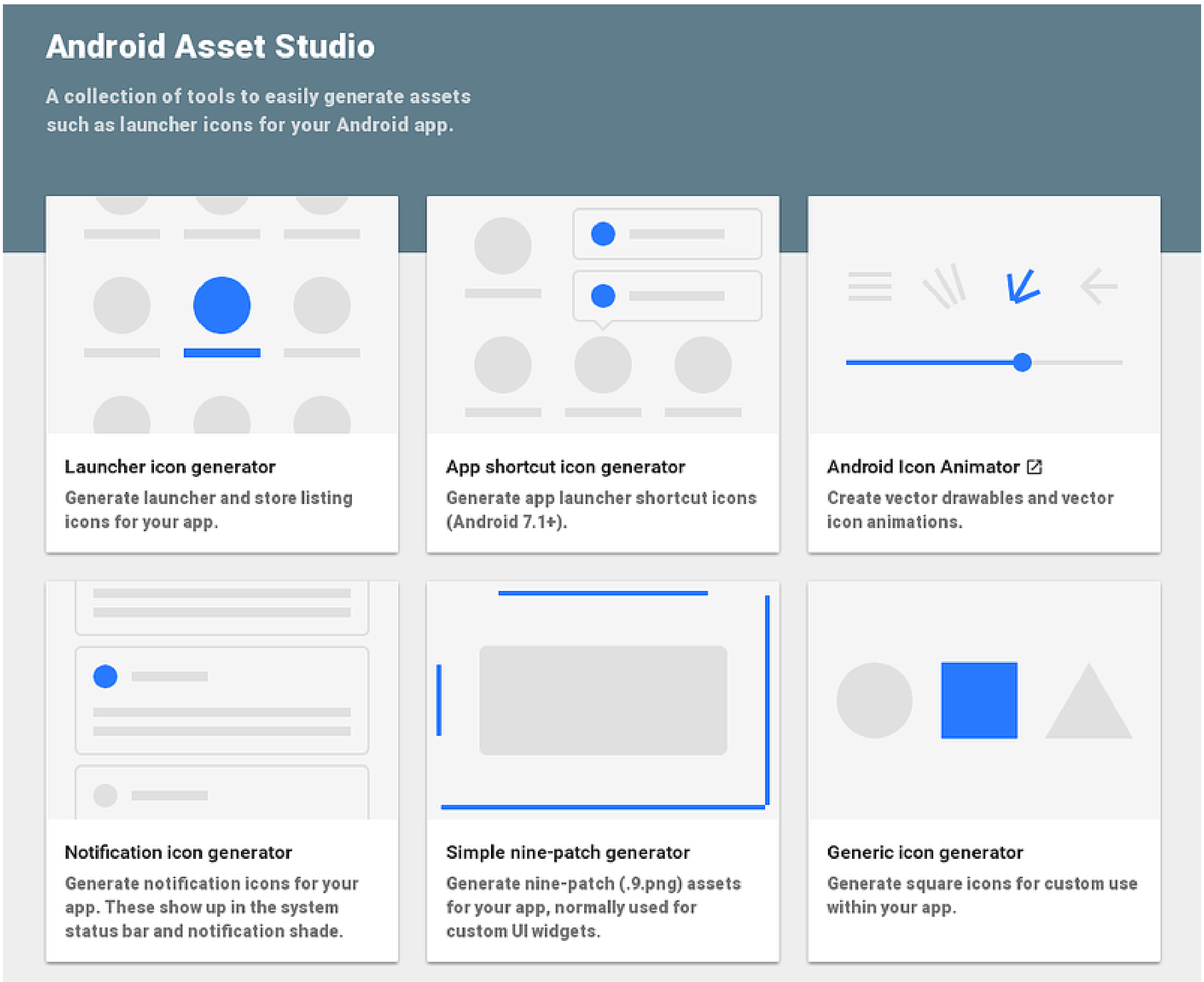


Compare this to a layout that is designed with density-specific resources:



Create varying density resources with Android Asset Studio

The creation of these bitmaps of various densities can be a bit tedious. As such, Google has created an online utility that can reduce some of the tedium involved with the creation of these bitmaps called the Android Asset Studio.



This website will help with creation of bitmaps that target the four common screen densities by providing one image. Android Asset Studio will then create the bitmaps with some customizations and then allow them to be downloaded as a zip file.

Tips for multiple screens

Android runs on a bewildering number of devices, and the combination of screen sizes and screen densities can seem overwhelming. The following tips can help minimize the effort necessary to support various devices:

- **Only design and develop for what you need** – There are many different devices out there, but some exist in rare form factors that may take significant effort to design and develop for. The **Screen Size and Density** dashboard is a page provided by Google that provides data on breakdown of the screen size/screen density matrix. This breakdown provides insight on how to development effort on supporting screens.

- **Use DPs rather than Pixels** - Pixels become troublesome as screen density changes. Do not hardcode pixel values. Avoid pixels in favor of dp (density-independent pixels).

- **Avoid** AbsoluteLayout **Wherever Possible** – it is deprecated in API level 3 (Android 1.5) and will result in brittle layouts. It should not be used. Instead, try to use more flexible layout widgets such as **LinearLayout**, **RelativeLayout**, or the new **GridLayout**.

- **Pick one layout orientation as your default** – For example, instead of providing the alternate resources **layout-land** and **layout-port**, put the resources for landscape in **layout**, and the resources for portrait into **layout-port**.

- **Use LayoutParams for Height and Width** - When defining UI elements in an XML layout file, an Android application using the **wrap_content** and **fill_parent** values will have more success ensure a proper look across different devices than using pixel or density-independent units. These dimension values cause Android to scale bitmap resources as appropriate. For this same reason, density-independent units are best reserved for when specifying the margins and padding of UI elements.

Testing multiple screens

An Android application must be tested against all configurations that will be supported. Ideally devices should be tested on the actual devices themselves but in many cases this is not possible or practical. In this case, the use of the emulator and Android Virtual Devices setup for each device configuration will be useful.

The Android SDK provides some emulator skins may be used to create AVDs will replicate the size, density, and resolution of many devices. Many of the hardware vendors likewise provide skins for their devices.

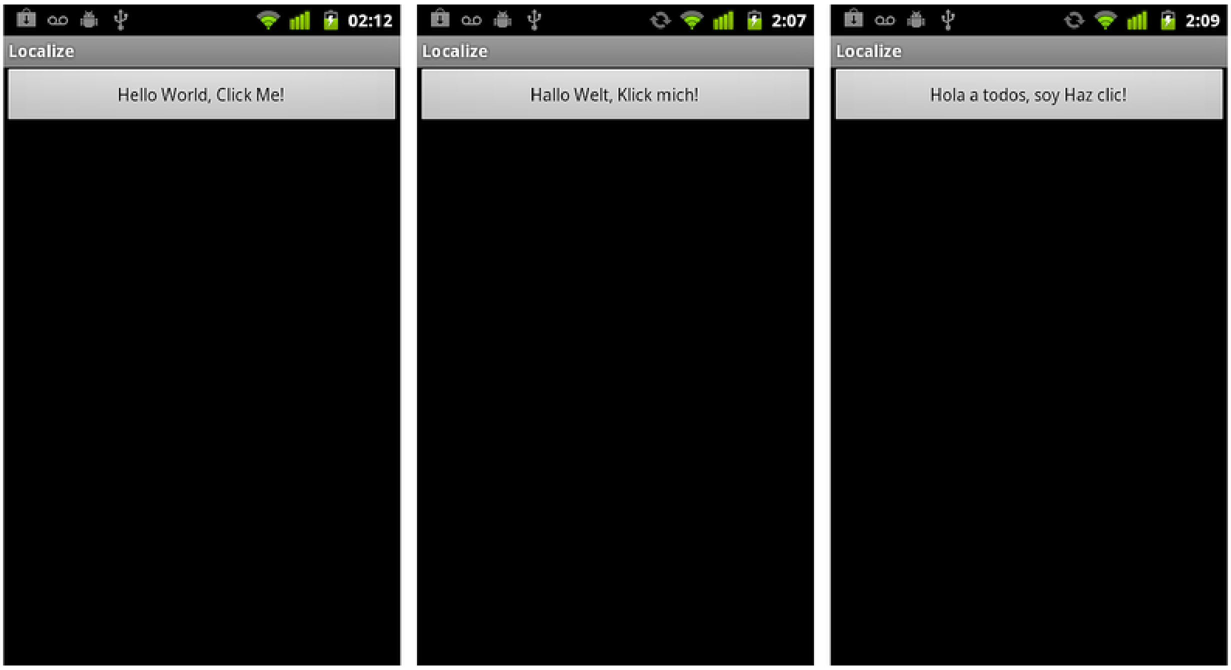
Another option is to use the services of a third party testing service. These services will take an APK, run it on many different devices, and then provide feedback how the application worked.

Application Localization and String Resources

7/8/2021 • 2 minutes to read • Edit Online

Application localization is the act of providing alternate resources to target a specific region or locale. For example, you might provide localized language strings for various countries, or you might change colors or layout to match particular cultures. Android will load and use the resources appropriate for the device's locale at runtime time without any changes to the source code.

For example, the image below shows the same application running in three different device locales, but the text displayed in each button is specific to the locale that each device is set to:

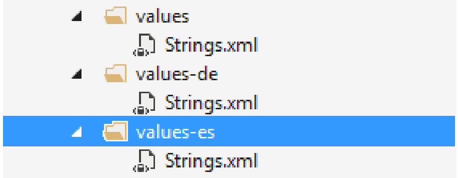


In this example, the contents of a layout file, **Main.axml** looks something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   >
<Button
   android:id="@+id/myButton"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
android:text="@string/hello"
   />
</LinearLayout>
```

In the example above, the string for the button was loaded from the resources by providing the resource ID for the string:

- Visual Studio
- Visual Studio for Mac



Localizing Android Apps

Read the Introduction to Localization for tips and guidance on localizing mobile apps.

The Localizing Android Apps guide contains more specific examples on how to translate strings and localize images using Xamarin.Android.

Related Links

- Localizing Android Apps
- Cross-Platform Localization Overview

Using Android Assets

7/8/2021 • 2 minutes to read • Edit Online

Assets provide a way to include arbitrary files like text, xml, fonts, music, and video in your application. If you try to include these files as "resources", Android will process them into its resource system and you will not be able to get the raw data. If you want to access data untouched, Assets are one way to do it.

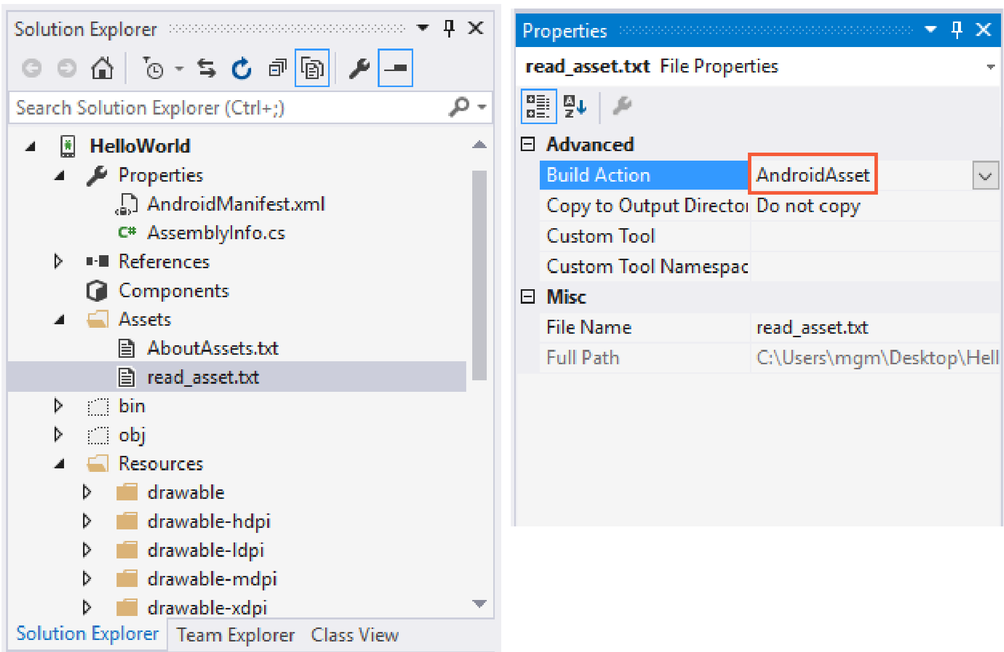
Assets added to your project will show up just like a file system that can read from by your application using AssetManager. In this simple demo, we are going to add a text file asset to our project, read it using `AssetManager`, and display it in a TextView.

Add Asset to Project

Assets go in the `Assets` folder of your project. Add a new text file to this folder called `read_asset.txt`. Place some text in it like "I came from an asset!".

- Visual Studio
- Visual Studio for Mac

Visual Studio should have set the **Build Action** for this file to **AndroidAsset**:



Selecting the correct **BuildAction** ensures that the file will be packaged into the APK at compile time.

Reading Assets

Assets are read using an AssetManager. An instance of the `AssetManager` is available by accessing the Assets property on an `Android.Content.Context`, such as an Activity. In the following code, we open our **read_asset.txt** asset, read the contents, and display it using a TextView.

```
protected override void OnCreate (Bundle bundle)
{
    base.OnCreate (bundle);

    // Create a new TextView and set it as our view
    TextView tv = new TextView (this);

    // Read the contents of our asset
    string content;
    AssetManager assets = this.Assets;
    using (StreamReader sr = new StreamReader (assets.Open ("read_asset.txt")))
    {
        content = sr.ReadToEnd ();
    }

    // Set TextView.Text to our asset content
    tv.Text = content;
    SetContentView (tv);
}
```

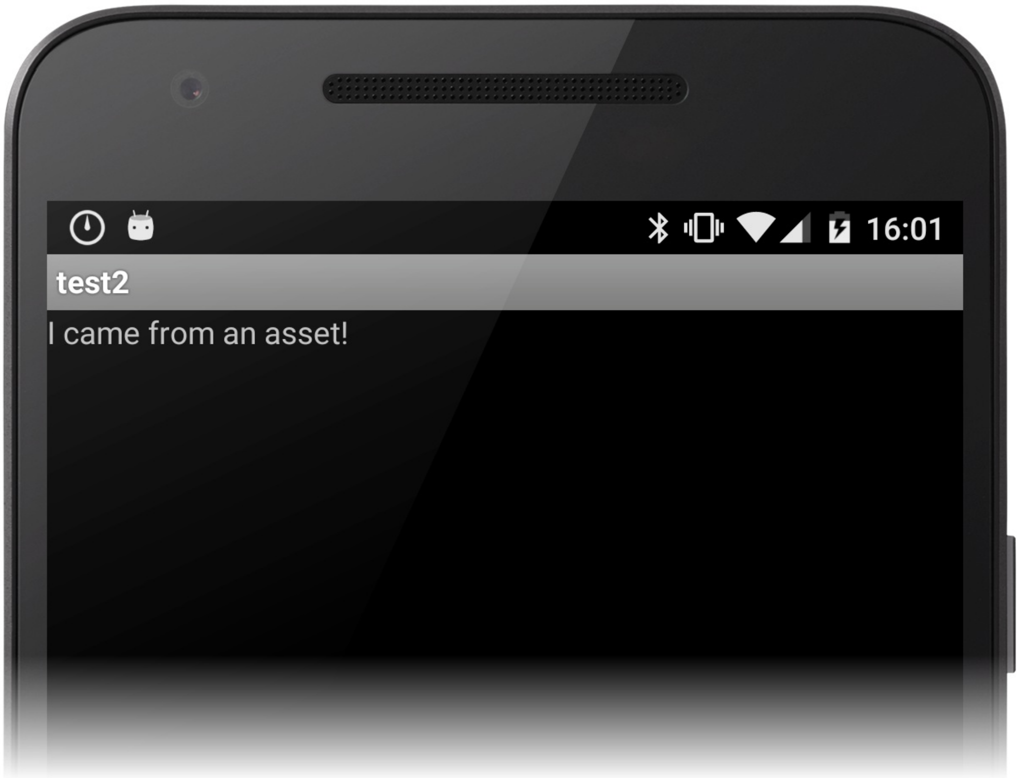
Reading Binary Assets

The use of `StreamReader` in the above example is ideal for text assets. For binary assets, use the following code:

```
protected override void OnCreate (Bundle bundle)
{
    base.OnCreate (bundle);

    // Read the contents of our asset
    const int maxReadSize = 256 * 1024;
    byte[] content;
    AssetManager assets = this.Assets;
    using (BinaryReader br = new BinaryReader (assets.Open ("mydatabase.db")))
    {
        content = br.ReadBytes (maxReadSize);
    }

    // Do something with it...

}
```

Running the Application

Run the application and you should see the following:



Related Links

- AssetManager
- Context

Fonts

11/2/2020 • 11 minutes to read • Edit Online

Overview

Beginning with API level 26, the Android SDK allows fonts to be treated as resources, just like a layouts or drawables. The Android Support Library 26 NuGet will backport the new font API's to those apps that target API level 14 or higher.

After targeting API 26 or installing the Android Support Library v26, there are two ways to use fonts in an Android application:

1. **Package the font as an Android resource** – this ensures that the font is always available to the application, but will increase the size of the APK.
2. **Download the fonts** – Android also supports downloading a font from a *font provider*. The font provider checks if the font is already on the device. If necessary, the font will be downloaded and cached on the device. This font can be shared between multiple applications.

Similar fonts (or a font that may have several different styles) may be grouped into *font families*. This allows developers to specify certain attributes of the font, such as it's weight, and Android will automatically select the appropriate font from the font family.

The Android Support Library v26 will backport support for fonts to API level 26. When targeting the older API levels, it is necessary to declare the `app` XML namespace and to name the various font attributes using the `android:` namespace and the `app:` namespace. If only the `android:` namespace is used, then the fonts will not be displayed devices running API level 25 or less. For example, this XML snippet declares a new *font family* resource that will work in API level 14 and higher:

```
<?xml version="1.0" encoding="utf-8"?>
<font-family
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

     <font  android:font="@font/sourcesanspro_regular"
            android:fontStyle="normal"
            android:fontWeight="400"
            app:font="@font/sourcesanspro_regular"
            app:fontStyle="normal"
            app:fontWeight="400" />

</font-family>
```

As long as fonts are provided to an Android application in a proper way, they can be applied to a UI widget by setting the `fontFamily` attribute. For example, the following snippet demonstrates how to display a font in a TextView:

```
<TextView
    android:text="The quick brown fox jumped over the lazy dog."
    android:fontFamily="@font/sourcesanspro_regular"
    app:fontFamily="@font/sourcesanspro_regular"
    android:textAppearance="?android:attr/textAppearanceLarge"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

This guide will first discuss how to use fonts as an Android resource, and then move on to discuss how to download fonts at runtime.

Fonts as a Resource

Packaging a font into an Android APK ensures that it is always available to the application. A font file (either a .TTF or a .OTF file) is added to a Xamarin.Android application just like any other resource, by copying files to a subdirectory in the **Resources** folder of a Xamarin.Android project. Fonts resources are kept in a **font** sub-directory of the **Resources** folder of the project.

> **NOTE**
>
> The fonts should have a **Build Action** of **AndroidResource** or they will not be packaged into the final APK. The build action should be automatically set by the IDE.

When there are many similar font files (for example, the same font with different weights or styles) it is possible to group them into a font family.

Font Families

A font family is a set of fonts that have different weights and styles. For example, there might be separate font files for bold or italic fonts. The font family is defined by `font` elements in an XML file that is kept in the **Resources/font** directory. Each font family should have it's own XML file.

To create a font family, first add all the fonts to the **Resources/font** folder. Then create a new XML file in the font folder for the font family. The name of the XML file has no affinity or relationship to the fonts being referenced; the resource file can be any legal Android resource file name. This XML file will have a root `font-family` element that contains one or more `font` elements. Each `font` element declares the attributes of a font.

The following XML is an example of a font family for the *Sources Sans Pro* font that defines many different font weights. This is saved as file in the **Resources/font** folder named **sourcesanspro.xml**:

```
<?xml version="1.0" encoding="utf-8"?>
<font-family xmlns:android="http://schemas.android.com/apk/res/android"
             xmlns:app="http://schemas.android.com/apk/res-auto">
    <font android:font="@font/sourcesanspro_regular"
          android:fontStyle="normal"
          android:fontWeight="400"
          app:font="@font/sourcesanspro_regular"
          app:fontStyle="normal"
          app:fontWeight="400" />
    <font android:font="@font/sourcesanspro_bold"
          android:fontStyle="normal"
          android:fontWeight="800"
          app:font="@font/sourcesanspro_bold"
          app:fontStyle="normal"
          app:fontWeight="800" />
    <font android:font="@font/sourcesanspro_italic"
          android:fontStyle="italic"
          android:fontWeight="400"
          app:font="@font/sourcesanspro_italic"
          app:fontStyle="italic"
          app:fontWeight="400" />
</font-family>
```

The `fontStyle` attribute has two possible values:

- **normal** – a normal font

- **italic** – an italic font

The `fontWeight` attribute corresponds to the CSS `font-weight` attribute and refers to the thickness of the font. This is a value in the range of 100 - 900. The following list describes the common font weight values and their name:

- **Thin** – 100
- **Extra Light** – 200
- **Light** – 300
- **Normal** – 400
- **Medium** – 500
- **Semi Bold** – 600
- **Bold** – 700
- **Extra Bold** – 800
- **Black** – 900

Once a font family has been defined, it can be used declaratively by setting the `fontFamily`, `textStyle`, and `fontWeight` attributes in the layout file. For example the following XML snippet sets a 600 weight font (normal) and an italic text style:

```
<TextView
    android:text="Sans Source Pro semi-bold italic, 600 weight, italic"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:fontFamily="@font/sourcesanspro"
    android:textAppearance="?android:attr/textAppearanceLarge"
    android:gravity="center_horizontal"
    android:fontWeight="600"
    android:textStyle="italic"
    />
```

Programmatically Assigning Fonts

Fonts can be programmatically set by using the `Resources.GetFont` method to retrieve a `Typeface` object. Many views have a `TypeFace` property that can be used to assign the font to the widget. This code snippet shows how to programmatically set the font on a TextView:

```
Android.Graphics.Typeface typeface = this.Resources.GetFont(Resource.Font.caveat_regular);
textView1.Typeface = typeface;
textView1.Text = "Changed the font";
```

The `GetFont` method will automatically load the first font within a font family. To load a font that matches a specific style, use the `Typeface.Create` method. This method will try to load a font that matches the specified style. As an example, this snippet will try to load a bold `Typeface` object from a font family that is defined in **Resources/fonts**:

```
var typeface = Typeface.Create("<FONT FAMILY NAME>", Android.Graphics.TypefaceStyle.Bold);
textView1.Typeface = typeface;
```

Downloading Fonts

Instead of packaging fonts as an application resource, Android can download fonts from a remote source. This will have the desirable effect of reducing the size of the APK.

Fonts are downloaded with the assistance of a *font provider*. This is a specialized content provider that manages the downloading and caching of fonts to all applications on the device. Android 8.0 includes a font provider to download fonts from the Google Font Repository. This default font provider is backported to API level 14 with the Android Support Library v26.

When an app makes a request for a font, the font provider will first check to see if the font is already on the device. If not, it will then attempt to download the font. If the font cannot be downloaded, then Android will use the default system font. Once the font has been downloaded, it is available to all applications on the device, not just the app that made the initial request.

When a request is made to download a font, the app does not directly query the font provider. Instead, apps will use an instance of the `FontsContract` API (or the `FontsContractCompat` if the Support Library 26 is being used).

Android 8.0 supports downloading fonts in two different ways:

1. **Declare Downloadable Fonts as a Resource** – An app may declare downloadable fonts to Android via XML resource files. These files will contain all of the meta-data that Android needs to asynchronously download the fonts when the app starts and cache them on the device.
2. **Programmatically** – APIs in Android API level 26 allow an application to download the fonts programmatically, while the application is running. Apps will create a `FontRequest` object for a given font, and pass this object to the `FontsContract` class. The `FontsContract` takes the `FontRequest` and retrieves the font from a *font provider*. Android will synchronously download the font. An example of creating a `FontRequest` will be shown later in this guide.

Regardless of which approach is used, resources files that must be added to the Xamarin.Android application before fonts can be downloaded. First, the font(s) must be declared in an XML file in the **Resources/font** directory as part of a font family. This snippet is an example of how to download fonts from the Google Fonts Open Source collection using the default font provider that comes with Android 8.0 (or Support Library v26):

```
<?xml version="1.0" encoding="utf-8"?>
<font-family xmlns:android="http://schemas.android.com/apk/res/android"
             xmlns:app="http://schemas.android.com/apk/res-auto"
             android:fontProviderAuthority="com.google.android.gms.fonts"
             android:fontProviderPackage="com.google.android.gms"
             android:fontProviderQuery="VT323"
             android:fontProviderCerts="@array/com_google_android_gms_fonts_certs"
             app:fontProviderAuthority="com.google.android.gms.fonts"
             app:fontProviderPackage="com.google.android.gms"
             app:fontProviderQuery="VT323"
             app:fontProviderCerts="@array/com_google_android_gms_fonts_certs"
>
</font-family>
```

The `font-family` element contains the following attributes, declaring the information that Android requires to download the fonts:

1. **fontProviderAuthority** – The authority of the Font Provider to be used for the request.
2. **fontPackage** – The package for the Font Provider to be used for the request. This is used to verify the identity of the provider.
3. **fontQuery** – This is a string that will help the font provider locate the requested font. Details on the font query are specific to the font provider. The `QueryBuilder` class in the Downloadable Fonts sample app provides some information on the query format for fonts from the Google Fonts Open Source Collection.
4. **fontProviderCerts** – A resource array with the list of sets of hashes for the certificates that the provider should be signed with.

Once the fonts are defined, it may be necessary to provide information about the *font certificates* involved with the download.

Font Certificates

If the font provider is not preinstalled on the device, or if the app is using the `Xamarin.Android.Support.Compat` library, Android requires the security certificates of the font provider. These certificates will be listed in an array resource file that is kept in **Resources/values** directory.

For example, the following XML is named **Resources/values/fonts_cert.xml** and stores the certificates for the Google font provider:

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <array name="com_google_android_gms_fonts_certs">
        <item>@array/com_google_android_gms_fonts_certs_dev</item>
        <item>@array/com_google_android_gms_fonts_certs_prod</item>
    </array>
    <string-array name="com_google_android_gms_fonts_certs_dev">
        <item>

MIIEqDCCA5CgAwIBAgIJANWFuGx90071MA0GCSqGSIb3DQEBBAUAMIGUMQswCQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWZvcm5pYTEWMBQG
A1UEBxMNTW91bnRhaW4gVmlldzEQMA4GA1UEChMHQW5kcm9pZDEQMA4GA1UECxMHQW5kcm9pZDEQMA4GA1UEAxMHQW5kcm9pZDEiMCAGCSqG
SIb3DQEJARYTYW5kcm9pZEBhbmRyb2lkLmNvbTAeFw0wODA0MTUyMzM2NTZaFw0zNTA5MDEyMzM2NTZaMIGUMQswCQYDVQQGEwJVUzETMBEG
A1UECBMKQ2FsaWZvcm5pYTEWMBQGA1UEBxMNTW91bnRhaW4gVmlldzEQMA4GA1UEChMHQW5kcm9pZDEQMA4GA1UECxMHQW5kcm9pZDEQMA4G
A1UEAxMHQW5kcm9pZDEiMCAGCSqGSIb3DQEJARYTYW5kcm9pZEBhbmRyb2lkLmNvbTCCASAwDQYJKoZIhvcNAQEBBQADggENADCCAQgCggEB
ANbOLggKv+IxTdGNs8/TGFy0PTP6DHThvbbR24kT9ixcOd9W+EaBPWW+wPPKQmsHxajtWjmQwWfna8mZuSeJS48LIgAZlKkpFeVyxW0qMBuj
b8X8ETrWy550NaFtI6t9+u7hZeTfHwqNvacKhp1RbE6dBRGWynwMVX8XW8N1+UjFaq6GCJukT4qmpN2afb8sCjUigq0GuMwYXrFVee74bQgL
HWGJwPmvmLHC69EH6kWr22ijx4OKXlSIx2xT1AsSHee70w5iDBiK4aph27yH3TxkXy9V89TDdexAcKk/cVHYNnDBapcavl7y0RiQ4biu8ymM
8Ga/nmzhRKya6G0cGw8CAQOjgfwwgfkwHQYDVR0OBBYEFI0cxb6VTEM8YYY6FbBMvAPyT+CyMIHJBgNVHSMEgcEwgb6AFI0cxb6VTEM8YYY6
FbBMvAPyT+CyoYGapIGXMIGUMQswCQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWZvcm5pYTEWMBQGA1UEBxMNTW91bnRhaW4gVmlldzEQMA4G
A1UEChMHQW5kcm9pZDEQMA4GA1UECxMHQW5kcm9pZDEQMA4GA1UEAxMHQW5kcm9pZDEiMCAGCSqGSIb3DQEJARYTYW5kcm9pZEBhbmRyb2lk
LmNvbYIJANWFuGx90071MAwGA1UdEwQFMAMBAf8wDQYJKoZIhvcNAQEEBQADggEBABnTDPEF+3iSP0wNfdIjIz1AlnrPzgAIHVvXxunW7SBr
DhEglQZBbKJEk5kT0mtKoOD1JMrSu1xuTKEBahWRbqHsXclaXjoBADb0kkjVEJu/Lh5hgYZnOjvlba8Ld7HCKePCVePoTJBdI4fvugnL8Tsg
K05aIskyY0hKI9L8KfqfGTl1lzOv2KoWD0KWwtAWPoGChZxmQ+nBli+gwYMzM1vAkP+aayLe0a1EQimlOalO762r0GXO0ks+UeXde2Z4e+8S
/pf7pITEI/tP+MxJTALw9QUWEv9lKTk+jkbqxbsh8nfBUapfKqYn0eidpwq2AzVp3juYl7//fKnaPhJD9gs=
        </item>
    </string-array>
    <string-array name="com_google_android_gms_fonts_certs_prod">
        <item>

MIIEQzCCAyugAwIBAgIJAMLgh0ZkSjCNMA0GCSqGSIb3DQEBBAUAMHQxCzAJBgNVBAYTAlVTMRMwEQYDVQQIEwpDYWxpZm9ybmlhMRYwFAYD
VQQHEw1Nb3VudGFpbiBWaWV3MRQwEgYDVQQKEwtHb29nbGUgSW5jLjEQMA4GA1UECxMHQW5kcm9pZDEQMA4GA1UEAxMHQW5kcm9pZDAeFw0w
ODA4MjEyMzEzMzRaFw0zNjAxMDcyMzEzMzRaMHQxCzAJBgNVBAYTAlVTMRMwEQYDVQQIEwpDYWxpZm9ybmlhMRYwFAYDVQQHEw1Nb3VudGFp
biBWaWV3MRQwEgYDVQQKEwtHb29nbGUgSW5jLjEQMA4GA1UECxMHQW5kcm9pZDEQMA4GA1UEAxMHQW5kcm9pZDCCASAwDQYJKoZIhvcNAQEB
BQADggENADCCAQgCggEBAKtWLgDYO6IIrgqWbxJOKdoR8qtW0I9Y4sypEwPpt1TTcvZApxsdyxMJZ2JORland2qSGT2y5b+3JKkedxiLDmpH
pDsz2WCbdxgxRczfey5YZnTJ4VZbH0xqWVW/8lGmPav5xVwnIiJS6HXk+BVKZF+JcWjAsb/GEuq/eFdpuzSqeYTcfi6idkyugwfYwXFU1+5f
ZKUaRKYCwkkFQVfcAs1fXA5V+++FGfvjJ/CxURaSxaBvGdGDhfXE28LWuT9ozCl5xw4Yq5OGazvV24mZVSoOO0yZ31j7kYvtwYK6NeADwbSx
DdJEqO4k//0zOHKrUiGYXtqw/A0LFFtqoZKFjnkCAQOjgdkwgdYwHQYDVR0OBBYEFMd9jMIhF1Ylmn/Tgt9r45jk14alMIGmBgNVHSMEgZ4w
gZuAFMd9jMIhF1Ylmn/Tgt9r45jk14aloXikdjB0MQswCQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWZvcm5pYTEWMBQGA1UEBxMNTW91bnRh
aW4gVmlldzEUMBIGA1UEChMLR29vZ2xlIEluYy4xEDAOBgNVBAsTB0FuZHJvaWQxEDAOBgNVBAMTB0FuZHJvaWSCCQDC4IdGZEowjTAMBgNV
HRMEBTADAQH/MA0GCSqGSIb3DQEBBAUAA4IBAQBt0lLO74UwLDYKqs6Tm8/yzKkEu116FmH4rkaymUIE0P9KaMftGlMexFlaYjzmB2OxZyl6
euNXEsQH8gjwyxCUKRJNexBiGcCEyj6z+a1fuHHvkiaai+KL8W1EyNmgjmyy8AW7P+LLlkR+ho5zEHatRbM/YAnqGcFh5iZBqpknHf1SKMXF
h4dd239FJ1jWYfbMDMy3NS5CTMQ2XFI1MvcyUTdZPErjQfTbQe3aDQsQcafEQPD+nqActifKZ0Np0IS9L9kR/wbNvyz6ENwPiTrjV2KRkEjH
78ZMcUQXg0L3BYHJ3lc69Vs5Ddf9uUGGMYldX3WfMBEmh/9iFBDAaTCK
        </item>
    </string-array>
</resources>
```

With these resource files in place, the app is capable of downloading the fonts.

Declaring Downloadable Fonts as Resources

By listing the downloadable fonts in the **AndroidManifest.XML**, Android will asynchronously download the fonts when the app first starts. The font's themselves are listed in an array resource file, similar to this one:

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <array name="downloadable_fonts" translatable="false">
        <item>@font/vt323</item>
    </array>
</resources>
```

To download these fonts, they have to be declared in **AndroidManifest.XML** by adding `meta-data` as a child of the `application` element. For example, if the downloadable fonts are declared in a resource file at **Resources/values/downloadable_fonts.xml**, then this snippet would have to be added to the manifest:

```xml
<meta-data android:name="downloadable_fonts" android:resource="@array/downloadable_fonts" />
```

Downloading a Font with the Font APIs

It is possible to programmatically download a font by instantiating a `FontRequest` object and passing that to the `FontContractCompat.RequestFont` method. The `FontContractCompat.RequestFont` method will first check to see if the font exists on the device, and then if necessary will asynchronously query the font provider and try to download the font for the app. If `FontRequest` is unable to download the font, then Android will use the default system font.

A `FontRequest` object contains information that will be used by the font provider to locate and download a font. A `FontRequest` requires four pieces of information:

1. **Font Provider Authority** – The authority of the Font Provider to be used for the request.
2. **Font Package** – The package for the Font Provider to be used for the request. This is used to verify the identity of the provider.
3. **Font Query** – This is a string that will help the font provider locate the requested font. Details on the font query are specific to the font provider. The details of the string are specific to the font provider. The `QueryBuilder` class in the Downloadable Fonts sample app provides some information on the query format for fonts from the Google Fonts Open Source Collection.
4. **Font Provider Certificates** – A resource array with the list of sets of hashes for the certificates the provider should be signed with.

This snippet is an example of instantiating a new `FontRequest` object:

```csharp
FontRequest request = new FontRequest("com.google.android.gms.fonts", "com.google.android.gms",
<FontToDownload>, Resource.Array.com_google_android_gms_fonts_certs);
```

In the previous snippet `FontToDownload` is a query that will help the font from the Google Fonts Open Source collection.

Before passing the `FontRequest` to the `FontContractCompat.RequestFont` method, there are two objects that must be created:

- `FontsContractCompat.FontRequestCallback` – This is an abstract class which must be extended. It is a callback that will be invoked when `RequestFont` is finished. A Xamarin.Android app must subclass `FontsContractCompat.FontRequestCallback` and override the `OnTypefaceRequestFailed` and `OnTypefaceRetrieved`, providing the actions to be taken when the download fails or succeeds respectively.
- `Handler` – This is a `Handler` which will be used by `RequestFont` to download the font on a thread, if necessary. Fonts should **not** be downloaded on the UI thread.

This snippet is an example of a C# class that will asynchronously download a font from Google Fonts Open Source collection. It implements the `FontRequestCallback` interface, and raises a C# event when `FontRequest` has finished.

```
public class FontDownloadHelper : FontsContractCompat.FontRequestCallback
{
    // A very simple font query; replace as necessary
    public static readonly String FontToDownload = "Courgette";

    Android.OS.Handler Handler = null;

    public event EventHandler<FontDownloadEventArg> FontDownloaded = delegate
    {
        // just an empty delegate to avoid null reference exceptions.
    };

    public void DownloadFonts(Context context)
    {
        FontRequest request = new FontRequest("com.google.android.gms.fonts",
"com.google.android.gms",FontToDownload , Resource.Array.com_google_android_gms_fonts_certs);
        FontsContractCompat.RequestFont(context, request, this, GetHandlerThreadHandler());
    }

    public override void OnTypefaceRequestFailed(int reason)
    {
        base.OnTypefaceRequestFailed(reason);
        FontDownloaded(this, new FontDownloadEventArg(null));
    }

    public override void OnTypefaceRetrieved(Android.Graphics.Typeface typeface)
    {
        base.OnTypefaceRetrieved(typeface);
        FontDownloaded(this, new FontDownloadEventArg(typeface));
    }

    Handler GetHandlerThreadHandler()
    {
        if (Handler == null)
        {
            HandlerThread handlerThread = new HandlerThread("fonts");
            handlerThread.Start();
            Handler = new Handler(handlerThread.Looper);
        }
        return Handler;
    }
}

public class FontDownloadEventArg : EventArgs
{
    public FontDownloadEventArg(Android.Graphics.Typeface typeface)
    {
        Typeface = typeface;
    }
    public Android.Graphics.Typeface Typeface { get; private set; }
    public bool RequestFailed
    {
        get
        {
            return Typeface != null;
        }
    }
}
```

To use this helper, a new `FontDownloadHelper` is created, and an event handler is assigned:

```
var fontHelper = new FontDownloadHelper();

fontHelper.FontDownloaded += (object sender, FontDownloadEventArg e) =>
{
    //React to the request
};
fontHelper.DownloadFonts(this); // this is an Android Context instance.
```

Summary

This guide discussed the new APIs in Android 8.0 to support downloadable fonts and fonts as resources. It discussed how to embed existing fonts in an APK and to use them in a layout. It also discussed how Android 8.0 supports downloading fonts from a font provider, either programmatically or by declaring the font meta-data in resource files.

Related Links

- fontFamily
- FontConfig
- FontRequest
- FontsContractCompat
- Resources.GetFont
- Typeface
- Android Support Library 26 NuGet
- Using Fonts in Android
- CSS font weight specification
- Google Fonts Open Source collection
- Source Sans Pro

Activity Lifecycle

7/30/2021 • 17 minutes to read • Edit Online

Activities are a fundamental building block of Android applications and they can exist in a number of different states. The activity lifecycle begins with instantiation and ends with destruction, and includes many states in between. When an activity changes state, the appropriate lifecycle event method is called, notifying the activity of the impending state change and allowing it to execute code to adapt to that change. This article examines the lifecycle of activities and explains the responsibility that an activity has during each of these state changes to be part of a well-behaved, reliable application.

Activity Lifecycle Overview

Activities are an unusual programming concept specific to Android. In traditional application development there is usually a static main method, which is executed to launch the application. With Android, however, things are different; Android applications can be launched via any registered activity within an application. In practice, most applications will only have a specific activity that is specified as the application entry point. However, if an application crashes, or is terminated by the OS, the OS can try to restart the application at the last open activity or anywhere else within the previous activity stack. Additionally, the OS may pause activities when they're not active, and reclaim them if it is low on memory. Careful consideration must be made to allow the application to correctly restore its state in the event that an activity is restarted, especially if that activity depends on data from previous activities.

The activity lifecycle is implemented as a collection of methods the OS calls throughout the lifecycle of an activity. These methods allow developers to implement the functionality that is necessary to satisfy the state and resource management requirements of their applications.

It is extremely important for the application developer to analyze the requirements of each activity to determine which methods exposed by the activity lifecycle need to be implemented. Failure to do this can result in application instability, crashes, resource bloat, and possibly even underlying OS instability.

This chapter examines the activity lifecycle in detail, including:

- Activity States
- Lifecycle Methods
- Retaining the State of an Application

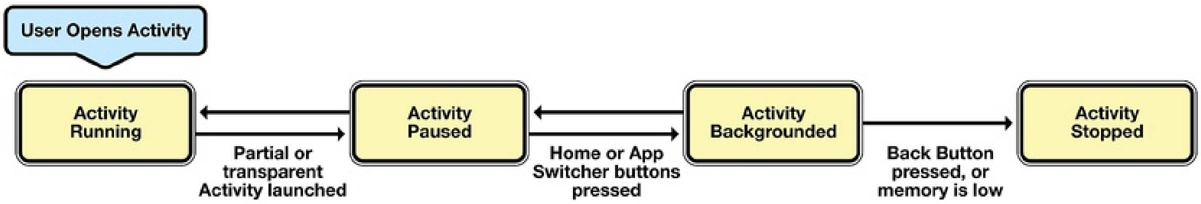
This section also includes a walkthrough that provide practical examples on how to efficiently save state during the Activity lifecycle. By the end of this chapter you should have an understanding of the Activity lifecycle and how to support it in an Android application.

Activity Lifecycle

The Android activity lifecycle comprises a collection of methods exposed within the Activity class that provide the developer with a resource management framework. This framework allows developers to meet the unique state management requirements of each activity within an application and properly handle resource management.

Activity States

The Android OS arbitrates Activities based on their state. This helps Android identify activities that are no longer in use, allowing the OS to reclaim memory and resources. The following diagram illustrates the states an Activity can go through during its lifetime:



These states can be broken into 4 main groups as follows:

1. *Active or Running* – Activities are considered active or running if they are in the foreground, also known as the top of the activity stack. This is considered the highest priority activity in Android, and as such will only be killed by the OS in extreme situations, such as if the activity tries to use more memory than is available on the device as this could cause the UI to become unresponsive.

2. *Paused* – When the device goes to sleep, or an activity is still visible but partially hidden by a new, non-full-sized or transparent activity, the activity is considered paused. Paused activities are still alive, that is, they maintain all state and member information, and remain attached to the window manager. This is considered to be the second highest priority activity in Android and, as such, will only be killed by the OS if killing this activity will satisfy the resource requirements needed to keep the Active/Running Activity stable and responsive.

3. *Stopped/Backgrounded* – Activities that are completely obscured by another activity are considered stopped or in the background. Stopped activities still try to retain their state and member information for as long as possible, but stopped activities are considered to be the lowest priority of the three states and, as such, the OS will kill activities in this state first to satisfy the resource requirements of higher priority activities.

4. *Restarted* – It is possible for an activity that is anywhere from paused to stopped in the lifecycle to be removed from memory by Android. If the user navigates back to the activity it must be restarted, restored to its previously saved state, and then displayed to the user.

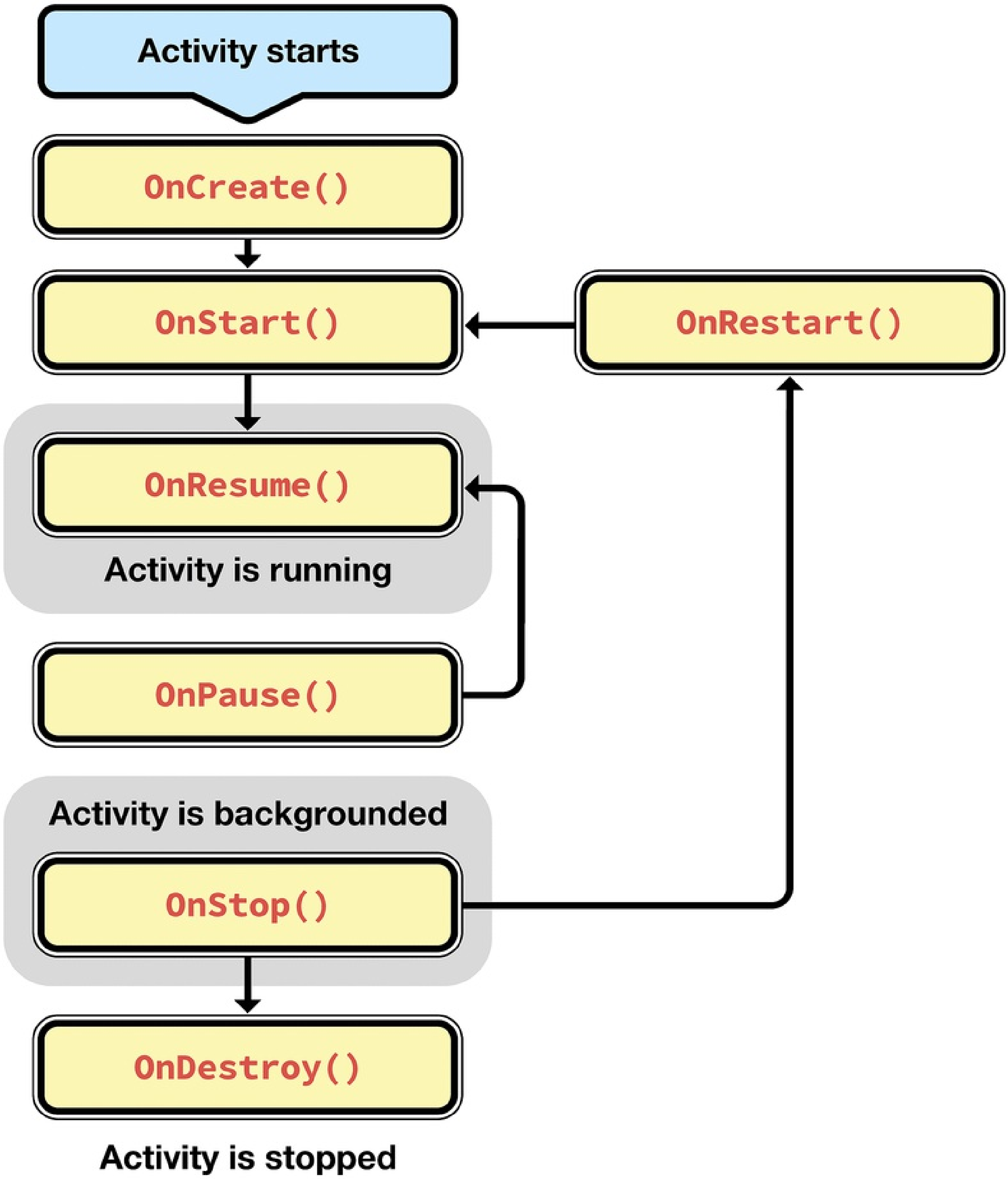
Activity Re-Creation in Response to Configuration Changes

To make matters more complicated, Android throws one more wrench in the mix called Configuration Changes. Configuration changes are rapid activity destruction/re-creation cycles that occur when the configuration of an activity changes, such as when the device is rotated (and the activity needs to get re-built in landscape or portrait mode), when the keyboard is displayed (and the activity is presented with an opportunity to resize itself), or when the device is placed in a dock, among others.

Configuration changes still cause the same Activity State changes that would occur during stopping and restarting an activity. However, in order to make sure that an application feels responsive and performs well during configuration changes, it's important that they are handled as quickly as possible. Because of this, Android has a specific API that can be used to persist state during configuration changes. We'll cover this later in the Managing State Throughout the Lifecycle section.

Activity Lifecycle Methods

The Android SDK and, by extension, the Xamarin.Android framework provide a powerful model for managing the state of activities within an application. When an activity's state is changing, the activity is notified by the OS, which calls specific methods on that activity. The following diagram illustrates these methods in relation to the Activity Lifecycle:



As a developer, you can handle state changes by overriding these methods within an activity. It's important to note, however, that all lifecycle methods are called on the UI thread and will block the OS from performing the next piece of UI work, such as hiding the current activity, displaying a new activity, etc. As such, code in these methods should be as brief as possible to make an application feel well performing. Any long-running tasks should be executed on a background thread.

Let's examine each of these lifecycle methods and their use:

OnCreate

OnCreate is the first method to be called when an activity is created. `OnCreate` is always overridden to perform any startup initializations that may be required by an Activity such as:

- Creating views
- Initializing variables
- Binding static data to lists

`OnCreate` takes a Bundle parameter, which is a dictionary for storing and passing state information and objects between activities If the bundle is not null, this indicates the activity is restarting and it should restore its state from the previous instance. The following code illustrates how to retrieve values from the bundle:

```
protected override void OnCreate(Bundle bundle)
{
   base.OnCreate(bundle);

   string intentString;
   bool intentBool;

   if (bundle != null)
   {
      intentString = bundle.GetString("myString");
      intentBool = bundle.GetBoolean("myBool");
   }

   // Set our view from the "main" layout resource
   SetContentView(Resource.Layout.Main);
}
```

Once `OnCreate` has finished, Android will call `OnStart`.

OnStart

OnStart is always called by the system after `OnCreate` is finished. Activities may override this method if they need to perform any specific tasks right before an activity becomes visible such as refreshing current values of views within the activity. Android will call `OnResume` immediately after this method.

OnResume

The system calls OnResume when the Activity is ready to start interacting with the user. Activities should override this method to perform tasks such as:

- Ramping up frame rates (a common task in game development)
- Starting animations
- Listening for GPS updates
- Display any relevant alerts or dialogs
- Wire up external event handlers

As an example, the following code snippet shows how to initialize the camera:

```
protected override void OnResume()
{
    base.OnResume(); // Always call the superclass first.

    if (_camera==null)
    {
        // Do camera initializations here
    }
}
```

`OnResume` is important because any operation that is done in `OnPause` should be un-done in `OnResume`, since it's the only lifecycle method that is guaranteed to execute after `OnPause` when bringing the activity back to life.

OnPause

OnPause is called when the system is about to put the activity into the background or when the activity becomes partially obscured. Activities should override this method if they need to:

- Commit unsaved changes to persistent data

- Destroy or clean up other objects consuming resources

- Ramp down frame rates and pausing animations
- Unregister external event handlers or notification handlers (i.e. those that are tied to a service). This must be done to prevent Activity memory leaks.
- Likewise, if the Activity has displayed any dialogs or alerts, they must be cleaned up with the `.Dismiss()` method.

As an example, the following code snippet will release the camera, as the Activity cannot make use of it while paused:

```
protected override void OnPause()
{
    base.OnPause(); // Always call the superclass first

    // Release the camera as other activities might need it
    if (_camera != null)
    {
        _camera.Release();
        _camera = null;
    }
}
```

There are two possible lifecycle methods that will be called after `OnPause`:

1. `OnResume` will be called if the Activity is to be returned to the foreground.
2. `OnStop` will be called if the Activity is being placed in the background.

OnStop

OnStop is called when the activity is no longer visible to the user. This happens when one of the following occurs:

- A new activity is being started and is covering up this activity.
- An existing activity is being brought to the foreground.
- The activity is being destroyed.

`OnStop` may not always be called in low-memory situations, such as when Android is starved for resources and cannot properly background the Activity. For this reason, it is best not to rely on `OnStop` getting called when preparing an Activity for destruction. The next lifecycle methods that may be called after this one will be `OnDestroy` if the Activity is going away, or `OnRestart` if the Activity is coming back to interact with the user.

OnDestroy

OnDestroy is the final method that is called on an Activity instance before it's destroyed and completely removed from memory. In extreme situations Android may kill the application process that is hosting the Activity, which will result in `OnDestroy` not being invoked. Most Activities will not implement this method because most clean up and shut down has been done in the `OnPause` and `OnStop` methods. The `OnDestroy` method is typically overridden to clean up long running tasks that might leak resources. An example of this might be background threads that were started in `OnCreate`.

There will be no lifecycle methods called after the Activity has been destroyed.

OnRestart

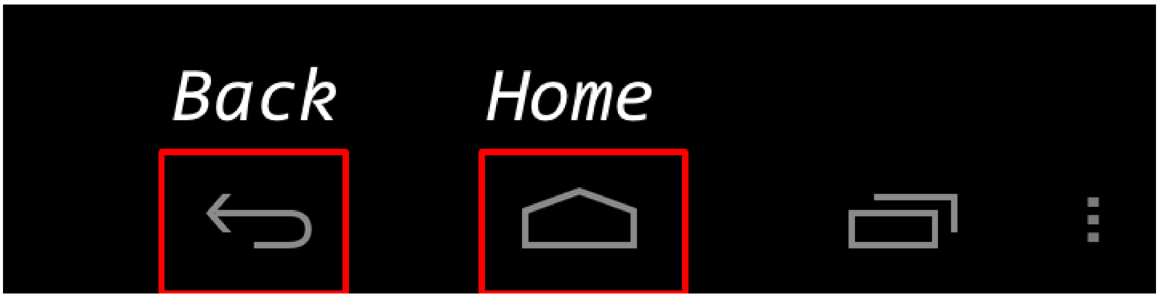
OnRestart is called after your activity has been stopped, prior to it being started again. A good example of this would be when the user presses the home button while on an activity in the application. When this happens `OnPause` and then `OnStop` methods are called, and the Activity is moved to the background but is not destroyed. If the user were then to restore the application by using the task manager or a similar application, Android will call the `OnRestart` method of the activity.

There are no general guidelines for what kind of logic should be implemented in `OnRestart`. This is because `OnStart` is always invoked regardless of whether the Activity is being created or being restarted, so any resources required by the Activity should be initialized in `OnStart`, rather than `OnRestart`.

The next lifecycle method called after `OnRestart` will be `OnStart`.

Back vs. Home

Many Android devices have two distinct buttons: a "Back" button and a "Home" button. An example of this can be seen in the following screenshot of Android 4.0.3:



There is a subtle difference between the two buttons, even though they appear to have the same effect of putting an application in the background. When a user clicks the Back button, they are telling Android that they are done with the activity. Android will destroy the Activity. In contrast, when the user clicks the Home button the activity is merely placed into the background – Android will not kill the activity.

Managing State Throughout the Lifecycle

When an Activity is stopped or destroyed the system provides an opportunity to save the state of the Activity for later rehydration. This saved state is referred to as instance state. Android provides three options for storing instance state during the Activity lifecycle:

1. Storing primitive values in a `Dictionary` known as a Bundle that Android will use to save state.
2. Creating a custom class that will hold complex values such as bitmaps. Android will use this custom class to save state.
3. Circumventing the configuration change lifecycle and assuming complete responsibility for maintaining state in the activity.

This guide covers the first two options.

Bundle State

The primary option for saving instance state is to use a key/value dictionary object known as a Bundle. Recall that when an Activity is created that the `OnCreate` method is passed a bundle as a parameter, this bundle can be used to restore the instance state. It is not recommended to use a bundle for more complex data that won't quickly or easily serialize to key/value pairs (such as bitmaps); rather, it should be used for simple values like strings.

An Activity provides methods to help with saving and retrieving the instance state in the Bundle:

- OnSaveInstanceState – This is invoked by Android when the activity is being destroyed. Activities can implement this method if they need to persist any key/value state items.
- OnRestoreInstanceState – This is called after the `OnCreate` method is finished, and provides another opportunity for an Activity to restore its state after initialization is complete.

The following diagram illustrates how these methods are used:

OnSaveInstanceState

OnSaveInstanceState will be called as the Activity is being stopped. It will receive a bundle parameter that the Activity can store its state in. When a device experiences a configuration change, an Activity can use the `Bundle` object that is passed in to preserve the Activity state by overriding `OnSaveInstanceState`. For example, consider the following code:

```
int c;

protected override void OnCreate (Bundle bundle)
{
  base.OnCreate (bundle);

  this.SetContentView (Resource.Layout.SimpleStateView);

  var output = this.FindViewById<TextView> (Resource.Id.outputText);

  if (bundle != null) {
    c = bundle.GetInt ("counter", -1);
  } else {
    c = -1;
  }

  output.Text = c.ToString ();

  var incrementCounter = this.FindViewById<Button> (Resource.Id.incrementCounter);

  incrementCounter.Click += (s,e) => {
    output.Text = (++c).ToString();
  };
}
```

The code above increments an integer named `c` when a button named `incrementCounter` is clicked, displaying the result in a `TextView` named `output`. When a configuration change happens - for example, when the device is rotated - the above code would lose the value of `c` because the `bundle` would be `null`, as shown in the figure below:

To preserve the value of `c` in this example, the Activity can override `OnSaveInstanceState`, saving the value in the bundle as shown below:

```
protected override void OnSaveInstanceState (Bundle outState)
{
  outState.PutInt ("counter", c);
  base.OnSaveInstanceState (outState);
}
```

Now when the device is rotated to a new orientation, the integer is saved in the bundle and is retrieved with the line:

```
c = bundle.GetInt ("counter", -1);
```

NOTE

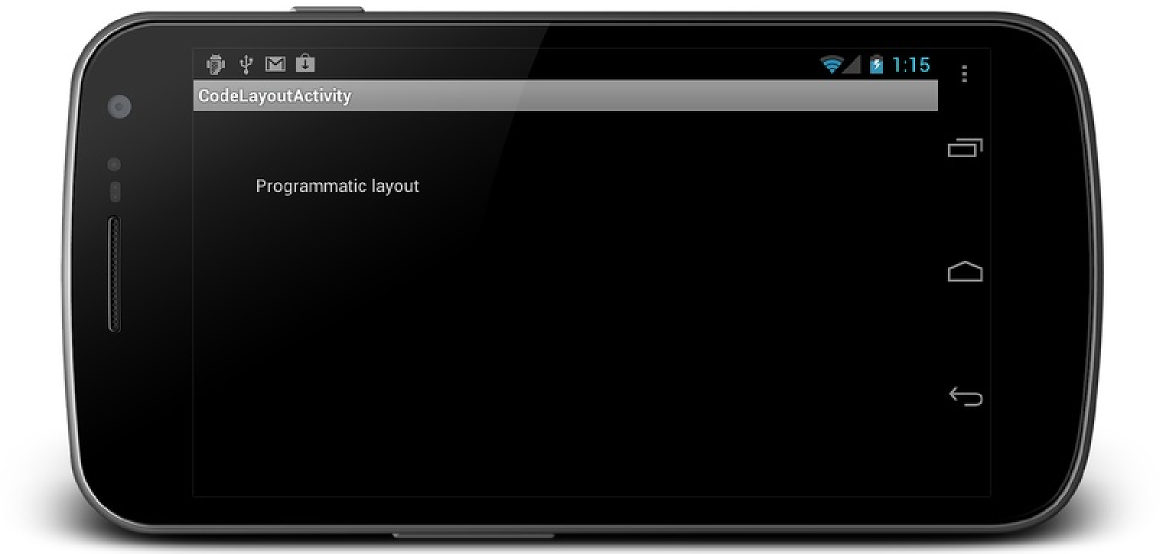
It is important to always call the base implementation of `OnSaveInstanceState` so that the state of the view hierarchy can also be saved.

View State

Overriding `OnSaveInstanceState` is an appropriate mechanism for saving transient data in an Activity across orientation changes, such as the counter in the above example. However, the default implementation of `OnSaveInstanceState` will take care of saving transient data in the UI for every view, so long as each view has an ID assigned. For example, say an application has an `EditText` element defined in XML as follows:

```
<EditText android:id="@+id/myText"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"/>
```

Since the `EditText` control has an `id` assigned, when the user enters some data and rotates the device, the data is still displayed, as shown below:



OnRestoreInstanceState

OnRestoreInstanceState will be called after `OnStart`. It provides an activity the opportunity to restore any state that was previously saved to a Bundle during the previous `OnSaveInstanceState`. This is the same bundle that is provided to `OnCreate`, however.

The following code demonstrates how state can be restored in `OnRestoreInstanceState`:

```
protected override void OnRestoreInstanceState(Bundle savedState)
{
    base.OnRestoreInstanceState(savedState);
    var myString = savedState.GetString("myString");
    var myBool = savedState.GetBoolean("myBool");
}
```

This method exists to provide some flexibility around when state should be restored. Sometimes it is more appropriate to wait until all initializations are done before restoring instance state. Additionally, a subclass of an existing Activity may only want to restore certain values from the instance state. In many cases, it's not necessary to override `OnRestoreInstanceState`, since most activities can restore state using the bundle provided to `OnCreate`.

For an example of saving state using a `Bundle`, refer to the Walkthrough - Saving the Activity state.

Bundle Limitations

Although `OnSaveInstanceState` makes it easy to save transient data, it has some limitations:

- It is not called in all cases. For example, pressing **Home** or **Back** to exit an Activity will not result in `OnSaveInstanceState` being called.
- The bundle passed into `OnSaveInstanceState` is not designed for large objects, such as images. In the case of large objects, saving the object from OnRetainNonConfigurationInstance is preferable, as discussed below.
- Data saved by using the bundle is serialized, which can lead to delays.

Bundle state is useful for simple data that doesn't use much memory, whereas *non-configuration instance data* is useful for more complex data, or data that is expensive to retrieve, such as from a web service call or a complicated database query. Non-configuration instance data gets saved in an object as needed. The next section introduces `OnRetainNonConfigurationInstance` as a way of preserving more complex data types through configuration changes.

Persisting Complex Data

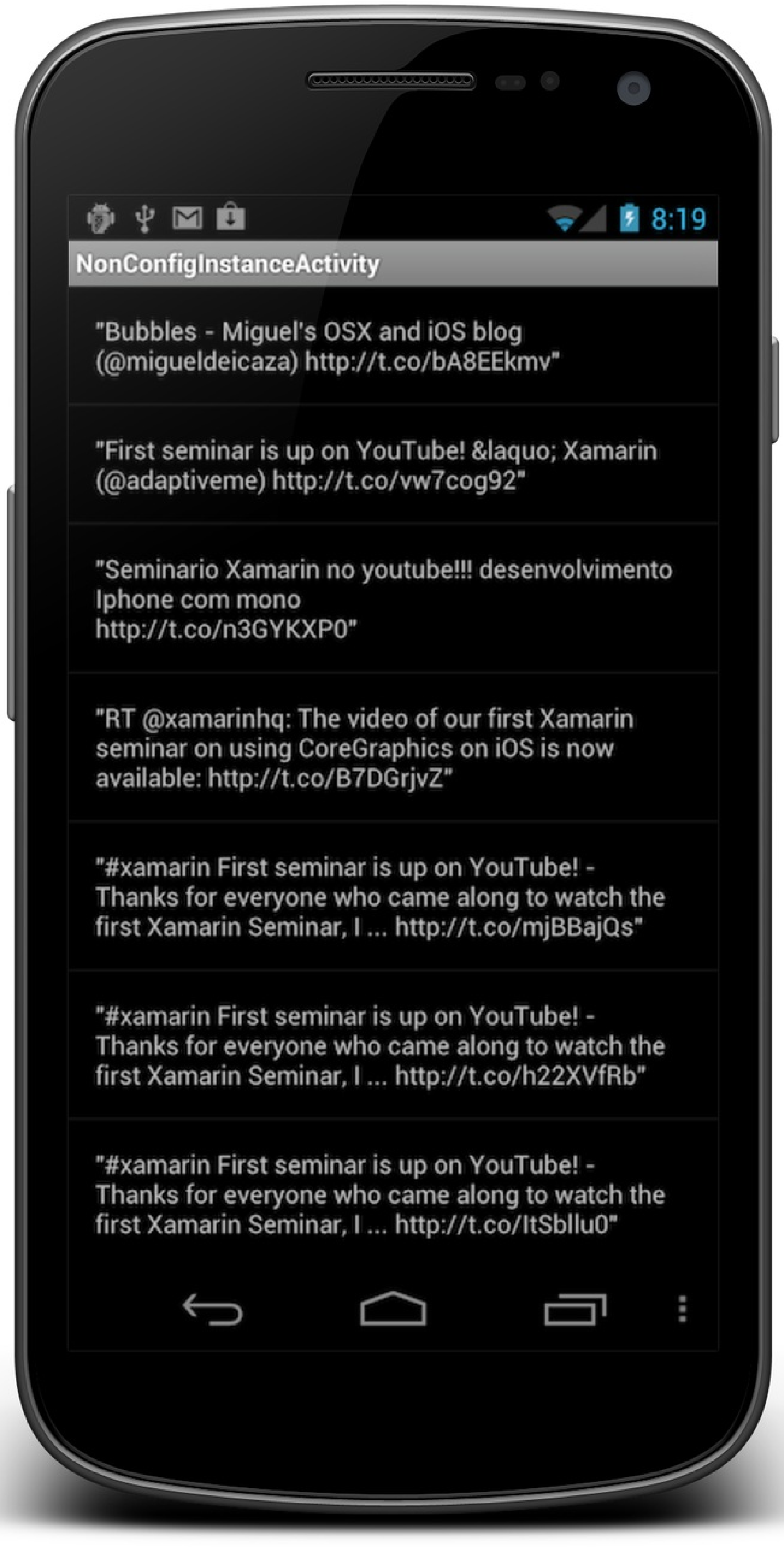
In addition to persisting data in the bundle, Android also supports saving data by overriding OnRetainNonConfigurationInstance and returning an instance of a `Java.Lang.Object` that contains the data to persist. There are two primary benefits of using `OnRetainNonConfigurationInstance` to save state:

- The object returned from `OnRetainNonConfigurationInstance` performs well with larger, more complex data types because memory retains this object.
- The `OnRetainNonConfigurationInstance` method is called on demand, and only when needed. This is more economical than using a manual cache.

Using `OnRetainNonConfigurationInstance` is suitable for scenarios where it is expensive to retrieve the data multiple times, such as in web service calls. For example, consider the following code that searches Twitter:

```
public class NonConfigInstanceActivity : ListActivity
{
  protected override void OnCreate (Bundle bundle)
  {
    base.OnCreate (bundle);
    SearchTwitter ("xamarin");
  }

  public void SearchTwitter (string text)
  {
    string searchUrl = String.Format("http://search.twitter.com/search.json?" + "q=
{0}&rpp=10&include_entities=false&" + "result_type=mixed", text);

    var httpReq = (HttpWebRequest)HttpWebRequest.Create (new Uri (searchUrl));
    httpReq.BeginGetResponse (new AsyncCallback (ResponseCallback), httpReq);
  }

  void ResponseCallback (IAsyncResult ar)
  {
    var httpReq = (HttpWebRequest)ar.AsyncState;

    using (var httpRes = (HttpWebResponse)httpReq.EndGetResponse (ar)) {
      ParseResults (httpRes);
    }
  }

  void ParseResults (HttpWebResponse httpRes)
  {
    var s = httpRes.GetResponseStream ();
    var j = (JsonObject)JsonObject.Load (s);

    var results = (from result in (JsonArray)j ["results"] let jResult = result as JsonObject select jResult
["text"].ToString ()).ToArray ();

    RunOnUiThread (() => {
      PopulateTweetList (results);
    });
  }

  void PopulateTweetList (string[] results)
  {
    ListAdapter = new ArrayAdapter<string> (this, Resource.Layout.ItemView, results);
  }
}
```

This code retrieves results from the web formatted as JSON, parses them, and then presents the results in a list, as shown in the following screenshot:



When a configuration change occurs - for example, when a device is rotated - the code repeats the process. To reuse the originally retrieved results and not cause needless, redundant network calls, we can use `OnRetainNonconfigurationInstance` to save the results, as shown below:

```
public class NonConfigInstanceActivity : ListActivity
{
  TweetListWrapper _savedInstance;

  protected override void OnCreate (Bundle bundle)
  {
    base.OnCreate (bundle);

    var tweetsWrapper = LastNonConfigurationInstance as TweetListWrapper;

    if (tweetsWrapper != null) {
      PopulateTweetList (tweetsWrapper.Tweets);
    } else {
      SearchTwitter ("xamarin");
    }

    public override Java.Lang.Object OnRetainNonConfigurationInstance ()
    {
      base.OnRetainNonConfigurationInstance ();
      return _savedInstance;
    }

    ...

    void PopulateTweetList (string[] results)
    {
      ListAdapter = new ArrayAdapter<string> (this, Resource.Layout.ItemView, results);
      _savedInstance = new TweetListWrapper{Tweets=results};
    }
}
```

Now when the device is rotated, the original results are retrieved from the `LastNonConfiguartionInstance` property. In this example, the results consist of a `string[]` containing tweets. Since `OnRetainNonConfigurationInstance` requires that a `Java.Lang.Object` be returned, the `string[]` is wrapped in a class that subclasses `Java.Lang.Object`, as shown below:

```
class TweetListWrapper : Java.Lang.Object
{
  public string[] Tweets { get; set; }
}
```

For example, attempting to use a `TextView` as the object returned from `OnRetainNonConfigurationInstance` will leak the Activity, as illustrated by the code below:

```
TextView _textView;

protected override void OnCreate (Bundle bundle)
{
  base.OnCreate (bundle);

  var tv = LastNonConfigurationInstance as TextViewWrapper;

  if(tv != null) {
    _textView = tv;
    var parent = _textView.Parent as FrameLayout;
    parent.RemoveView(_textView);
  } else {
    _textView = new TextView (this);
    _textView.Text = "This will leak.";
  }

  SetContentView (_textView);
}

public override Java.Lang.Object OnRetainNonConfigurationInstance ()
{
  base.OnRetainNonConfigurationInstance ();
  return _textView;
}
```

In this section, we learned how to preserve simple state data with the `Bundle`, and persist more complex data types with `OnRetainNonConfigurationInstance`.

Summary

The Android activity lifecycle provides a powerful framework for state management of activities within an application but it can be tricky to understand and implement. This chapter introduced the different states that an activity may go through during its lifetime, as well as the lifecycle methods that are associated with those states. Next, guidance was provided as to what kind of logic should be performed in each of these methods.

Related Links

- Handling Rotation
- Android Activity

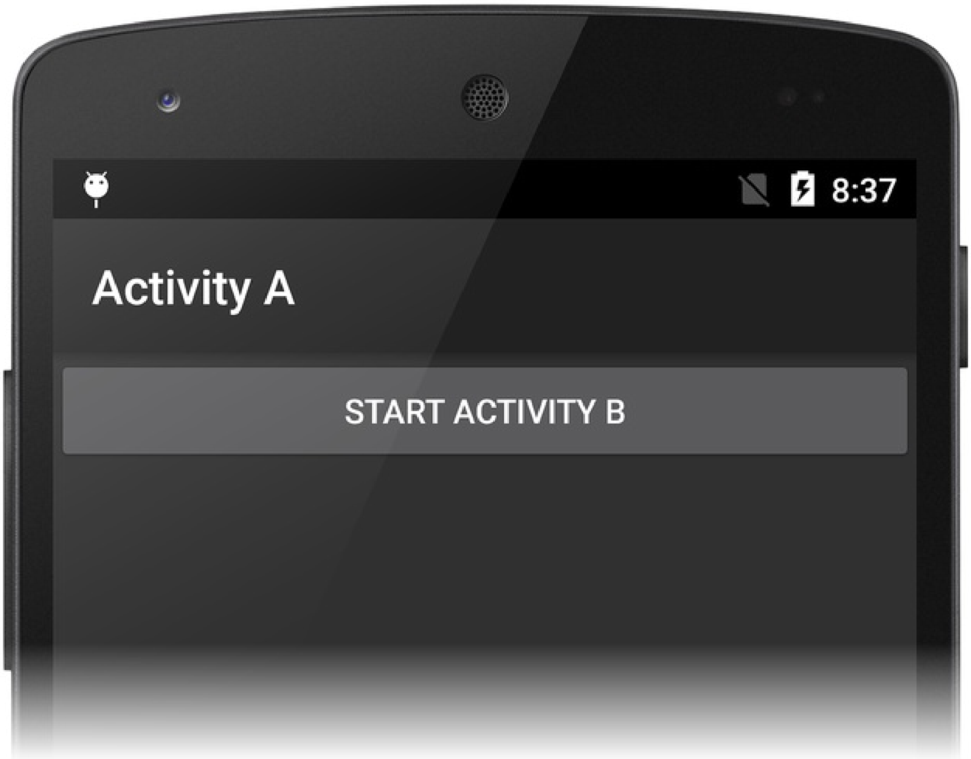
Walkthrough - Saving the Activity state

7/8/2021 • 4 minutes to read • Edit Online

We have covered the theory behind saving state in the Activity Lifecycle guide; now, let's walk through an example.

Activity State Walkthrough

Let's open the **ActivityLifecycle_Start** project (in the ActivityLifecycle sample), build it, and run it. This is a very simple project that has two activities to demonstrate the activity lifecycle and how the various lifecycle methods are called. When you start the application, the screen of `MainActivity` is displayed:



Viewing State Transitions

Each method in this sample writes to the IDE application output window to indicate activity state. (To open the output window in Visual Studio, type **CTRL-ALT-O**; to open the output window in Visual Studio for Mac, click **View > Pads > Application Output**.)

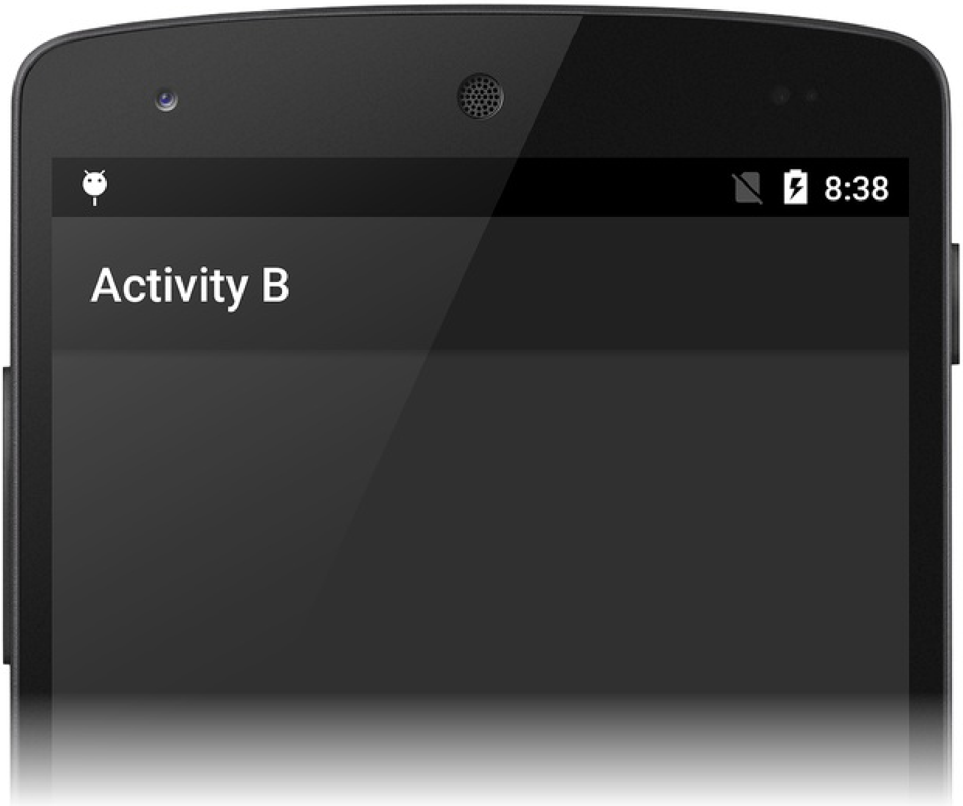
When the app first starts, the output window displays the state changes of *Activity A*:

```
[ActivityLifecycle.MainActivity] Activity A - OnCreate
[ActivityLifecycle.MainActivity] Activity A - OnStart
[ActivityLifecycle.MainActivity] Activity A - OnResume
```

When we click the **Start Activity B** button, we see *Activity A* pause and stop while *Activity B* goes through its state changes:

```
[ActivityLifecycle.MainActivity] Activity A - OnPause
[ActivityLifecycle.SecondActivity] Activity B - OnCreate
[ActivityLifecycle.SecondActivity] Activity B - OnStart
[ActivityLifecycle.SecondActivity] Activity B - OnResume
[ActivityLifecycle.MainActivity] Activity A - OnStop
```

As a result, *Activity B* is started and displayed in place of *Activity A*:



When we click the **Back** button, *Activity B* is destroyed and *Activity A* is resumed:

```
[ActivityLifecycle.SecondActivity] Activity B - OnPause
[ActivityLifecycle.MainActivity] Activity A - OnRestart
[ActivityLifecycle.MainActivity] Activity A - OnStart
[ActivityLifecycle.MainActivity] Activity A - OnResume
[ActivityLifecycle.SecondActivity] Activity B - OnStop
[ActivityLifecycle.SecondActivity] Activity B - OnDestroy
```

Adding a Click Counter

Next, we're going to change the application so that we have a button that counts and displays the number of times it is clicked. First, let's add a `_counter` instance variable to `MainActivity`:

```
int _counter = 0;
```

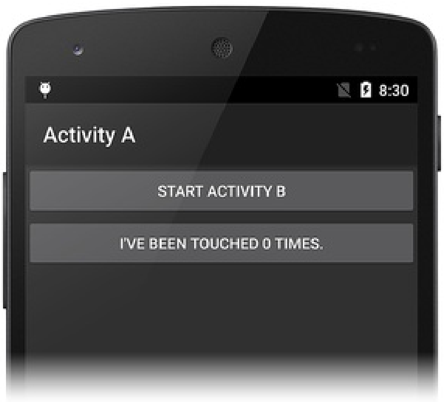
Next, let's edit the **Resource/layout/Main.axml** layout file and add a new `clickButton` that displays the number of times the user has clicked the button. The resulting **Main.axml** should resemble the following:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <Button
        android:id="@+id/myButton"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/mybutton_text" />
    <Button
        android:id="@+id/clickButton"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/counterbutton_text" />
</LinearLayout>
```

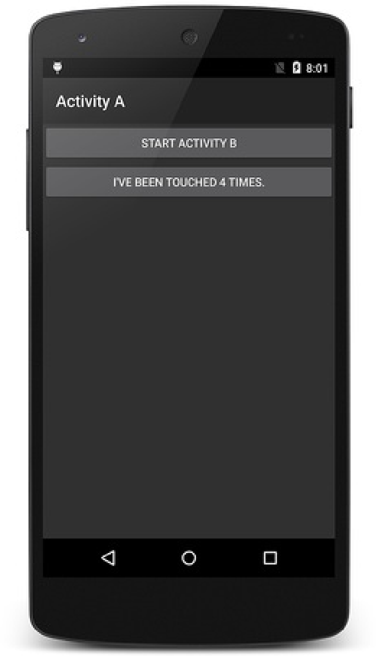
Let's add the following code to the end of the OnCreate method in `MainActivity` – this code handles click events from the `clickButton`:

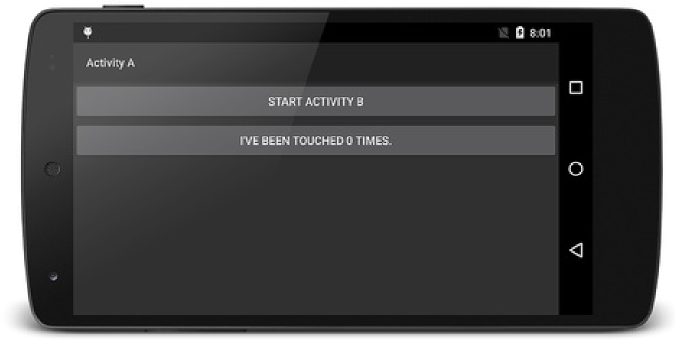
```
var clickbutton = FindViewById<Button> (Resource.Id.clickButton);
clickbutton.Text = Resources.GetString (
    Resource.String.counterbutton_text, _counter);
clickbutton.Click += (object sender, System.EventArgs e) =>
{
    _counter++;
    clickbutton.Text = Resources.GetString (
        Resource.String.counterbutton_text, _counter);
} ;
```

When we build and run the app again, a new button appears that increments and displays the value of `_counter` on each click:



But when we rotate the device to landscape mode, this count is lost:





Examining the application output, we see that *Activity A* was paused, stopped, destroyed, recreated, restarted, then resumed during the rotation from portrait to landscape mode:

```
[ActivityLifecycle.MainActivity] Activity A - OnPause
[ActivityLifecycle.MainActivity] Activity A - OnStop
[ActivityLifecycle.MainActivity] Activity A - On Destroy

[ActivityLifecycle.MainActivity] Activity A - OnCreate
[ActivityLifecycle.MainActivity] Activity A - OnStart
[ActivityLifecycle.MainActivity] Activity A - OnResume
```

Because *Activity A* is destroyed and recreated again when the device is rotated, its instance state is lost. Next, we will add code to save and restore the instance state.

Adding Code to Preserve Instance State

Let's add a method to `MainActivity` to save the instance state. Before *Activity A* is destroyed, Android automatically calls OnSaveInstanceState and passes in a Bundle that we can use to store our instance state. Let's use it to save our click count as an integer value:

```
protected override void OnSaveInstanceState (Bundle outState)
{
    outState.PutInt ("click_count", _counter);
    Log.Debug(GetType().FullName, "Activity A - Saving instance state");

    // always call the base implementation!
    base.OnSaveInstanceState (outState);
}
```

When *Activity A* is recreated and resumed, Android passes this `Bundle` back into our `OnCreate` method. Let's add code to `OnCreate` to restore the `_counter` value from the passed-in `Bundle`. Add the following code just before the line where `clickbutton` is defined:

```
if (bundle != null)
{
    _counter = bundle.GetInt ("click_count", 0);
    Log.Debug(GetType().FullName, "Activity A - Recovered instance state");
}
```

Build and run the app again, then click the second button a few times. When we rotate the device to landscape mode, the count is preserved!

Let's take a look at the output window to see what happened:

```
[ActivityLifecycle.MainActivity] Activity A - OnPause
[ActivityLifecycle.MainActivity] Activity A - Saving instance state
[ActivityLifecycle.MainActivity] Activity A - OnStop
[ActivityLifecycle.MainActivity] Activity A - On Destroy

[ActivityLifecycle.MainActivity] Activity A - OnCreate
[ActivityLifecycle.MainActivity] Activity A - Recovered instance state
[ActivityLifecycle.MainActivity] Activity A - OnStart
[ActivityLifecycle.MainActivity] Activity A - OnResume
```

Before the OnStop method was called, our new `OnSaveInstanceState` method was called to save the `_counter` value in a `Bundle`. Android passed this `Bundle` back to us when it called our `OnCreate` method, and we were able to used it to restore the `_counter` value to where we left off.

Summary

In this walkthough, we have used our knowledge of the Activity Lifecycle to preserve state data.

Related Links

- ActivityLifecycle (sample)
- Activity Lifecycle
- Android Activity

Creating Android Services

7/8/2021 • 9 minutes to read • Edit Online

This guide discusses Xamarin.Android services, which are Android components that allow work to be done without an active user interface. Services are very commonly used for tasks that are performed in the background, such as time consuming calculations, downloading files, playing music, and so on. It explains the different scenarios that services are suited for and shows how to implement them both for performing long-running background tasks as well as for providing an interface for remote procedure calls.

Android Services Overview

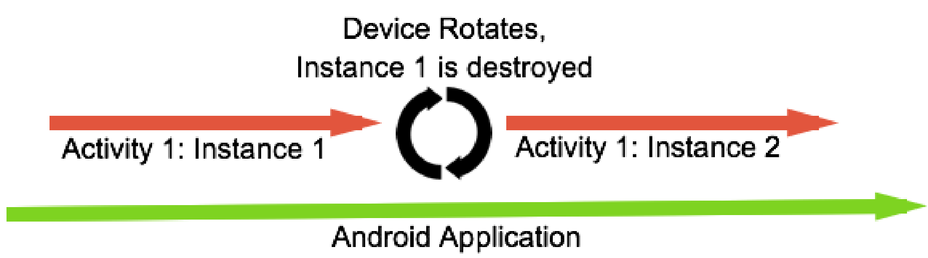
Mobile apps are not like desktop apps. Desktops have copious amounts of resources such as screen real estate, memory, storage space, and a connected power supply, mobile devices do not. These constraints force mobile apps to behave differently. For example, the small screen on a mobile device typically means that only one app (i.e. Activity) is visible at a time. Other Activities are moved to the background and pushed into a suspended state where they cannot perform any work. However, just because an Android application is in the background does not mean that it is impossible for app to keep working.

Android applications are made up of at least one of the following four primary components: *Activities*, *Broadcast Receivers*, *Content Providers*, and *Services*. Activities are the cornerstone of many great Android applications because they provide the UI that allows a user to interact with the application. However, when it comes to performing concurrent or background work, Activities are not always the best choice.

The primary mechanism for background work in Android is the *service*. An Android service is a component that is designed to do some work without a user interface. A service might download a file, play music, or apply a filter to an image. Services can also be used for interprocess communication (*IPC*) between Android applications. For example one Android app might use the music player service that is from another app or an app might expose data (such as a person's contact information) to other apps via a service.

Services, and their ability to perform background work, are crucial to providing a smooth and fluid user interface. All Android applications have a *main thread* (also known as a *UI thread*) on which the Activities are run. To keep the device responsive, Android must be able to update the user interface at the rate of 60 frames per second. If an Android app performs too much work on the main thread, then Android will drop frames, which in turn causes the UI to appear jerky (also sometimes referred to as *janky*). This means that any work performed on the UI thread should complete in the time span between two frames, approximately 16 milliseconds (1 second every 60 frames).

To address this concern, a developer may use threads in an Activity to perform some work that would block the UI. However, this could cause problems. It is very possible that Android will destroy and recreate the multiple instances of the Activity. However, Android will not automatically destroy the threads, which could result in memory leaks. A prime example of this is when the device is rotated – Android will try to destroy the instance of the Activity and then recreate a new one:



This is a potential memory leak – the thread created by the first instance of the Activity will still be running. If the thread has a reference to the first instance of the Activity, this will prevent Android from garbage collecting the object. However, the second instance of the Activity is still created (which in turn might create a new thread). Rotating the device several times in rapid succession may exhaust all the RAM and force Android to terminate the entire application to reclaim memory.

As a rule of thumb, if the work to be performed should outlive an Activity, then a service should be created to perform that work. However, if the work is only applicable in the context of an Activity, then creating a thread to perform the work might be more appropriate. For example, creating a thumbnail for a photo that was just added to a photo gallery app should probably occur in a service. However, a thread might be more appropriate to play some music that should only be heard while an Activity is in the foreground.

Background work can be broken down into two broad classifications:

1. **Long Running Task** – This is work that is ongoing until explicitly stopped. An example of a *long running task* is an app that streams music or that must monitor data collected from a sensor. These tasks must run even though the application has no visible user interface.
2. **Periodic Tasks** – (sometimes referred to as a *job*) A periodic task is one that is of relatively short in duration (several seconds) and is run on a schedule (i.e. once a day for a week or perhaps just once in the next 60 seconds). An example of this is downloading a file from the internet or generating a thumbnail for an image.

There are four different types of Android services:

- **Bound Service** – A *bound service* is a service that has some other component (typically an Activity) bound to it. A bound service provides an interface that allows the bound component and the service to interact with each other. Once there are no more clients bound to the service, Android will shut the service down.

- `IntentService` – An `IntentService` is a specialized subclass of the `Service` class that simplifies service creation and usage. An `IntentService` is meant to handle individual autonomous calls. Unlike a service, which can concurrently handle multiple calls, an `IntentService` is more like a *work queue processor* – work is queued up and an `IntentService` processes each job one at a time on a single worker thread. Typically, an `IntentService` is not bound to an Activity or a Fragment.

- **Started Service** – A *started service* is a service that has been started by some other Android component (such as an Activity) and is run continuously in the background until something explicitly tells the service to stop. Unlike a bound service, a started service does not have any clients directly bound to it. For this reason, it is important to design started services so that they may be gracefully restarted as necessary.

- **Hybrid Service** – A *hybrid service* is a service that has the characteristics of a *started service* and a *bound service*. A hybrid service can be started by when a component binds to it or it may be started by some event. A client component may or may not be bound to the hybrid service. A hybrid service will keep running until it is explicitly told to stop, or until there are no more clients bound to it.

Which type of service to use is very dependent on application requirements. As a rule of thumb, an `IntentService` or a bound service are sufficient for most tasks that an Android application must perform, so preference should be given to one of those two types of services. An `IntentService` is a good choice for "one-shot" tasks, such as downloading a file, while a bound service would be suitable when frequent interactions with an Activity/Fragment is required.

While most services run in the background, there is a special sub-category known as a *foreground service*. This is a service that is given a higher priority (compared to a normal service) to perform some work for the user (such as playing music).

It is also possible to run a service in its own process on the same device, this is sometimes referred to as a *remote service* or as an *out-of-process service*. This does require more effort to create, but can be useful for when an application needs to share functionality with other applications, and can, in some cases, improve the user experience of an application.

Background Execution Limits in Android 8.0

Starting in Android 8.0 (API level 26), an Android application no longer have the ability to run freely in the background. When in the foreground, an app can start and run services without restriction. When an application moves into the background, Android will grant the app a certain amount of time to start and use services. Once that time has elapsed, the app can no longer start any services and any services that were started will be terminated. At this point it is not possible for the app to perform any work. Android considers an application to be in the foreground if one of the following conditions are met:

- There is a visible activity (either started or paused).
- The app has started a foreground service.
- Another app is in the foreground and is using components from an app that would be otherwise in the background. An example of this is if Application A, which is in the foreground, is bound to a service provided by Application B. Application B would then also be considered in the foreground, and not terminated by Android for being in the background.

There are some situations where, even though an app is in the background, Android will wake up the app and relax these restrictions for a few minutes, allowing the app to perform some work:

- A high priority Firebase Cloud Message is received by the app.
- The app receives a broadcast.
- The application receives and executes a `PendingIntent` in response to a Notification.

Existing Xamarin.Android applications may have to change how they perform background work to avoid any issues that might arise on Android 8.0. Here are some practical alternatives to an Android service:

- **Schedule work to run in the background using the Android Job Scheduler or the Firebase Job Dispatcher** – These two libraries provide a framework for applications to segregate background work in to *jobs*, a discrete unit of work. Apps can then schedule the job with the operating system along with some criteria about when the job can run.
- **Start the service in the foreground** – a foreground service is useful for when the app must perform some task in the background and the user may need to periodically interact with that task. The foreground service will display a persistent notification so that the user is aware that the app is running a background task and also provides a way to monitor or interact with the task. An example of this would be a podcasting app that is playing back a podcast to the user or perhaps downloading a podcast episode so that it can be enjoyed later.
- **Use a high priority Firebase Cloud Message (FCM)** – When Android receives a high priority FCM for an app, it will allow that app to run services in the background for a short period of time. This would be a good alternative to having a background service that polls an app in the background.
- **Defer work for when the app comes into the foreground** – If none of the previous solutions are viable, then apps must develop their own way to pause and resume work when the app comes to the foreground.

Related Links

- Android Oreo Background Execution Limits

Creating a Service

7/8/2021 • 4 minutes to read • Edit Online

Xamarin.Android services must obey two inviolable rules of Android services:

- They must extend the `Android.App.Service`.
- They must be decorated with the `Android.App.ServiceAttribute`.

Another requirement of Android services is that they must be registered in the **AndroidManifest.xml** and given a unique name. Xamarin.Android will automatically register the service in the manifest at build time with the necessary XML attribute.

This code snippet is the simplest example of creating a service in Xamarin.Android that meets these two requirements:

```
[Service]
public class DemoService : Service
{
    // Magical code that makes the service do wonderful things.
}
```

At compile time, Xamarin.Android will register the service by injecting the following XML element into **AndroidManifest.xml** (notice that Xamarin.Android generated a random name for the service):

```
<service android:name="md5a0cbbf8da641ae5a4c781aaf35e00a86.DemoService" />
```

It is possible to share a service with other Android applications by *exporting* it. This is accomplished by setting the `Exported` property on the `ServiceAttribute`. When exporting a service, the `ServiceAttribute.Name` property should also be set to provide a meaningful public name for the service. This snippet demonstrates how to export and name a service:

```
[Service(Exported=true, Name="com.xamarin.example.DemoService")]
public class DemoService : Service
{
    // Magical code that makes the service do wonderful things.
}
```

The **AndroidManifest.xml** element for this service will then look something like:

```
<service android:exported="true" android:name="com.xamarin.example.DemoService" />
```

Services have their own lifecycle with callback methods that are invoked as the service is created. Exactly which methods are invoked depends on the type of service. A started service must implement different lifecycle methods than a bound service, while a hybrid service must implement the callback methods for both a started service and a bound service. These methods are all members of the `Service` class; how the service is started will determine what lifecycle methods will be invoked. These lifecycle methods will be discussed in more detail later.

By default, a service will start in the same process as an Android application. It is possible to start a service in its own process by setting the `ServiceAttribute.IsolatedProcess` property to true:

```
[Service(IsolatedProcess=true)]
public class DemoService : Service
{
    // Magical code that makes the service do wonderful things, in it's own process!
}
```

The next step is to examine how to start a service and then move on to examine how to implement the three different types of services.

> **NOTE**
>
> A service runs on the UI thread, so if any work is to be performed which blocks the UI, the service must use threads to perform the work.

Starting A Service

The most basic way to start a service in Android is to dispatch an `Intent` which contains meta-data to help identify which service should be started. There are two different styles of Intents that can be used to start a service:

- **Explicit Intent** – An *explicit Intent* will identify exactly what service should be used to complete a given action. An explicit Intent can be thought of as a letter that has a specific address; Android will route the intent to the service that is explicitly identified. This snippet is one example of using an explicit Intent to start a service called `DownloadService`:

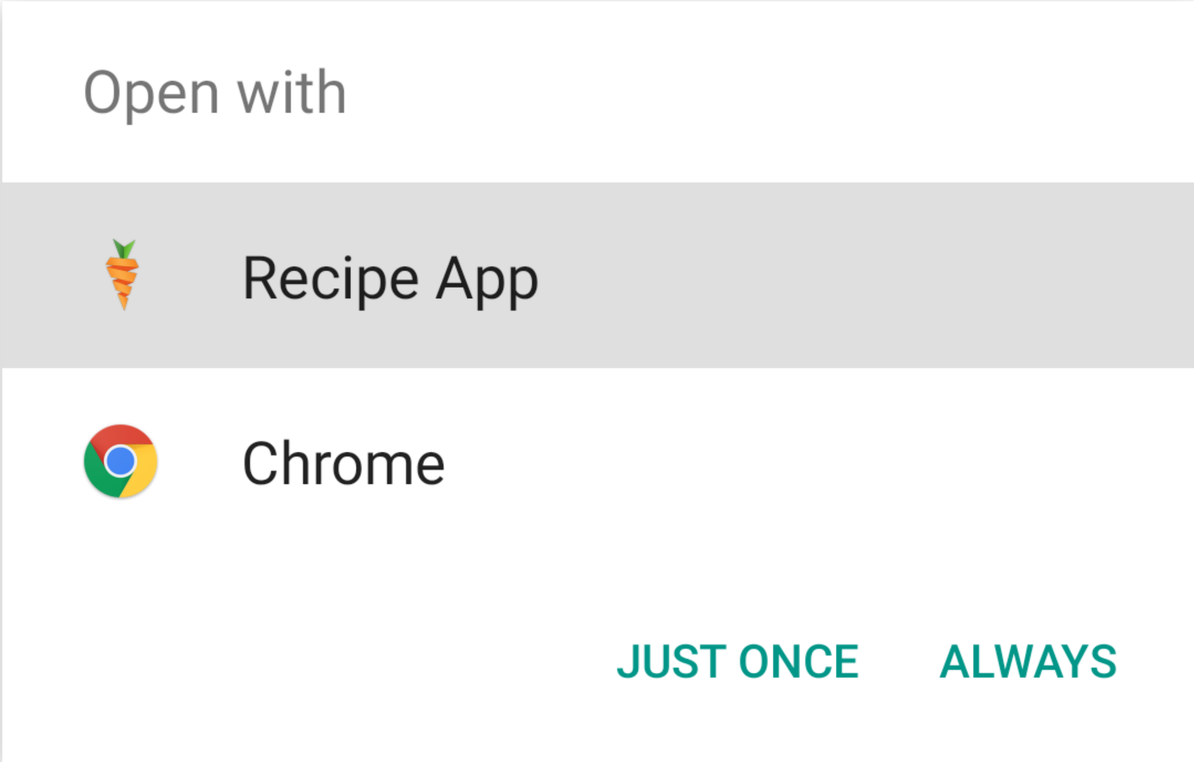
  ```
  // Example of creating an explicit Intent in an Android Activity
  Intent downloadIntent = new Intent(this, typeof(DownloadService));
  downloadIntent.data = Uri.Parse(fileToDownload);
  ```

- **Implicit Intent** – This type of Intent loosely identifies the type of action that the user wishes to perform, but the exact service to complete that action is unknown. An implicit Intent can be thought of as a letter that is addressed "To Whom It May Concern...". Android will examine the contents of the Intent, and determine if there is an existing service which matches the intent.

  An *intent filter* is used to help match the implicit intent with a registered service. An intent filter is an XML element that is added to **AndroidManifest.xml** which contains the necessary meta-data to help match a Service with an implicit intent.

  ```
  Intent sendIntent = new Intent("common.xamarin.DemoService");
  sendIntent.Data = Uri.Parse(fileToDownload);
  ```

If Android has more than one possible match for an implicit intent, then it may ask the user to select the component to handle the action:



IMPORTANT

Starting in Android 5.0 (AP level 21) an implicit intent cannot be used to start a service.

Where possible, applications should use explicit Intents to start a service. An implicit Intent does not ask for a specific service to start – it is a request for some service installed on the device to handle the request. This ambiguous request can result in the wrong service handling the request or another app needlessly starting (which increases the pressure for resources on the device).

How the Intent is dispatched depends on the type of service and will be discussed in more detail later in the guides specific to each type of service.

Creating an Intent Filter for Implicit Intents

To associate a service with an implicit Intent, an Android app must provide some meta-data to identify the capabilities of the service. This meta-data is provided by *intent filters*. Intent filters contain some information, such as an action or a type of data, that must be present in an Intent to start a service. In Xamarin.Android, the intent filter is registered in **AndroidManifest.xml** by decorating a service with the `IntentFilterAttribute`. For example, the following code adds an intent filter with an associated action of `com.xamarin.DemoService`:

```
[Service]
[IntentFilter(new String[]{"com.xamarin.DemoService"})]
public class DemoService : Service
{
}
```

This results in an entry being included in the **AndroidManifest.xml** file – an entry that is packaged with the application in a way analogous to the following example:

```
<service android:name="demoservice.DemoService">
    <intent-filter>
        <action android:name="com.xamarin.DemoService" />
    </intent-filter>
</service>
```

With the basics of a Xamarin.Android service out of the way, let's examine the different subtypes of services in more detail.

Related Links

- Android.App.Service
- Android.App.ServiceAttribute
- Android.App.Intent
- Android.App.IntentFilterAttribute

Bound Services in Xamarin.Android

7/8/2021 • 11 minutes to read • Edit Online

Bound services are Android services that provide a client-server interface that a client (such as an Android Activity) can interact with. This guide will discuss the key components involved with creating a bound service and how to use it in a Xamarin.Android application.

Bound Services Overview

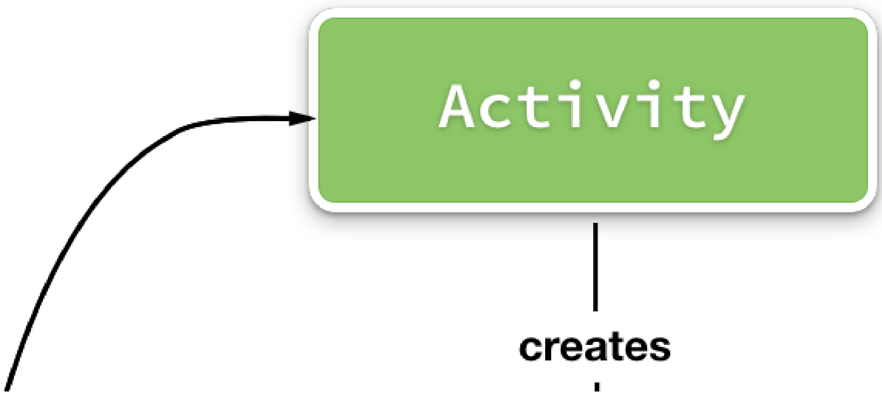
Services that provide a client-server interface for clients to directly interact with the service are referred to as *bound services*. There can be multiple clients connected to a single instance of a service at the same time. The bound service and the client are isolated from each other. Instead, Android provides a series of intermediate objects that manage the state of the connection between the two. This state is maintained by an object that implements the `Android.Content.IServiceConnection` interface. This object is created by the client and passed as a parameter to the `BindService` method. The `BindService` is available on any `Android.Content.Context` object (such as an Activity). It is a request to the Android operating system to start up the service and bind a client to it. There are three ways to a client may bind to a service using the `BindService` method:

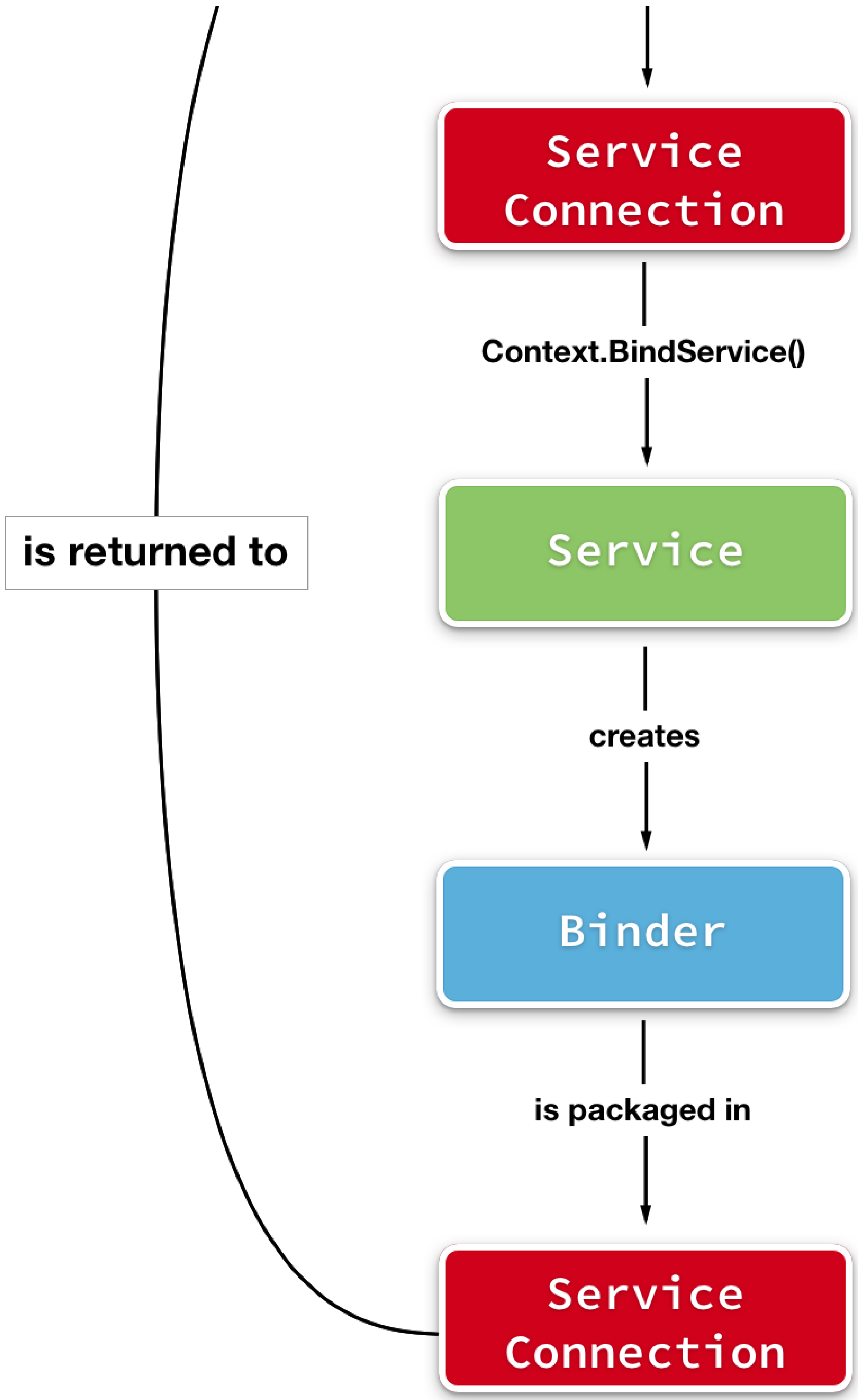
- **A service binder** – A service binder is a class that implements the `Android.OS.IBinder` interface. Most applications will not implement this interface directly, instead they will extend the `Android.OS.Binder` class. This is the most common approach and is suitable for when the service and the client exist in the same process.
- **Using a Messenger** – This technique is suitable for when the service might exist in a separate process. Instead, service requests are marshalled between the client and service via an `Android.OS.Messenger`. An `Android.OS.Handler` is created in the service which will handle the `Messenger` requests. This will be covered in another guide.
- **Using Android Interface Definition Language (AIDL)** – AIDL is an advanced technique that will not be covered in this guide.

Once a client has been bound to a service, communication between the two is occurs via `Android.OS.IBinder` object. This object is responsible for the interface that will allow the client to interact with the service. It is not necessary for each Xamarin.Android application to implement this interface from scratch, the Android SDK provides the `Android.OS.Binder` class which takes care of most of the code required with marshalling the object between the client and the service.

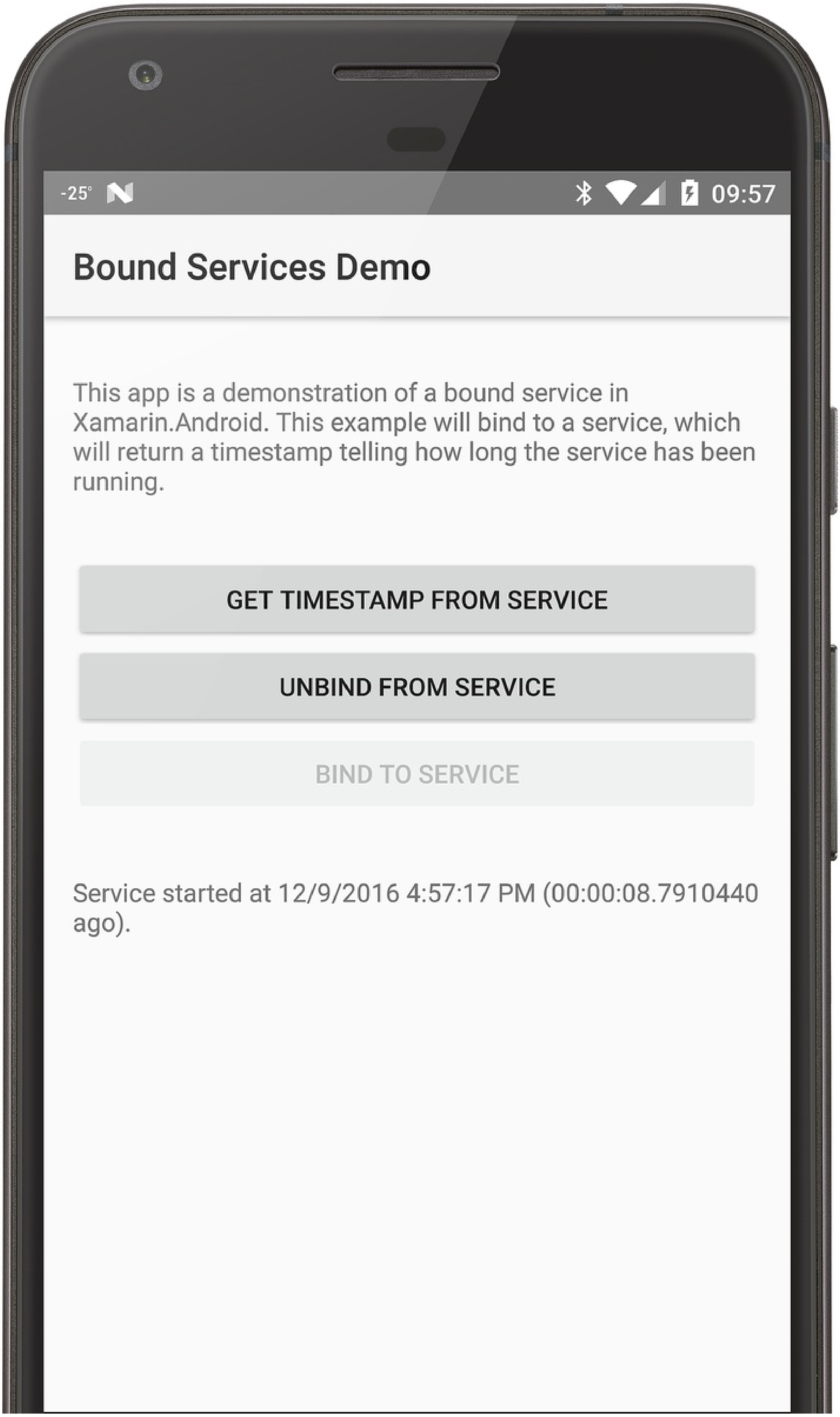
When a client is done with the service, it must unbind from it by calling the `UnbindService` method. Once the last client has unbound from a service, Android will stop and dispose of the bound service.

This diagram illustrates how the Activity, service connection, binder, and service all related to each other:





This guide will discuss how to extend the `Service` class to implement a bound service. It will also cover implementing `IServiceConnection` and extending `Binder` to allow a client to communicate with the service. A sample app accompanies this guide, which contain a solution with a single Xamarin.Android project called BoundServiceDemo . This is a very basic application which demonstrates how to implement a service and how to bind an activity to it. The bound service has a very simple API with only one method, `GetFormattedTimestamp` , which returns a string that tells the user when the service has started and how long it has been running. The app also allows the user to manually unbind and bind to the service.



Implementing and Consuming a Bound Service

There are three components that must be implemented in order for an Android application to consume a bound service:

1. **Extend the `Service` class and Implement the Lifecycle Callback Methods** – This class will contain the code that will perform the work that will be requested of the service. This will be covered in more detail below.
2. **Create a Class that Implements `IServiceConnection`** – This interface provides callback methods will invoked by Android to notify the client when the connection to the service has changed, i.e. the client has connected or disconnected to the service. The service connection will also provide a reference to an object that the client can use to directly interact with the service. This reference is known as the *binder*.
3. **Create a Class that Implements `IBinder`** – A *Binder* implementation provides the API that a client uses to communicate with the service. The Binder can either provide a reference to the bound service, allowing methods to be directly invoked or the Binder can provide a client API that encapsulates and hides the bound service from the application. An `IBinder` must provide the necessary code for remote procedure calls. It is not necessary (or recommended) to implement the `IBinder` interface directly. Instead applications should extend the `Binder` type which provides most of the base functionality required by an `IBinder`.
4. **Starting and Binding to a Service** – Once the service connection, binder, and service have been created the Android application is responsible for starting the service and binding to it.

Each of these steps will be discussed in the following sections in more detail.

Extend the `Service` Class

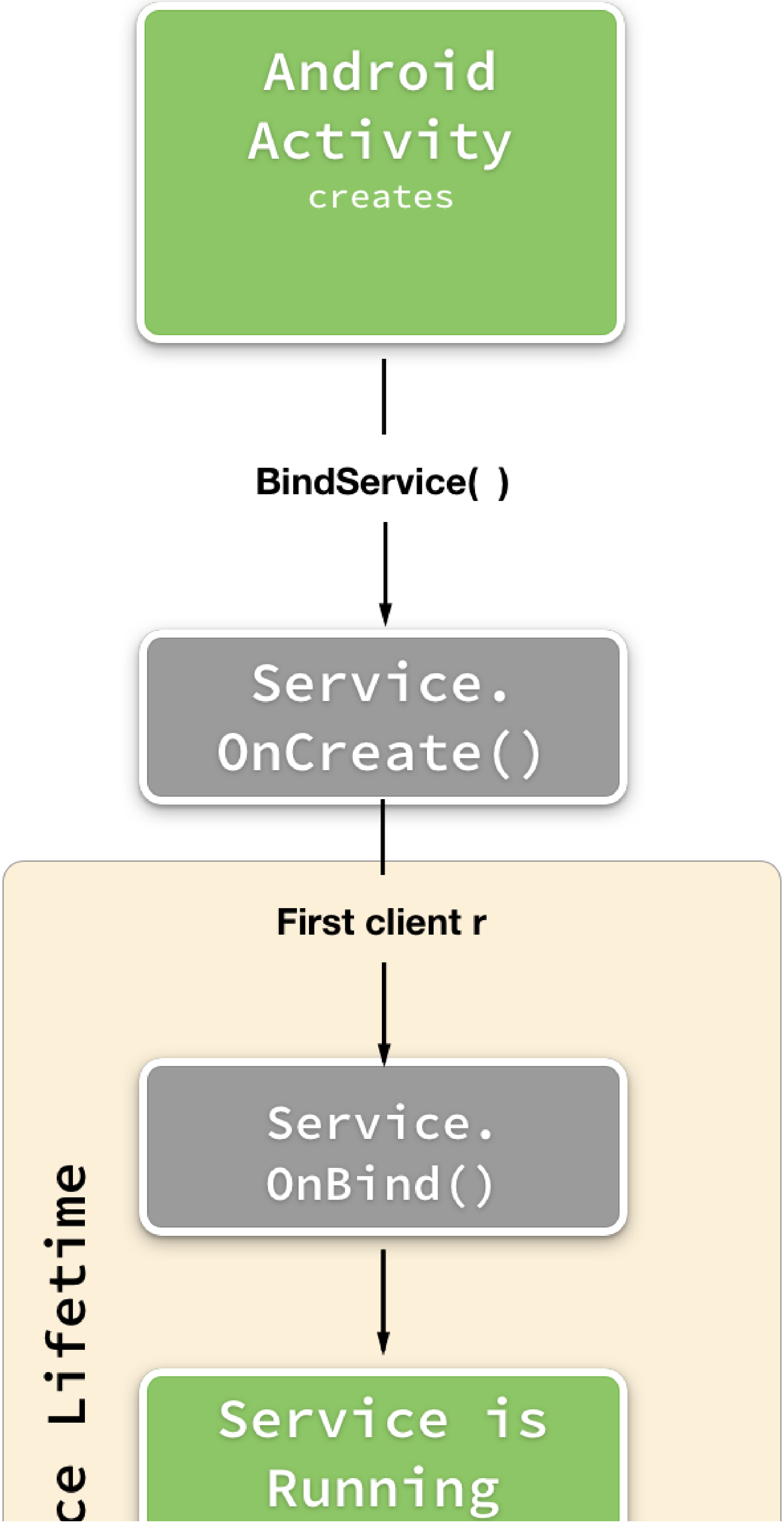
To create a service using Xamarin.Android, it is necessary to subclass `Service` and to adorn the class with the `ServiceAttribute`. The attribute is used by the Xamarin.Android build tools to properly register the service in the app's **AndroidManifest.xml** file Much like an activity, a bound service has it's own lifecycle and callback methods associated with the significant events in it's lifecycle. The following list is an example of some of the more common callback methods that a service will implement:

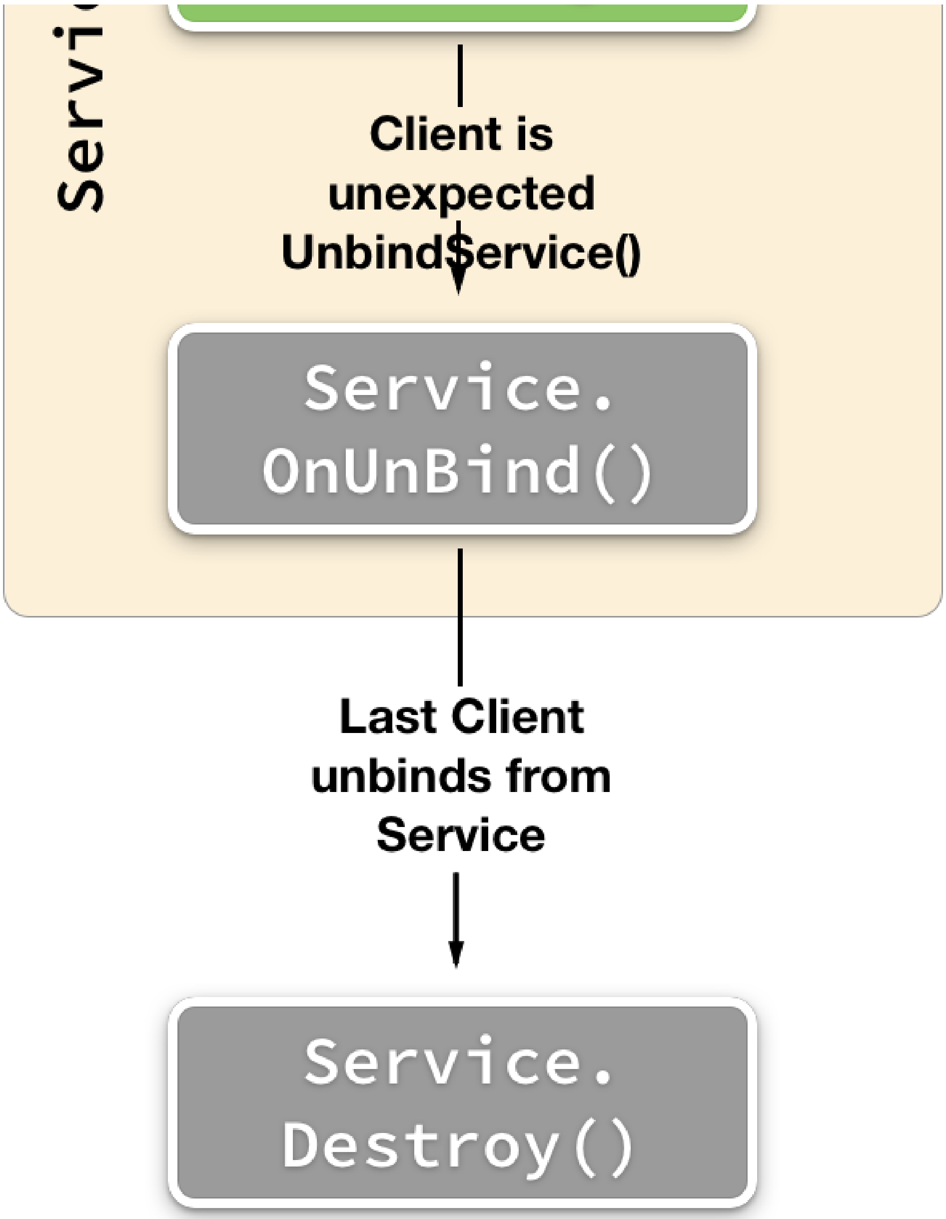
- `OnCreate` – This method is invoked by Android as it is instantiating the service. It is used to initialize any variables or objects that are required by the service during it's lifetime. This method is optional.
- `OnBind` – This method must be implemented by all bound services. It is invoked when the first client tries to connect to the service. It will return an instance of `IBinder` so that the client may interact with the service. As long as the service is running, the `IBinder` object will be used to fulfill any future client requests to bind to the service.
- `OnUnbind` – This method is called when all bound clients have unbound. By returning `true` from this method, the service will later call `OnRebind` with the intent passed to `OnUnbind` when new clients bind to it. You would do this when a service continues running after it has been unbound. This would happen if the recently unbound service were also a started service, and `StopService` or `StopSelf` hadn't been called. In such a scenario, `OnRebind` allows the intent to be retrieved. The default returns `false`, which does nothing.

Optional.

- `OnDestroy` – This method is called when Android is destroying the service. Any necessary cleanup, such as releasing resources, should be performed in this method. Optional.

The key lifecycle events of a bound service are shown in this diagram:





The following code snippet, from the companion application that accompanies this guide, shows how to implement a bound service in Xamarin.Android:

```
using Android.App;
using Android.Util;
using Android.Content;
using Android.OS;

namespace BoundServiceDemo
{
    [Service(Name="com.xamarin.ServicesDemo1")]
    public class TimestampService : Service, IGetTimestamp
    {
        static readonly string TAG = typeof(TimestampService).FullName;
        IGetTimestamp timestamper;

        public IBinder Binder { get; private set; }

        public override void OnCreate()
        {
            // This method is optional to implement
            base.OnCreate();
            Log.Debug(TAG, "OnCreate");
            timestamper = new UtcTimestamper();
        }

        public override IBinder OnBind(Intent intent)
        {
            // This method must always be implemented
            Log.Debug(TAG, "OnBind");
            this.Binder = new TimestampBinder(this);
            return this.Binder;
        }

        public override bool OnUnbind(Intent intent)
        {
            // This method is optional to implement
            Log.Debug(TAG, "OnUnbind");
            return base.OnUnbind(intent);
        }

        public override void OnDestroy()
        {
            // This method is optional to implement
            Log.Debug(TAG, "OnDestroy");
            Binder = null;
            timestamper = null;
            base.OnDestroy();
        }

        /// <summary>
        /// This method will return a formatted timestamp to the client.
        /// </summary>
        /// <returns>A string that details what time the service started and how long it has been running.
</returns>
        public string GetFormattedTimestamp()
        {
            return timestamper?.GetFormattedTimestamp();
        }
    }
}
```

In the sample, the `OnCreate` method initializes an object that holds the logic for retrieving and formatting a timestamp that would be requested by a client. When the first client tries to bind to the service, Android will invoke the `OnBind` method. This service will instantiate a `TimestampBinder` object that will allow the clients to access this instance of the running service. The `TimestampBinder` class is discussed in the next section.

Implementing IBinder

As mentioned, an `IBinder` object provides the communication channel between a client and a service. Android applications should not implement the `IBinder` interface, the `Android.OS.Binder` should be extended. The `Binder` class provides much of the necessary infrastructure which is necessary marshal the binder object from the service (which may be running in a separate process) to the client. In most cases, the `Binder` subclass is only a few lines of code and wraps a reference to the service. In this example, `TimestampBinder` has a property that exposes the `TimestampService` to the client:

```
public class TimestampBinder : Binder
{
    public TimestampBinder(TimestampService service)
    {
        this.Service = service;
    }

    public TimestampService Service { get; private set; }
}
```

This `Binder` makes it possible to invoke the public methods on the service; for example:

```
string currentTimestamp = serviceConnection.Binder.Service.GetFormattedTimestamp()
```

Once `Binder` has been extended, it is necessary to implement `IServiceConnection` to connect everything together.

Creating the Service Connection

The `IServiceConnection` will present|introduce|expose|connect the `Binder` object to the client. In addition to implementing the `IServiceConnection` interface, the class must extend `Java.Lang.Object`. The service connection should also provide some way that the client can access the `Binder` (and therefore communicate with the bound service).

This code is from the accompanying sample project is one possible way to implement `IServiceConnection`:

```
using Android.Util;
using Android.OS;
using Android.Content;

namespace BoundServiceDemo
{
    public class TimestampServiceConnection : Java.Lang.Object, IServiceConnection, IGetTimestamp
    {
        static readonly string TAG = typeof(TimestampServiceConnection).FullName;

        MainActivity mainActivity;
        public TimestampServiceConnection(MainActivity activity)
        {
            IsConnected = false;
            Binder = null;
            mainActivity = activity;
        }

        public bool IsConnected { get; private set; }
        public TimestampBinder Binder { get; private set; }

        public void OnServiceConnected(ComponentName name, IBinder service)
        {
            Binder = service as TimestampBinder;
            IsConnected = this.Binder != null;

            string message = "onServiceConnected - ";
            Log.Debug(TAG, $"OnServiceConnected {name.ClassName}");

            if (IsConnected)
            {
                message = message + " bound to service " + name.ClassName;
                mainActivity.UpdateUiForBoundService();
            }
            else
            {
                message = message + " not bound to service " + name.ClassName;
                mainActivity.UpdateUiForUnboundService();
            }

            Log.Info(TAG, message);
            mainActivity.timestampMessageTextView.Text = message;

        }

        public void OnServiceDisconnected(ComponentName name)
        {
            Log.Debug(TAG, $"OnServiceDisconnected {name.ClassName}");
            IsConnected = false;
            Binder = null;
            mainActivity.UpdateUiForUnboundService();
        }

        public string GetFormattedTimestamp()
        {
            if (!IsConnected)
            {
                return null;
            }

            return Binder?.GetFormattedTimestamp();
        }
    }
}
```

As a part of the binding process, Android will invoke the `OnServiceConnected` method, providing the `name` of the service that is being bound and the `binder` that holds a reference to the service itself. In this example, the service connection has two properties, one that holds a reference to the Binder and a boolean flag for if the client is connected to the service or not.

The `OnServiceDisconnected` method is only invoked when the connection between a client and a service is unexpectedly lost or broken. This method allows the client a chance to respond to the interruption in service.

Starting and Binding to a Service with an Explicit Intent

To use a bound service, a client (such as an Activity) must instantiate an object that implements `Android.Content.IServiceConnection` and invoke the `BindService` method. `BindService` will return `true` if the service is bound to, `false` if it is not. The `BindService` method takes three parameters:

- **An** `Intent` – The Intent should explicitly identify which service to connect to.
- **An** `IServiceConnection` **Object** – This object is an intermediary that provides callback methods to notify the client when the bound service is started and stopped.
- `Android.Content.Bind` **enum** – This parameter is a set of flags are used by the system to when bind the object. The most commonly used value is `Bind.AutoCreate`, which will automatically start the service if it isn't already running.

The following Code snippet is an example of how to start a bound service in an Activity using an explicit intent:

```
protected override void OnStart ()
{
    base.OnStart ();

    if (serviceConnection == null)
    {
        this.serviceConnection = new TimestampServiceConnection(this);
    }

    Intent serviceToStart = new Intent(this, typeof(TimestampService));
    BindService(serviceToStart, this.serviceConnection, Bind.AutoCreate);

}
```

IMPORTANT

Starting with Android 5.0 (API level 21) it is only possible to bind to a service with an explicit intent.

Architectural Notes About the Service Connection and the Binder.

Some OOP purists may disapprove of the previous implementation of the `TimestampBinder` class as it is a violation of the Law of Demeter which, in it's simplest form states "Don't talk to strangers; only talk to your friends". This particular implementation exposes the concrete `TimestampService` class to all clients.

Strictly speaking, it is not necessary for the client to know about the `TimestampService` and exposing that concrete class to clients can make an application more brittle and harder to maintain over it's lifetime. An alternate approach is to use an interface which exposes the `GetFormattedTimestamp()` method, and proxy calls to the service through the `Binder` (or possible the service connection class):

```
public class TimestampBinder : Binder, IGetTimestamp
{
    TimestampService service;
    public TimestampBinder(TimestampService service)
    {
        this.service = service;
    }

    public string GetFormattedTimestamp()
    {
        return service?.GetFormattedTimestamp();
    }
}
```

This particular example allow an activity to invoke methods on the service itself:

```
// In this example the Activity is only talking to a friend, i.e. the IGetTimestamp interface provided by
the Binder.
string currentTimestamp = serviceConnection.Binder.GetFormattedTimestamp()
```

Related Links

- Android.App.Service
- Android.Content.Bind
- Android.Content.Context
- Android.Content.IServiceConnection
- Android.OS.Binder
- Android.OS.IBinder
- BoundServiceDemo (sample)

Intent Services in Xamarin.Android

3/19/2020 • 2 minutes to read • Edit Online

Both started and bound services run on the main thread, which means that to keep performance smooth, a service needs to perform the work asynchronously. One of the simplest ways to address this concern is with a *worker queue processor pattern*, where the work to be done is placed in a queue that is serviced by a single thread.

The `IntentService` is a subclass of the `Service` class that provides an Android specific implementation of this pattern. It will manage queueing work, starting up a worker thread to service the queue, and pulling requests off the queue to be run on the worker thread. An `IntentService` will quietly stop itself and remove the worker thread when there is no more work in the queue.

Work is submitted to the queue by creating an `Intent` and then passing that `Intent` to the `StartService` method.

It is not possible to stop or interrupt the `OnHandleIntent` method `IntentService` while it is working. Because of this design, an `IntentService` should be kept stateless – it should not rely on an active connection or communication from the rest of the application. An `IntentService` is meant to statelessly process the work requests.

There are two requirements for subclassing `IntentService`:

1. The new type (created by subclassing `IntentService`) only overrides the `OnHandleIntent` method.
2. The constructor for the new type requires a string which is used to name the worker thread that will handle the requests. The name of this worker thread is primarily used when debugging the application.

The following code shows an `IntentService` implementation with the overridden `OnHandleIntent` method:

```
[Service]
public class DemoIntentService: IntentService
{
    public DemoIntentService () : base("DemoIntentService")
    {
    }

    protected override void OnHandleIntent (Android.Content.Intent intent)
    {
        Console.WriteLine ("perform some long running work");
        ...
        Console.WriteLine ("work complete");
    }
}
```

Work is sent to an `IntentService` by instantiating an `Intent` and then calling the `StartService` method with that Intent as a parameter. The Intent will be passed to the service as a parameter in the `OnHandleIntent` method. This code snippet is an example of sending a work request to an Intent:

```
// This code might be called from within an Activity, for example in an event
// handler for a button click.
Intent downloadIntent = new Intent(this, typeof(DemoIntentService));

// This is just one example of passing some values to an IntentService via the Intent:
downloadIntent.PutExtra("file_to_download", "http://www.somewhere.com/file/to/download.zip");

StartService(downloadIntent);
```

The `IntentService` can extract the values from the Intent, as demonstrated in this code snippet:

```
protected override void OnHandleIntent (Android.Content.Intent intent)
{
    string fileToDownload = intent.GetStringExtra("file_to_download");

    Log.Debug("DemoIntentService", $"File to download: {fileToDownload}.");
}
```

Related Links

- IntentService
- StartService

Started Services with Xamarin.Android

7/8/2021 • 5 minutes to read • Edit Online

Started Services Overview

Started services typically perform a unit of work without providing any direct feedback or results to the client. An example of a unit of work is a service that uploads a file to a server. The client will make a request to a service to upload a file from the device to a website. The service will quietly upload the file (even if the app has no Activities in the foreground), and terminate itself when the upload is finished. It is important to realize that a started service will run on the UI thread of an application. This means that if a service is to perform work that will block the UI thread, it must create and dispose of threads as necessary.

Unlike a bound service, there is no communication channel between a "pure" started service and its clients. This means that a started service will implement some different lifecycle methods than a bound service. The following list highlights the common lifecycle methods in a started service:

- `OnCreate` – Called one time when the service is first started. This is where initialization code should be implemented.
- `OnBind` – This method must be implemented by all service classes, however a started service does not typically have a client bound to it. Because of this, a started service just returns `null`. In contrast, a hybrid service (which is the combination of a bound service and a started service) has to implement and return a `Binder` for the client.
- `OnStartCommand` – Called for each request to start the service, either in response to a call to `StartService` or a restart by the system. This is where the service can begin any long-running task. The method returns a `StartCommandResult` value that indicates how or if the system should handle restarting the service after a shutdown due to low memory. This call takes place on the main thread. This method is described in more detail below.
- `OnDestroy` – This method is called when the Service is being destroyed. It is used to perform any final clean up required.

The important method for a started service is the `OnStartCommand` method. It will be invoked each time the service receives a request to do some work. The following code snippet is an example of `OnStartCommand`:

```
public override StartCommandResult OnStartCommand (Android.Content.Intent intent, StartCommandFlags flags,
int startId)
{
    // This method executes on the main thread of the application.
    Log.Debug ("DemoService", "DemoService started");
    ...
    return StartCommandResult.Sticky;
}
```

The first parameter is an `Intent` object containing the meta-data about the work to perform. The second parameter contains a `StartCommandFlags` value that provides some information about the method call. This parameter has one of two possible values:

- `StartCommandFlag.Redelivery` – This means that the `Intent` is a re-delivery of a previous `Intent`. This value is provided when the service had returned `StartCommandResult.RedeliverIntent` but was stopped before it could be properly shut down. - `StartCommandFlag.Retry` I This value is received when a previous `OnStartCommand` call failed and Android is trying to start the service again with the same intent as the previous failed attempt.

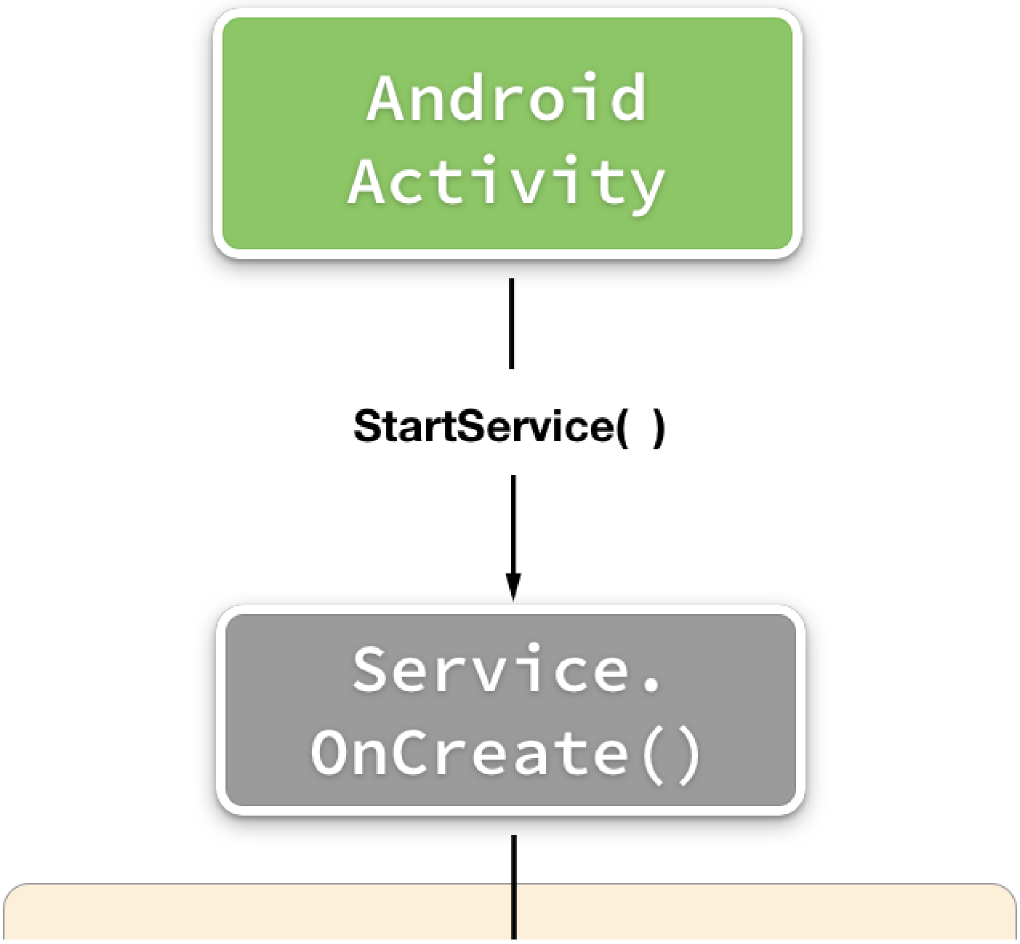
Finally, the third parameter is an integer value that is unique to the application that identifies the request. It is possible that multiple callers may invoke the same service object. This value is used to associate a request to stop a service with a given request to start a service. It will be discussed in more detail in the section Stopping the Service.

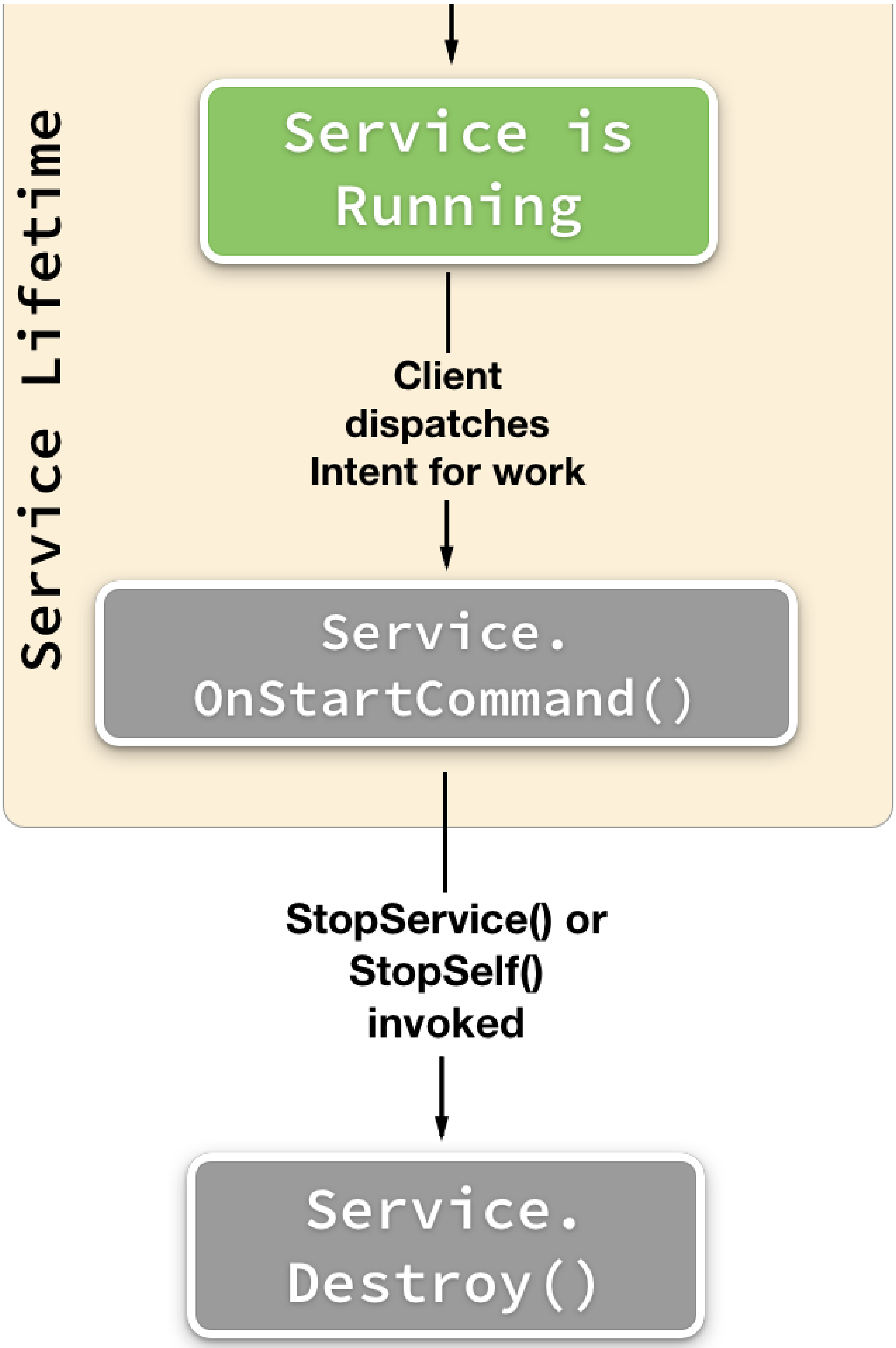
The value `StartCommandResult` is returned by the service as a suggestion to Android on what to do if the service is killed due to resource constraints. There are three possible values for `StartCommandResult`:

- StartCommandResult.NotSticky – This value tells Android that it is not necessary to restart the service that it has killed. As an example of this, consider a service that generates thumbnails for a gallery in an app. If the service is killed, it isn't crucial to recreate the thumbnail immediately – the thumbnail can be recreated the next time the app is run.
- StartCommandResult.Sticky – This tells Android to restart the Service, but not to deliver the last Intent that started the Service. If there are no pending Intents to handle, then a `null` will be provided for the Intent parameter. An example of this might be a music player app; the service will restart ready to play music, but it will play the last song.
- StartCommandResult.RedeliverIntent – This value is will tell Android to restart the service and re-deliver the last `Intent`. An example of this is a service that downloads a data file for an app. If the service is killed, the data file still needs to be downloaded. By returning `StartCommandResult.RedeliverIntent`, when Android restarts the service it will also provide the Intent (which holds the URL of the file to download) to the service. This will enable the download to either restart or resume (depending on the exact implementation of the code).

There is a fourth value for `StartCommandResult` – `StartCommandResult.ContinuationMask`. This value is returned by `OnStartCommand` and it describes how Android will continue the service it has killed. This value isn't typically used to start a service.

The key lifecycle events of a started service are shown in this diagram:





Stopping the Service

A started service will keep running indefinitely; Android will keep the service running as long as there are sufficient system resources. Either the client must stop the service, or the service may stop itself when it is done its work. There are two ways to stop a service:

- [Android.Content.Context.StopService()]() – A client (such as an Activity) can request a service stop by calling the `StopService` method:

```
StopService(new Intent(this, typeof(DemoService));
```

- Android.App.Service.StopSelf() – A service may shut itself down by invoking the `StopSelf`:

```
StopSelf();
```

Using startId to Stop a Service

Multiple callers can request that a service be started. If there is an outstanding start request, the service can use the `startId` that is passed into `OnStartCommand` to prevent the service from being stopped prematurely. The `startId` will correspond to the latest call to `StartService`, and will be incremented each time it is called. Therefore, if a subsequent request to `StartService` has not yet resulted in a call to `OnStartCommand`, the service can call `StopSelfResult`, passing it the latest value of `startId` it has received (instead of simply calling `StopSelf`). If a call to `StartService` has not yet resulted in a corresponding call to `OnStartCommand`, the system will not stop the service, because the `startId` used in the `StopSelf` call will not correspond to the latest `StartService` call.

Related Links

- StartedServicesDemo (sample)
- Android.App.Service
- Android.App.StartCommandFlags
- Android.App.StartCommandResult
- Android.Content.BroadcastReceiver
- Android.Content.Intent
- Android.OS.Handler
- Android.Widget.Toast
- Status Bar Icons

Foreground Services

7/8/2021 • 3 minutes to read • Edit Online

A foreground service is a special type of a bound service or a started service. Occasionally services will perform tasks that users must be actively aware of, these services are known as *foreground services*. An example of a foreground service is an app that is providing the user with directions while driving or walking. Even if the app is in the background, it is still important that the service has sufficient resources to work properly and that the user has a quick and handy way to access the app. For an Android app, this means that a foreground service should receive higher priority than a "regular" service and a foreground service must provide a `Notification` that Android will display as long as the service is running.

To start a foreground service, the app must dispatch an Intent that tells Android to start the service. Then the service must register itself as a foreground service with Android. Apps that are running on Android 8.0 (or higher) should use the `Context.StartForegroundService` method to start the service, while apps that are running on devices with an older version of Android should use `Context.StartService`

This C# extension method is an example of how to start a foreground service. On Android 8.0 and higher it will use the `StartForegroundService` method, otherwise the older `StartService` method will be used.

```
public static void StartForegroundServiceCompat<T>(this Context context, Bundle args = null) where T :
Service
{
    var intent = new Intent(context, typeof(T));
    if (args != null)
    {
        intent.PutExtras(args);
    }

    if (Android.OS.Build.VERSION.SdkInt >= Android.OS.BuildVersionCodes.O)
    {
        context.StartForegroundService(intent);
    }
    else
    {
        context.StartService(intent);
    }
}
```

Registering as a Foreground Service

Once a foreground service has started, it must register itself with Android by invoking the `StartForeground`. If the service is started with the `Service.StartForegroundService` method but does not register itself, then Android will stop the service and flag the app as non-responsive.

`StartForeground` takes two parameters, both of which are mandatory:

- An integer value that is unique within the application to identify the service.
- A `Notification` object that Android will display in the status bar for as long as the service is running.

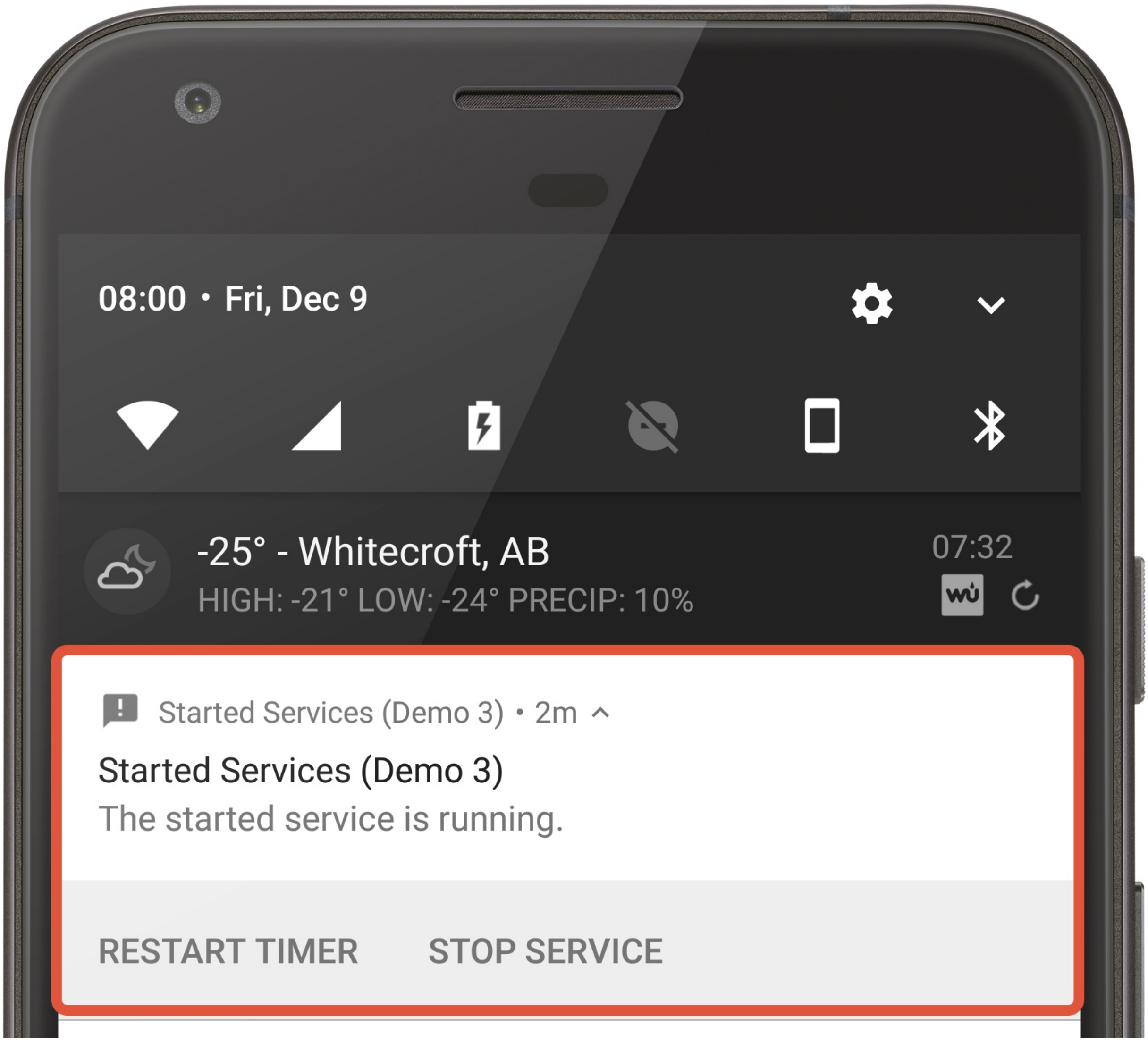
Android will display the notification in the status bar for as long as the service is running. The notification, at minimum, will provide a visual cue to the user that the service is running. Ideally, the notification should provide the user with a shortcut to the application or possibly some action buttons to control the application. An example of this is a music player – the notification that is displayed may have buttons to pause/play music, to rewind to the previous song, or to skip to the next song.

This code snippet is an example of registering a service as a foreground service:

```
// This is any integer value unique to the application.
public const int SERVICE_RUNNING_NOTIFICATION_ID = 10000;

public override StartCommandResult OnStartCommand(Intent intent, StartCommandFlags flags, int startId)
{
    // Code not directly related to publishing the notification has been omitted for clarity.
    // Normally, this method would hold the code to be run when the service is started.

    var notification = new Notification.Builder(this)
        .SetContentTitle(Resources.GetString(Resource.String.app_name))
        .SetContentText(Resources.GetString(Resource.String.notification_text))
        .SetSmallIcon(Resource.Drawable.ic_stat_name)
        .SetContentIntent(BuildIntentToShowMainActivity())
        .SetOngoing(true)
        .AddAction(BuildRestartTimerAction())
        .AddAction(BuildStopServiceAction())
        .Build();

    // Enlist this instance of the service as a foreground service
    StartForeground(SERVICE_RUNNING_NOTIFICATION_ID, notification);
}
```

The previous notification will display a status bar notification that is similar to the following:

This screenshot shows the expanded notification in the notification tray with two actions that allow the user to control the service:



More information about notifications is available in the Local Notifications section of the Android Notifications guide.

Unregistering as a Foreground Service

A service can de-list itself as a foreground service by calling the method `StopForeground`. `StopForeground` will not stop the service, but it will remove the notification icon and signals Android that this service can be shut down if necessary.

The status bar notification that is displayed can also be removed by passing `true` to the method:

```
StopForeground(true);
```

If the service is halted with a call to `StopSelf` or `StopService`, the status bar notification will be removed.

Related Links

- Android.App.Service
- Android.App.Service.StartForeground
- Local Notifications
- ForegroundServiceDemo (sample)

Running Android Services in Remote Processes

7/8/2021 • 26 minutes to read • Edit Online

Generally, all components in an Android application will run in the same process. Android services are a notable exception to this in that they can be configured to run in their own processes and shared with other applications, including those from other Android developers. This guide will discuss how to create and use an Android remote service using Xamarin.

Out of Process Services Overview

When an application starts up, Android creates a process in which to run the application. Typically, all the components the application will run in this one process. Android services are a notable exception to this in that they can be configured to run in their own processes and shared with other applications, including those from other Android developers. These types of services are referred to as *remote services* or *out-of-process services*. The code for these services will be contained in the same APK as the main application; however, when the service is started Android will create a new process for just that service. In contrast, a service that runs in the same process as the rest of the application is sometimes referred to as a *local service*.

In general, it is not necessary for an application to implement a remote service. A local service is sufficient (and desirable) for an app's requirements in many cases. An out-of-process has it's own memory space which must be managed by Android. Although this does introduce more overhead to the overall application, there are some scenarios where it can be advantageous to run a service in its own process:

1. **Sharing Functionality** – Some application developers may have multiple apps and functionality that is shared between all the applications. Packaging that functionality in an Android service which runs in its own process may simplify application maintenance. It is also possible to package the service in its own stand-alone APK and deploy it separately from the rest of the application.

2. **Improving the User Experience** – There are two possible ways that an out-of-process service can improve the user experience of the application. The first way deals with memory management. When a garbage collection (GC) cycle occurs, Android will pause all activity in the process until the GC is complete. The user might perceive this pause as a "stutter" or "jank". When a service is running in it's own process, it is the service process that is paused, not the application process. This pause will be much less noticeable to the user as the application process (and therefore the user interface) is not paused.

   Secondly, if the memory requirements of a process becomes too large, Android may kill that process to free up resources for the device. If a service does have a large memory footprint and it is running in the same process as the UI, then when Android forcibly reclaims those resources the UI will be shut down, forcing the user to start the app. However, if a service, running in its own process is shut down by Android, the UI process remains unaffected. The UI can bind (and restart) the service, transparent to the user, and resume normal functioning.

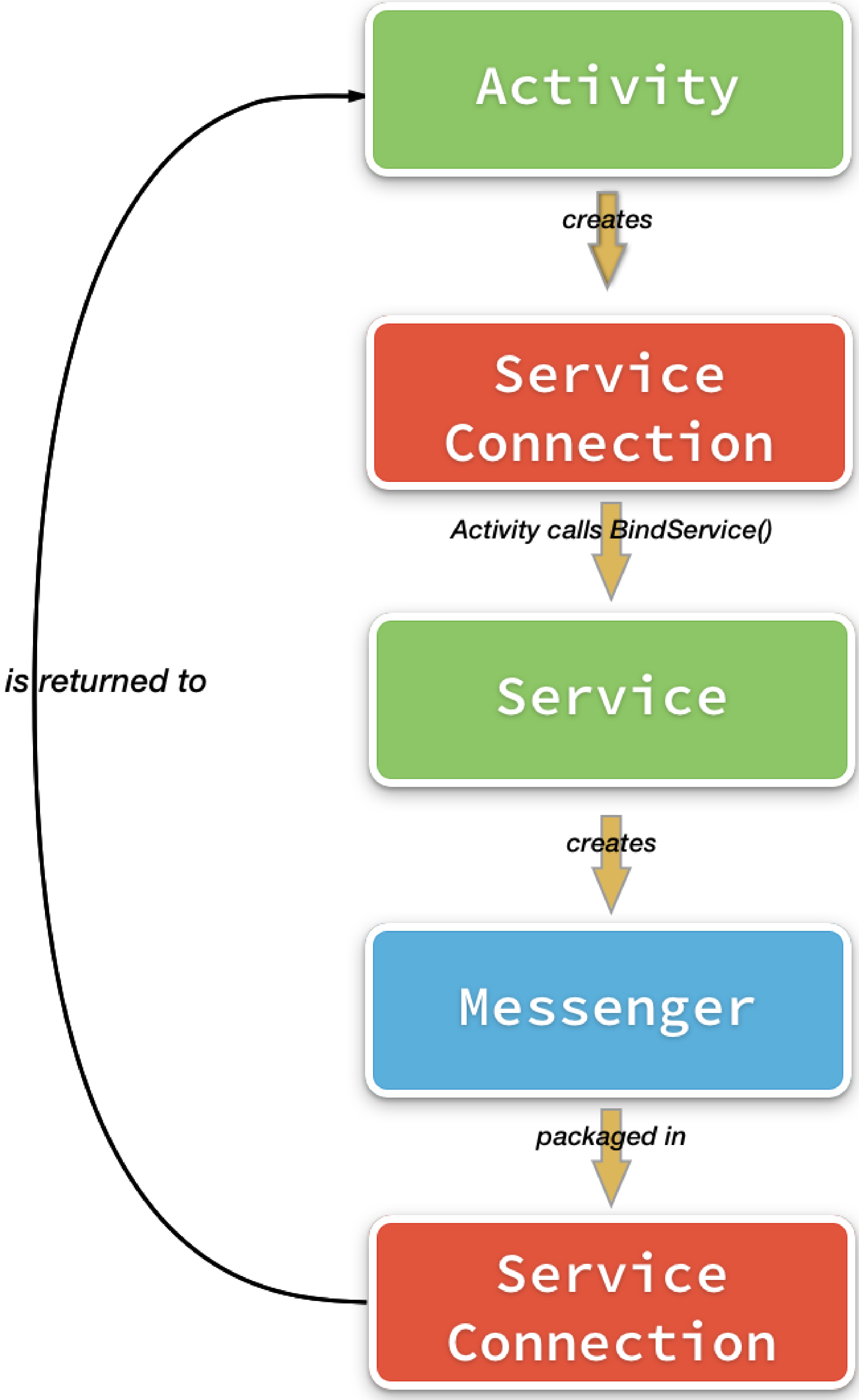
3. **Improving Application Performance** – The UI process may be terminated or shut down independent of the service process. By moving lengthy startup tasks to an out-of-process service, it is possible that the startup time of the UI maybe improved (assuming that the service process is kept alive in between the times that UI is launched).

In many ways, binding to a service running in another process is the same as binding to a local service. The client will invoke `BindService` to bind (and start, if necessary) the service. An `Android.OS.IServiceConnection` object will be created to manage the connection between the client and the service. If the client successfully binds to the service, then Android will return an object via the `IServiceConnection` that can be used to invoke methods on the service. The client then interacts with the service using this object. To review, here are the steps to bind to a service:

- **Create an Intent** – An explicit intent must be used to binding to the service.
- **Implement and Instantiate an `IServiceConnection` object** – The `IServiceConnection` object acts as an intermediary between the client and the service. It is responsible for monitoring the connection between client and server.
- **Invoke the `BindService` method** – Calling `BindService` will dispatch the intent and the service connection created in the previous steps to Android, which will take care of starting the service and establishing communication between client and service.

The need to cross process boundaries does introduce extra complexity: the communication is one-way (client to server) and the client can't directly invoke methods on the service class. Recall that when a service is running the same process as the client, Android provides an `IBinder` object which may allow for two-way communication. This is not the case with service running in its own process. A client communicates with a remote service with the help of the `Android.OS.Messenger` class.

When a client requests to bind with the remote service, Android will invoke the `Service.OnBind` lifecycle method, which will return the internal `IBinder` object that is encapsulated by the `Messenger`. The `Messenger` is a thin wrapper over a special `IBinder` implementation that is provided by the Android SDK. The `Messenger` takes care of the communication between the two different processes. The developer is unconcerned with the details of serializing a message, marshalling the message across the process boundary, and then deserializing it on the client. This work is handled by the `Messenger` object. This diagram shows the client-side Android components that are involved when a client initiates binding to an out-of-process service:

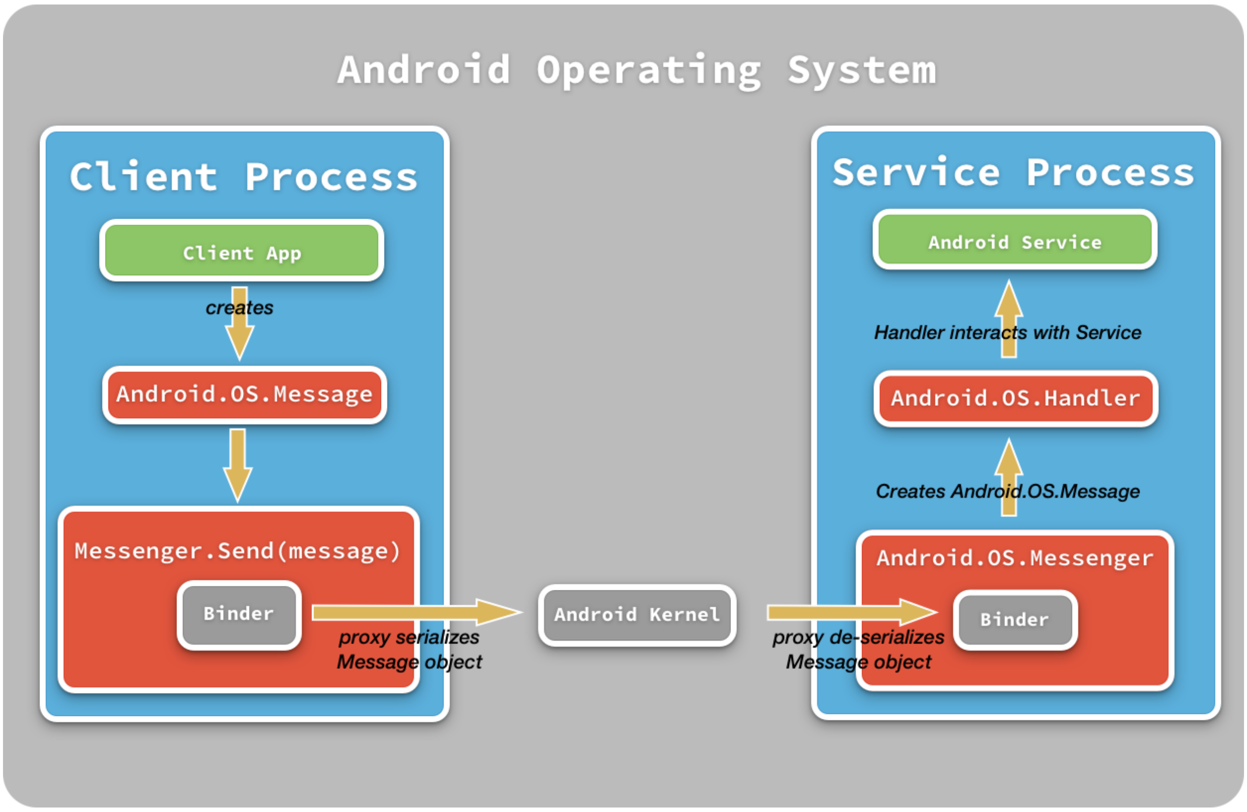


The `Service` class in the remote process will go through the same lifecycle callbacks that a bound service in a local process will go through, and many of the APIs involved are the same. `Service.OnCreate` is used to initialize a `Handler` and inject that into `Messenger` object. Likewise, `OnBind` is overridden, but instead of returning an `IBinder` object, the service will return the `Messenger`. This diagram illustrates what happens in the service when a client is binding to it:

When a `Message` is received by a service, it is processed by in instance of `Android.OS.Handler`. The service will implement its own `Handler` subclass that must override the `HandleMessage` method. This method is invoked by the `Messenger` and receives the `Message` as a parameter. The `Handler` will inspect the `Message` meta-data and use that information to invoke methods on the service.

One-way communication occurs when a client creates a `Message` object and dispatches it to the service using the `Messenger.Send` method. `Messenger.Send` will serialize the `Message` and hand that serialized data off to

Android, which will route the message across the process boundary and to the service. The `Messenger` that is hosted by the service will create a `Message` object from the incoming data. This `Message` is placed into a queue, where messages are submitted one at a time to the `Handler`. The `Handler` will inspect the meta-data contained in the `Message` and invoke the appropriate methods on the `Service`. The following diagram illustrates these various concepts in action:



This guide will discuss the details of implementing an out-of-process service. It will discuss how to implement a service that is meant to run in its own process and how a client may communicate with that service using the `Messenger` framework. It will also briefly discuss two-way communication: the client sending a message to a service and the service sending a message back to the client. Because services can be shared between different applications, this guide will also discuss one technique for limiting client access to the service by using Android permissions.

> **IMPORTANT**
>
> Bugzilla 51940/GitHub 1950 - Services with isolated processes and custom Application class fail to resolve overloads properly reports that a Xamarin.Android service will not start up properly when the `IsolatedProcess` is set to `true`. This guide is provided for a reference. A Xamarin.Android application should still be able to communicate with an out-of-process service that is written in Java.

Requirements

This guide assumes familiarity with creating services.

Although it is possible to use implicit intents with apps that target older Android APIs, this guide will focus exclusively on the use of explicit intents. Apps targeting Android 5.0 (API level 21) or higher must use an explicit intent to bind with a service; this is the technique that will be demonstrated in this guide..

Create a Service that Runs in a Separate Process

As described above, the fact that a service is running in its own process means that some different APIs are involved. As a quick overview, here are the steps to bind with and consume a remote service:

- **Create the `Service` subclass** – Subclass the `Service` type and implement the lifecycle methods for a bound service. It is also necessary to set meta-data that will inform Android that the service is to run in its own process.
- **Implement a `Handler`** – The `Handler` is responsible for analyzing the client requests, extracting any parameters that were passed from the client, and invoking the appropriate methods on the service.
- **Instantiate a `Messenger`** – As described above, each `Service` must maintain an instance of the `Messenger` class that will route client requests to the `Handler` that was created in the previous step.

A service that is meant to run in its own process is, fundamentally, still a bound service. The service class will extend the base `Service` class and is decorated with the `ServiceAttribute` containing the meta-data that Android needs to bundle in the Android manifest. To begin with, the following properties of the `ServiceAttribute` that are important to an out-of-process service:

1. `Exported` – This property must be set to `true` to allow other applications to interact with the service. The default value of this property is `false`.
2. `Process` – This property must be set. It is used to specify the name of the process that the service will run in.
3. `IsolatedProcess` – This property will enable extra security, telling Android to run the service in an isolated sandbox with minimal permission to interact with the rest of the system. See Bugzilla 51940 - Services with isolated processes and custom Application class fail to resolve overloads properly.
4. `Permission` – It is possible to control client access to the service by specifying a permission that clients must request (and be granted).

To run a service its own process, the `Process` property on the `ServiceAttribute` must be set to the name of the service. To interact with outside applications, the `Exported` property should be set to `true`. If `Exported` is `false`, then only clients in the same APK (i.e. the same application) and running in the same process will be able to interact with the service.

What kind of process the service will run in depends on the value of the `Process` property. Android identifies three different types of processes:

- **Private Process** – A private process is one that is only available to the application that started it. To identify a process as private, its name must start with a : (semi-colon). The service depicted in the previous code snippet and screenshot is a private process. The following code snippet is an example of the `ServiceAttribute`:

  ```
  [Service(Name = "com.xamarin.TimestampService",
           Process=":timestampservice_process",
           Exported=true)]
  ```

- **Global Process** – A service that is run in a global process is accessible to all applications running on the device. A global process must be a fully qualified class name that starts with a lower case character. (Unless steps are taken to secure the service, other applications may bind and interact with it. Securing the service against unauthorized use will be discussed later in this guide.)

  ```
  [Service(Name = "com.xamarin.TimestampService",
           Process="com.xamarin.xample.messengerservice.timestampservice_process",
           Exported=true)]
  ```

- **Isolated Process** – An isolated process is a process that runs in its own sandbox, isolated from the rest of the system and with no special permissions of its own. To run a service in an isolated process, the `IsolatedProcess` property of the `ServiceAttribute` is set to `true` as shown in this code snippet:

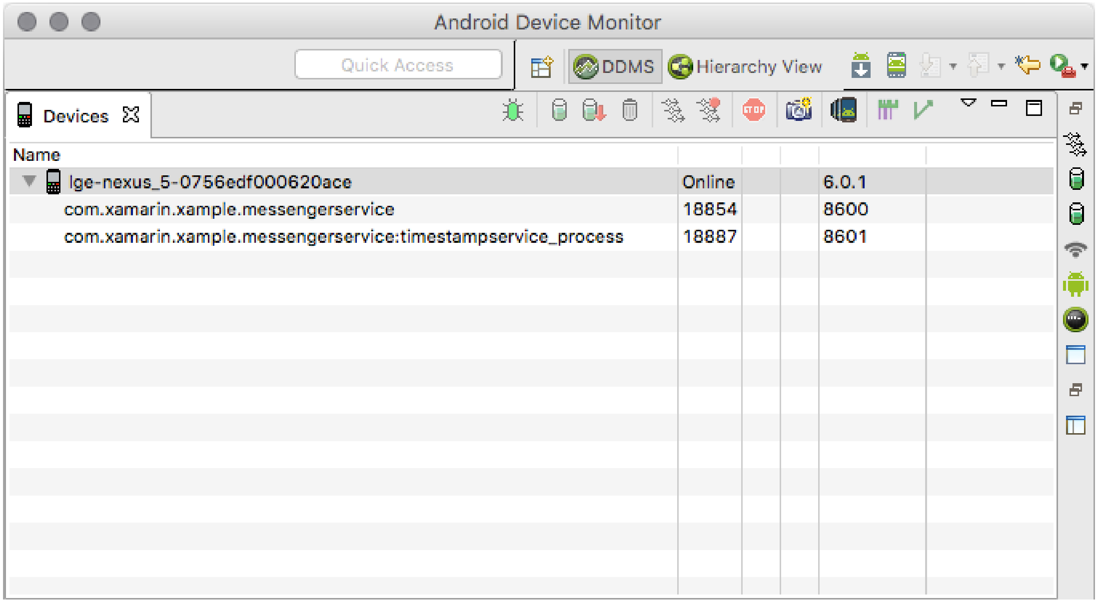
```
[Service(Name = "com.xamarin.TimestampService",
         IsolatedProcess= true,
         Process="com.xamarin.xample.messengerservice.timestampservice_process",
         Exported=true)]
```

> **IMPORTANT**
>
> See Bugzilla 51940 - Services with isolated processes and custom Application class fail to resolve overloads properly

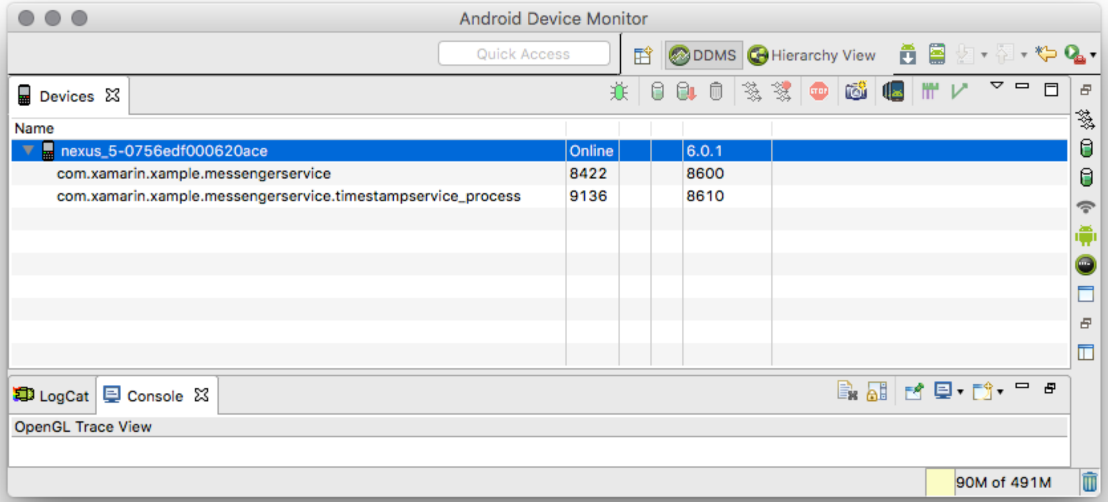
An isolated service is a simple way to secure an application and the device against untrusted code. For example, an app may download and execute a script from a website. In this case, performing this in an isolated process provides an additional layer of security against untrusted code compromising the Android device.

> **IMPORTANT**
>
> Once a service has been exported, the name of the service should not change. Changing the name of the service may break other applications that are using the service.

To see the effect that the `Process` property has, the following screenshot shows a service running in its own private process:



This next screenshot shows `Process="com.xamarin.xample.messengerservice.timestampservice_process"` and the service running in a global process:



Once the `ServiceAttribute` has been set, the service needs to implement a `Handler`.

Implementing a Handler

To process client requests, the service must implement a `Handler` and override the `HandleMessage` method. This is the method takes a `Message` instance which encapsulates the method call from the client and translates that call into some action or task that the service will perform. The `Message` object exposes a property called `What` which is an integer value, the meaning of which is shared between the client and the service and relates to some task that the service is to perform for the client.

The following code snippet from the sample application shows one example of `HandleMessage`. In this example, there are two actions that a client can request of the service:

- The first action is a *Hello, World* message, the client has sent a simple message to the service.
- The second action will invoke a method on the service and retrieve a string, in this case the string is a message that returns what time the service started and how long it has been running:

```
public class TimestampRequestHandler : Android.OS.Handler
{
    // other code omitted for clarity

    public override void HandleMessage(Message msg)
    {
        int messageType = msg.What;
        Log.Debug(TAG, $"Message type: {messageType}.");

        switch (messageType)
        {
            case Constants.SAY_HELLO_TO_TIMESTAMP_SERVICE:
                // The client has sent a simple Hello, say in the Android Log.
                break;

            case Constants.GET_UTC_TIMESTAMP:
                // Call methods on the service to retrieve a timestamp message.
                break;
            default:
                Log.Warn(TAG, $"Unknown messageType, ignoring the value {messageType}.");
                base.HandleMessage(msg);
                break;
        }
    }
}
```

It is also possible to package parameters for the service in the `Message`. This will be discussed later in this guide. The next topic to consider is creating the `Messenger` object to process the incoming `Message`s.

Instantiating the Messenger

As previously discussed, deserializing the `Message` object and invoking `Handler.HandleMessage` is the responsibility of the `Messenger` object. The `Messenger` class also provides an `IBinder` object that the client will use to send messages to the service.

When the service starts, it will instantiate the `Messenger` and inject the `Handler`. A good place to perform this initialization is on the `OnCreate` method of the service. This code snippet is one example of a service that initializes its own `Handler` and `Messenger`:

```
private Messenger messenger; // Instance variable for the Messenger

public override void OnCreate()
{
    base.OnCreate();
    messenger = new Messenger(new TimestampRequestHandler(this));
    Log.Info(TAG, $"TimestampService is running in process id {Android.OS.Process.MyPid()}.");
}
```

At this point, the final step is for the `Service` to override `OnBind`.

Implementing Service.OnBind

All bound services, whether they run in their own process or not, must implement the `OnBind` method. The return value of this method is some object that the client can use to interact with the service. Exactly what that object is depends whether the service is a local service or a remote service. While a local service will return a custom `IBinder` implementation, a remote service will return the `IBinder` that is encapsulated but the `Messenger` that was created in the previous section:

```
public override IBinder OnBind(Intent intent)
{
    Log.Debug(TAG, "OnBind");
    return messenger.Binder;
}
```

Once these three steps are accomplished, the remote service can be considered complete.

Consuming the Service

All clients must implement some code to be able to bind and consume the remote service. Conceptually, from the client's viewpoint, there are very few differences between binding to local service or a remote service. The client invokes the `BindService` method, passing an explicit intent to identify the service and an `IServiceConnection` that helps manage the connection between the client and the service.

This code snippet is an example of how to create an **explicit intent** for binding to a remote service. The intent must identify the package that contains the service and the name of the service. One way to set this information is to use an `Android.Content.ComponentName` object and to provide that to the intent. This code snippet is one example:

```
// This is the package name of the APK, set in the Android manifest
const string REMOTE_SERVICE_COMPONENT_NAME = "com.xamarin.TimestampService";
// This is the name of the service, according the value of ServiceAttribute.Name
const string REMOTE_SERVICE_PACKAGE_NAME   = "com.xamarin.xample.messengerservice";

// Provide the package name and the name of the service with a ComponentName object.
ComponentName cn = new ComponentName(REMOTE_SERVICE_PACKAGE_NAME, REMOTE_SERVICE_COMPONENT_NAME);
Intent serviceToStart = new Intent();
serviceToStart.SetComponent(cn);
```

When the service is bound, the `IServiceConnection.OnServiceConnected` method is invoked and provides an `IBinder` to a client. However, the client will not directly use the `IBinder`. Instead, it will instantiate a `Messenger` object from that `IBinder`. This is the `Messenger` that the client will use to interact with the remote service.

The following is an example of a very basic `IServiceConnection` implementation that demonstrates how a client would handle connecting to and disconnecting from a service. Notice that the `OnServiceConnected` method receives and `IBinder`, and the client creates a `Messenger` from that `IBinder`:

```
public class TimestampServiceConnection : Java.Lang.Object, IServiceConnection
{
    static readonly string TAG = typeof(TimestampServiceConnection).FullName;

    MainActivity mainActivity;
    Messenger messenger;

    public TimestampServiceConnection(MainActivity activity)
    {
        IsConnected = false;
        mainActivity = activity;
    }

    public bool IsConnected { get; private set; }
    public Messenger Messenger { get; private set; }

    public void OnServiceConnected(ComponentName name, IBinder service)
    {
        Log.Debug(TAG, $"OnServiceConnected {name.ClassName}");

        IsConnected = service != null;
        Messenger = new Messenger(service);

        if (IsConnected)
        {
            // things to do when the connection is successful. perhaps notify the client? enable UI
features?
        }
        else
        {
            // things to do when the connection isn't successful.
        }
    }

    public void OnServiceDisconnected(ComponentName name)
    {
        Log.Debug(TAG, $"OnServiceDisconnected {name.ClassName}");
        IsConnected = false;
        Messenger = null;

        // Things to do when the service disconnects. perhaps notify the client? disable UI features?

    }
}
```

Once the service connection and the intent are created, it is possible for the client to call `BindService` and initiate the binding process:

```
var serviceConnection = new TimestampServiceConnection(this);
BindService(serviceToStart, serviceConnection, Bind.AutoCreate);
```

After the client has successfully bound to the service and the `Messenger` is available, it is possible for the client to send `Messages` to the service.

Sending Messages to the Service

Once the client is connected and has a `Messenger` object, it is possible to communicate with the service by dispatching `Message` objects via the `Messenger`. This communication is one-way, the client sends the message but there is no return message from the service to the client. In this regard, the `Message` is a fire-and-forget mechanism.

The preferred way to create a `Message` object is to use the `Message.Obtain` factory method. This method will pull a `Message` object from a global pool that is maintained by Android. `Message.Obtain` also has some overloaded methods that allow the `Message` object to be initialized with the values and parameters required by the service. Once the `Message` is instantiated, it dispatched to the service by calling `Messenger.Send`. This snippet is one example of creating and dispatching a `Message` to the service process:

```
Message msg = Message.Obtain(null, Constants.SAY_HELLO_TO_TIMESTAMP_SERVICE);
try
{
    serviceConnection.Messenger.Send(msg);
}
catch (RemoteException ex)
{
    Log.Error(TAG, ex, "There was a error trying to send the message.");
}
```

There are several different forms of the `Message.Obtain` method. The previous example uses the `Message.Obtain(Handler h, Int32 what)`. Because this is an asynchronous request to an out-of-process service; there will be no response from the service, so the `Handler` is set to `null`. The second parameter, `Int32 what`, will be stored in the `.What` property of the `Message` object. The `.What` property is used by code in the service process to invoke methods on the service.

The `Message` class also exposes two additional properties that may be of use to the recipient: `Arg1` and `Arg2`. These two properties are integer values that may have some special agreed upon values that have meaning between the client and the service. For example, `Arg1` may hold a customer ID and `Arg2` may hold a purchase order number for that customer. The `Method.Obtain(Handler h, Int32 what, Int32 arg1, Int32 arg2)` can be used to set the two properties when the `Message` is created. Another way to populate these two values is to set the `.Arg` and `.Arg2` properties directly on the `Message` object after it has been created.

Passing Additional Values to the Service

It is possible to pass more complex data to the service by using a `Bundle`. In this case, extra values can be placed in a `Bundle` and sent along with the `Message` by setting the `.Data` property property before sending.

```
Bundle serviceParameters = new Bundle();
serviceParameters.

var msg = Message.Obtain(null, Constants.SERVICE_TASK_TO_PERFORM);
msg.Data = serviceParameters;

messenger.Send(msg);
```

NOTE

In general, a `Message` should not have a payload larger than 1MB. The size limit may vary according the version of Android and on any proprietary changes the vendor might have made to their implementation of the Android Open Source Project (AOSP) that is bundled with the device.

Returning Values from the Service

The messaging architecture that has been discussed to this point is one-way, the client sends a message to the service. If it is necessary for the service to return a value to a client then everything that has been discussed to this point is reversed. The service must create a `Message`, packaged any return values, and dispatch the `Message` via a `Messenger` to the client. However, the service does not create its own `Messenger`; instead, it relies on the client instantiating and package a `Messenger` as part of the initial request. The service will `Send` the message using this client-provided `Messenger`.

The sequence of events for two-way communication is this:

1. The client binds to the service. When the service and the client connect, the `IServiceConnection` that is maintained by the client will have a reference to a `Messenger` object that is used to transmit `Message`s to the service. To avoid confusion, this will be referred to as the *Service Messenger*.
2. Client instantiates a `Handler` (referred to as the *Client Handler*) and uses that to initialize its own `Messenger` (the *Client Messenger*). Note that the Service Messenger and the Client Messenger are two different objects that handle traffic in two different directions. The Service Messenger handles messages from the client to the service, while the Client Messenger will handle messages from the service to the client.
3. The client creates a `Message` object, and sets the `ReplyTo` property with the Client Messenger. The message is then sent to the service using the Service Messenger.
4. The service receives the message from the client, and performs the requested work.
5. When it is time for the service to send the response to the client, it will use `Message.Obtain` to create a new `Message` object.
6. To send this message to the client, the service will extract the Client Messenger from the `.ReplyTo` property of the client message and use that to `.Send` the `Message` back to the client.
7. When the response is received by the client, it has its own `Handler` that will process the `Message` by inspecting the `.What` property (and if necessary, extracting any parameters contained by the `Message`).

This sample code demonstrates how the client will instantiate the `Message` and package a `Messenger` that the service should use for its response:

```
Handler clientHandler = new ActivityHandler();
Messenger clientMessenger = new Messenger(activityHandler);

Message msg = Message.Obtain(null, Constants.GET_UTC_TIMESTAMP);
msg.ReplyTo = clientMessenger;

try
{
    serviceConnection.Messenger.Send(msg);
}
catch (RemoteException ex)
{
    Log.Error(TAG, ex, "There was a problem sending the message.");
}
```

The service must make some changes to its own `Handler` to extract the `Messenger` and use that to send replies to the client. This code snippet is an example of how the service's `Handler` would create a `Message` and send it back to the client:

```
// This is the message that the service will send to the client.
Message responseMessage = Message.Obtain(null, Constants.RESPONSE_TO_SERVICE);
Bundle dataToReturn = new Bundle();
dataToReturn.PutString(Constants.RESPONSE_MESSAGE_KEY, "This is the result from the service.");
responseMessage.Data = dataToReturn;

// The msg object here is the message that was received by the service. The service will not instantiate a
client,
// It will use the client that is encapsulated by the message from the client.
Messenger clientMessenger = msg.ReplyTo;
if (clientMessenger!= null)
{
    try
    {
        clientMessenger.Send(responseMessage);
    }
    catch (Exception ex)
    {
        Log.Error(TAG, ex, "There was a problem sending the message.");
    }
}
```

Note that in the code samples above, the `Messenger` instance that is created by the client is *not* the same object that is received by the service. These are two different `Messenger` objects running in two separate processes that represent the communication channel.

Securing the Service with Android Permissions

A service that runs in a global process is accessible by all applications running on that Android device. In some situations, this openness and availability is undesirable, and it is necessary to secure the service against access from unauthorized clients. One way to limit access to the remote service is to use Android Permissions.

Permissions can be identified by the `Permission` property of the `ServiceAttribute` that decorates the `Service` sub-class. This will name a permission that the client must be granted when binding to the service. If the client does not have the appropriate permissions, then Android will throw a `Java.Lang.SecurityException` when the client tries to bind to the service.

There are four different permission levels that Android provides:

- **normal** – This is the default permission level. It is used to identify low-risk permissions that can be automatically granted by Android to clients that request it. The user does not have to explicitly grant these permissions, but the permissions can be viewed in the app settings.

- **signature** – This is a special category of permission that will be granted automatically by Android to applications that are all signed with the same certificate. This permission is designed to make it easily for an application developer to share components or data between their apps without bothering the user for constant approvals.
- **signatureOrSystem** – This is very similar to the **signature** permissions described above. In addition to being automatically granted to apps that are signed by the same certificate, this permission will also be granted to apps that are signed the same certificate that was used to sign the apps installed with the Android system image. This permission is typically only used by Android ROM developers to allow their applications to work with third party apps. It is not commonly used by apps that are meant general distribution for the public at large.
- **dangerous** – Dangerous permissions are those that could cause problems for the user. For this reason, **dangerous** permissions must be explicitly approved by the user.

Because `signature` and `normal` permissions are automatically granted at installed time by Android, it is crucial that APK hosting the service be installed **before** the APK containing the client. If the client is installed first, Android will not grant the permissions. In this case, it will be necessary to uninstall the client APK, install the service APK, and then re-install the client APK.

There are two common ways to secure a service with Android permissions:

1. **Implement signature level security** – Signature level security means that permission is automatically granted to those applications that are signed with the same key that was used to sign the APK holding the service. This is a simple way for developers to secure their service yet keep them accessible from their own applications. Signature level permissions are declared by setting the `Permission` property of the `ServiceAttribute` to `signature`:

```
[Service(Name = "com.xamarin.TimestampService",
         Process="com.xamarin.TimestampService.timestampservice_process",
         Permission="signature")]
public class TimestampService : Service
{
}
```

2. **Create a custom permission** – It is possible for the developer of the service to create a custom permission for the service. This is best for when a developer wants to share their service with applications from other developers. A custom permission requires a bit more effort to implement and will be covered below.

A simplified example of creating a custom `normal` permission will be described in the next section. For more information about Android permissions, please consult Google's documentation for Best Practices & Security. For more information about Android permissions, see the Permissions section of the Android documentation for the application manifest for more information about Android permissions.

> **NOTE**
>
> In general, Google discourages the use of custom permissions as they may prove confusing to users.

Creating a Custom Permission

To use a custom permission, it is declared by the service while the client explicitly requests that permission.

To create a permission in the service APK, a `permission` element is added to the `manifest` element in **AndroidManifest.xml**. This permission must have the `name`, `protectionLevel`, and `label` attributes set. The `name` attribute must be set to a string that uniquely identifies the permission. The name will be displayed in the

App Info view of the **Android Settings** (as shown in the next section).

The `protectionLevel` attribute must be set to one of the four string values that were described above. The `label` and `description` must refer to string resources and are used to provide a user-friendly name and description to the user.

This snippet is an example of declaring a custom `permission` attribute in **AndroidManifest.xml** of the APK that contains the service:

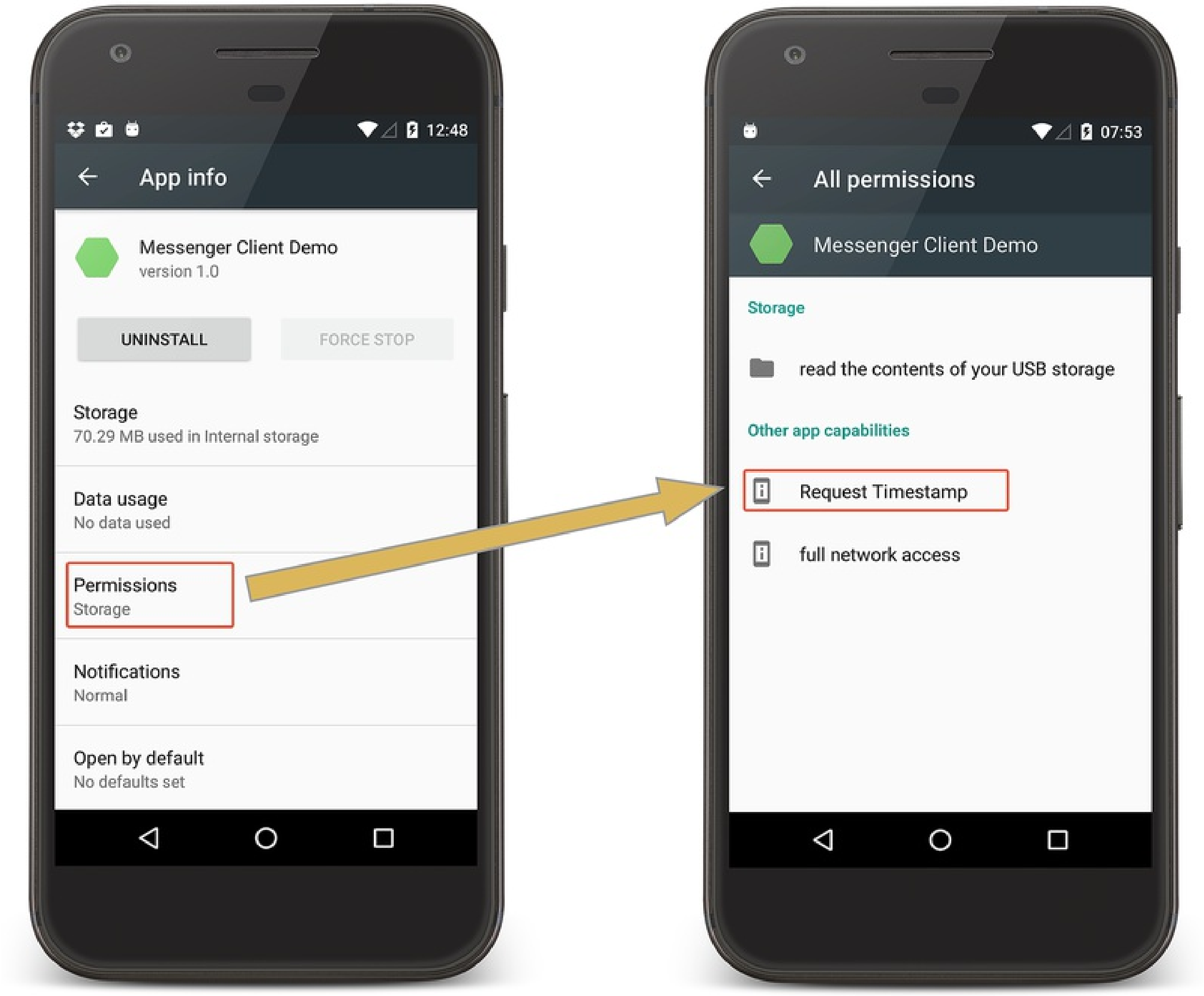
```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          android:versionCode="1"
          android:versionName="1.0"
          package="com.xamarin.xample.messengerservice">

    <uses-sdk android:minSdkVersion="21" />

    <permission android:name="com.xamarin.xample.messengerservice.REQUEST_TIMESTAMP"
                android:protectionLevel="signature"
                android:label="@string/permission_label"
                android:description="@string/permission_description"
                />

    <application android:allowBackup="true"
            android:icon="@mipmap/icon"
            android:label="@string/app_name"
            android:theme="@style/AppTheme">

    </application>
</manifest>
```

Then, the **AndroidManifest.xml** of the client APK must explicitly request this new permission. This is done by adding the `users-permission` attribute to the **AndroidManifest.xml**:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          android:versionCode="1"
          android:versionName="1.0"
          package="com.xamarin.xample.messengerclient">

    <uses-sdk android:minSdkVersion="21" />

    <uses-permission android:name="com.xamarin.xample.messengerservice.REQUEST_TIMESTAMP" />

    <application
            android:allowBackup="true"
            android:icon="@mipmap/icon"
            android:label="@string/app_name"
            android:theme="@style/AppTheme">
    </application>
    </manifest>
```

View the Permissions Granted to an App

To view the permissions that an application has been granted, open the Android Settings app, and select **Apps**. Find and select the application in the list. From the **App Info** screen, tap **Permissions** which will bring up a view that shows all the permissions granted to the app:



Summary

This guide was an advanced discussion about how to run an Android service in a remote process. The differences between a local and a remote service was explained, along with some reasons why a remote service can be helpful to stability and performance of an Android app. After explaining how to implement a remote service and how a client can communicate with the service, the guide went on to provide one way to limit access to the service from only authorized clients.

Related Links

- Handler
- Message
- Messenger
- ServiceAttribute
- The Exported attribute
- Services with isolated processes and custom Application class fail to resolve overloads properly
- Processes and Threads
- Android Manifest - Permissions
- Security Tips
- MessengerServiceDemo (sample)

Service Notifications

7/8/2021 • 2 minutes to read • Edit Online

This guide discusses how an Android service may use local notifications to dispatch information to a user.

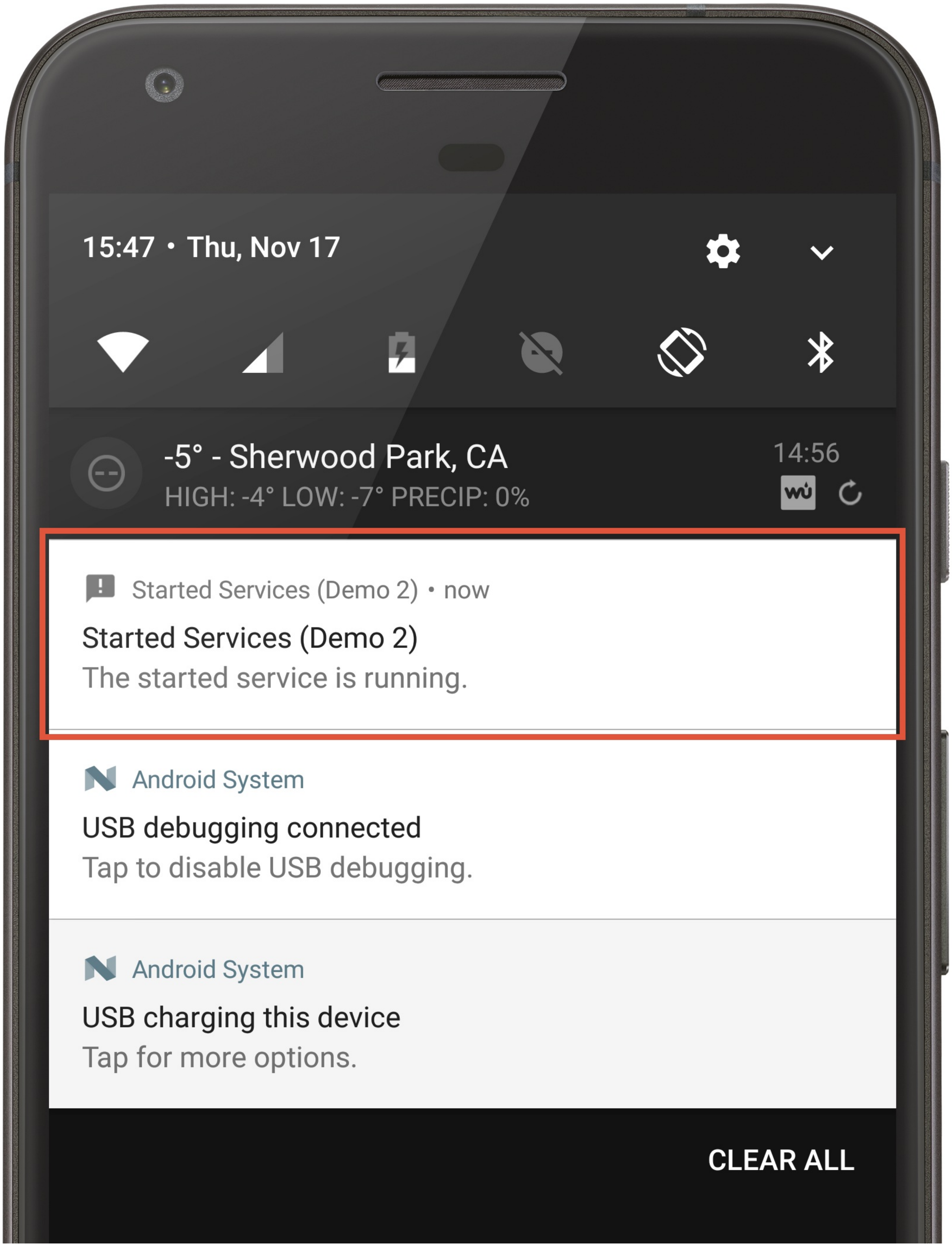
Service Notifications Overview

Service notifications allow an app to display information to the user, even if the Android application is not in the foreground. It is possible for a notification to provide actions for the user, such as displaying an Activity from an application. The following code sample demonstrates how a service might dispatch a notification to a user:

```
[Service]
public class MyService: Service
{
    // A notification requires an id that is unique to the application.
    const int NOTIFICATION_ID = 9000;

    public override StartCommandResult OnStartCommand(Intent intent, StartCommandFlags flags, int startId)
    {
        // Code omitted for clarity - here is where the service would do something.

        // Work has finished, now dispatch anotification to let the user know.
        Notification.Builder notificationBuilder = new Notification.Builder(this)
            .SetSmallIcon(Resource.Drawable.ic_notification_small_icon)
            .SetContentTitle(Resources.GetString(Resource.String.notification_content_title))
            .SetContentText(Resources.GetString(Resource.String.notification_content_text));

        var notificationManager = (NotificationManager)GetSystemService(NotificationService);
        notificationManager.Notify(NOTIFICATION_ID, notificationBuilder.Build());
    }
}
```

This screenshot is an example of the notification that is displayed:

When the user slides down the notification screen from the top, the full notification is displayed:



Updating A Notification

To update a notification, the service will republish the notification using the same notification ID. Android will display or update the notification in the status bar as necessary.

```
void UpdateNotification(string content)
{
   var notification = GetNotification(content, pendingIntent);

   NotificationManager notificationManager =
(NotificationManager)GetSystemService(Context.NotificationService);
   notificationManager.Notify(NOTIFICATION_ID, notification);
}

Notification GetNotification(string content, PendingIntent intent)
{
   return new Notification.Builder(this)
           .SetContentTitle(tag)
           .SetContentText(content)
           .SetSmallIcon(Resource.Drawable.NotifyLg)
           .SetLargeIcon(BitmapFactory.DecodeResource(Resources, Resource.Drawable.Icon))
           .SetContentIntent(intent).Build();
}
```

More information about notifications is available in the Local Notifications section of the Android Notifications guide.

Related Links

- Local Notifications in Android

Broadcast Receivers in Xamarin.Android

11/2/2020 • 8 minutes to read • Edit Online

This section discusses how to use a Broadcast Receiver.

Broadcast Receiver Overview

A *broadcast receiver* is an Android component that allows an application to respond to messages (an Android `Intent`) that are broadcast by the Android operating system or by an application. Broadcasts follow a *publish-subscribe* model – an event causes a broadcast to be published and received by those components that are interested in the event.

Android identifies two types of broadcasts:

- **Explicit broadcast** – These types of broadcasts target a specific application. The most common use of an explicit broadcast is to start an Activity. An example of an explicit broadcast when an app needs to dial a phone number; it will dispatch an Intent that targets the Phone app on Android and pass along the phone number to be dialed. Android will then route the intent to the Phone app.
- **Implicit broadcast** – These broadcasts are dispatched to all apps on the device. An example of an implicit broadcast is the `ACTION_POWER_CONNECTED` intent. This intent is published each time Android detects that the battery on the device is charging. Android will route this intent to all apps that have registered for this event.

The broadcast receiver is a subclass of the `BroadcastReceiver` type and it must override the `OnReceive` method. Android will execute `OnReceive` on the main thread, so this method should be designed to execute quickly. Care should be taken when spawning threads in `OnReceive` because Android may terminate the process when the method finishes. If a broadcast receiver must perform long running work then it is recommended to schedule a *job* using the `JobScheduler` or the *Firebase Job Dispatcher*. Scheduling work with a job will be discussed in a separate guide.

An *intent filter* is used to register a broadcast receiver so that Android can properly route messages. The intent filter can be specified at runtime (this is sometimes referred to as a *context-registered receiver* or as *dynamic registration*) or it can be statically defined in the Android Manifest (a *manifest-registered receiver*). Xamarin.Android provides a C# attribute, `IntentFilterAttribute`, that will statically register the intent filter (this will be discussed in more detail later in this guide). Starting in Android 8.0, it is not possible for an application to statically register for an implicit broadcast.

The primary difference between the manifest-registered receiver and the context-registered receiver is that a context-registered receiver will only respond to broadcasts while an application is running, while a manifest-registered receiver can respond to broadcasts even though the app may not be running.

There are two sets of APIs for managing a broadcast receiver and sending broadcasts:

1. `Context` – The `Android.Content.Context` class can be used to register a broadcast receiver that will respond to system-wide events. The `Context` is also used to publish system-wide broadcasts.
2. `LocalBroadcastManager` – This is an API that is available through the Xamarin Support Library v4 NuGet package. This class is used to keep broadcasts and broadcast receivers isolated in the context of the application that is using them. This class can be useful for preventing other applications from responding to application-only broadcasts or sending messages to private receivers.

A broadcast receiver may not display dialogs, and it is strongly discouraged to start an activity from within a broadcast receiver. If a broadcast receiver must notify the user, then it should publish a notification.

It is not possible to bind to or start a service from within a broadcast receiver.

This guide will cover how to create a broadcast receiver and how to register it so that it may receive broadcasts.

Creating a Broadcast Receiver

To create a broadcast receiver in Xamarin.Android, an application should subclass the `BroadcastReceiver` class, adorn it with the `BroadcastReceiverAttribute`, and override the `OnReceive` method:

```
[BroadcastReceiver(Enabled = true, Exported = false)]
public class SampleReceiver : BroadcastReceiver
{
    public override void OnReceive(Context context, Intent intent)
    {
        // Do stuff here.

        String value = intent.GetStringExtra("key");
    }
}
```

When Xamarin.Android compiles the class, it will also update the AndroidManifest with the necessary meta-data to register the receiver. For a statically-registered broadcast receiver, the `Enabled` properly must be set to `true`, otherwise Android will not be able to create an instance of the receiver.

The `Exported` property controls whether the broadcast receiver can receive messages from outside the application. If the property is not explicitly set, the default value of the property is determined by Android based on if there are any intent-filters associated with the broadcast receiver. If there is at least one intent-filter for the broadcast receiver then Android will assume that the `Exported` property is `true`. If there are no intent-filters associated with the broadcast receiver, then Android will assume that the value is `false`.

The `OnReceive` method receives a reference to the `Intent` that was dispatched to the broadcast receiver. This makes it possible for the sender of the intent to pass values to the broadcast receiver.

Statically registering a Broadcast Receiver with an Intent Filter

When a `BroadcastReceiver` is decorated with the `IntentFilterAttribute`, Xamarin.Android will add the necessary `<intent-filter>` element to the Android manifest at compile time. The following snippet is an example of a broadcast receiver that will run when a device has finished booting (if the appropriate Android permissions were granted by the user):

```
[BroadcastReceiver(Enabled = true)]
[IntentFilter(new[] { Android.Content.Intent.ActionBootCompleted })]
public class MyBootReceiver : BroadcastReceiver
{
    public override void OnReceive(Context context, Intent intent)
    {
        // Work that should be done when the device boots.
    }
}
```

NOTE

In Android 8.0 (API 26 and above), Google placed limitations on what apps can do while users aren't directly interacting with them. These limitations affect background services and implicit broadcast receivers such as `Android.Content.Intent.ActionBootCompleted`. Because of these limitations, you might have difficulties registering a `Boot Completed` broadcast receiver on newer versions of Android. If this is the case, note that these restrictions do not apply to foreground services, which can be called from your broadcast receiver.

It is also possible to create an intent filter that will respond to custom intents. Consider the following example:

```
[BroadcastReceiver(Enabled = true)]
[IntentFilter(new[] { "com.xamarin.example.TEST" })]
public class MySampleBroadcastReceiver : BroadcastReceiver
{
    public override void OnReceive(Context context, Intent intent)
    {
        // Do stuff here
    }
}
```

Apps that target Android 8.0 (API level 26) or higher may not statically register for an implicit broadcast. Apps may still statically register for an explicit broadcast. There is a small list of implicit broadcasts that are exempt from this restriction. These exceptions are described in the Implicit Broadcast Exceptions guide in the Android documentation. Apps that are interested in implicit broadcasts must do so dynamically using the `RegisterReceiver` method. This is described next.

Context-Registering a Broadcast Receiver

Context-registration (also referred to as dynamic registration) of a receiver is performed by invoking the `RegisterReceiver` method, and the broadcast receiver must be unregistered with a call to the `UnregisterReceiver` method. To prevent leaking resources, it is important to unregister the receiver when it is no longer relevant for the context (the Activity or service). For example, a service may broadcast an intent to inform an Activity that updates are available to be displayed to the user. When the Activity starts, it would register for those Intents. When the Activity is moved into the background and no longer visible to the user, it should unregister the receiver because the UI for displaying the updates is no longer visible. The following code snippet is an example of how to register and unregister a broadcast receiver in the context of an Activity:

```
[Activity(Label = "MainActivity", MainLauncher = true, Icon = "@mipmap/icon")]
public class MainActivity: Activity
{
    MySampleBroadcastReceiver receiver;

    protected override void OnCreate(Bundle savedInstanceState)
    {
        base.OnCreate(savedInstanceState);
        receiver = new MySampleBroadcastReceiver();

        // Code omitted for clarity
    }

    protected override void OnResume()
    {
        base.OnResume();
        RegisterReceiver(receiver, new IntentFilter("com.xamarin.example.TEST"));
        // Code omitted for clarity
    }

    protected override void OnPause()
    {
        UnregisterReceiver(receiver);
        // Code omitted for clarity
        base.OnPause();
    }
}
```

In the previous example, when the Activity comes into the foreground, it will register a broadcast receiver that will listen for a custom intent by using the `OnResume` lifecycle method. As the Activity moves into the background, the `OnPause()` method will unregister the receiver.

Publishing a Broadcast

A broadcast may be published to all apps installed on the device creating an Intent object and dispatching it with the `SendBroadcast` or the `SendOrderedBroadcast` method.

1. **Context.SendBroadcast methods** – There are several implementations of this method. These methods will broadcast the intent to the entire system. Broadcast receivers that will receive the intent in an indeterminate order. This provides a great deal of flexibility but means that it is possible for other applications to register and receive the intent. This can pose a potential security risk. Applications may need to implement addition security to prevent unauthorized access. One possible solution is to use the `LocalBroadcastManager` which will only dispatch messages within the private space of the app. This code snippet is one example of how to dispatch an intent using one of the `SendBroadcast` methods:

   ```
   Intent message = new Intent("com.xamarin.example.TEST");
   // If desired, pass some values to the broadcast receiver.
   message.PutExtra("key", "value");
   SendBroadcast(message);
   ```

   This snippet is another example of sending a broadcast by using the `Intent.SetAction` method to identify the action:

   ```
   Intent intent = new Intent();
   intent.SetAction("com.xamarin.example.TEST");
   intent.PutExtra("key", "value");
   SendBroadcast(intent);
   ```

2. **Context.SendOrderedBroadcast** – This is method is very similar to `Context.SendBroadcast`, with the difference being that the intent will be published one at time to receivers, in the order that the receivers were registered.

LocalBroadcastManager

The Xamarin Support Library v4 provides a helper class called `LocalBroadcastManager`. The `LocalBroadcastManager` is intended for apps that do not want to send or receive broadcasts from other apps on the device. The `LocalBroadcastManager` will only publish messages within the context of the application, and only to those broadcast receivers that are registered with the `LocalBroadcastManager`. This code snippet is an example of registering a broadcast receiver with `LocalBroadcastManager`:

```
Android.Support.V4.Content.LocalBroadcastManager.GetInstance(this). RegisterReceiver(receiver, new
IntentFilter("com.xamarin.example.TEST"));
```

Other apps on the device cannot receive the messages that are published with the `LocalBroadcastManager`. This code snippet shows how to dispatch an Intent using the `LocalBroadcastManager`:

```
Intent message = new Intent("com.xamarin.example.TEST");
// If desired, pass some values to the broadcast receiver.
message.PutExtra("key", "value");
Android.Support.V4.Content.LocalBroadcastManager.GetInstance(this).SendBroadcast(message);
```

Related Links

- BroadcastReceiver API
- Context.RegisterReceiver API

- Context.SendBroadcast API
- Context.UnregisterReceiver API
- Intent API
- IntentFilter API
- LocalBroadcastManager (Android docs)
- Local Notifications in Android guide
- Android Support Library v4 (NuGet)

Android Localization

7/8/2021 • 6 minutes to read • Edit Online

This document introduces the localization features of the Android SDK and how to access them with Xamarin.

Android Platform Features

This section describes the main localization features of Android. Skip to the next section to see specific code and examples.

Locale

Users choose their language in **Settings > Language & input**. This selection controls both the language displayed and regional settings used (eg. for date and number formatting).

The current locale can be queried via the current context's `Resources`:

```
var lang = Resources.Configuration.Locale; // eg. "es_ES"
```

This value will be a locale identifier that contains both a language code and a locale code, separated by an underscore. For reference, here is a list of Java locales and Android-supported locales via StackOverflow.

Common examples include:

- `en_US` for English (United States)
- `es_ES` for Spanish (Spain)
- `ja_JP` for Japanese (Japan)
- `zh_CN` for Chinese (China)
- `zh_TW` for Chinese (Taiwan)
- `pt_PT` for Portuguese (Portugal)
- `pt_BR` for Portuguese (Brazil)

LOCALE_CHANGED

Android generates `android.intent.action.LOCALE_CHANGED` when the user changes their language selection.

Activities can opt to handle this by setting the `android:configChanges` attribute on the activity, like this:

```
[Activity (Label = "@string/app_name", MainLauncher = true, Icon="@drawable/launcher",
    ConfigurationChanges = ConfigChanges.Locale | ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
```

Internationalization Basics in Android

Android's localization strategy has the following key parts:

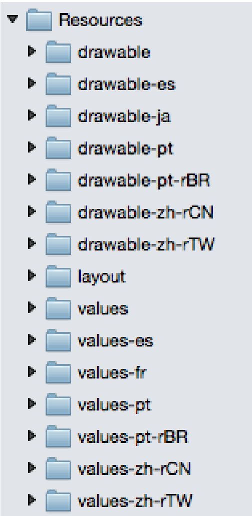
- Resource folders to contain localized strings, images, and other resources.
- `GetText` method, which is used to retrieve localized strings in code
- `@string/id` in AXML files, to automatically place localized strings in layouts.

Resource Folders

Android applications manage most content in resource folders, such as:

- **layout** - contains AXML layout files.
- **drawable** - contains images and other drawable resources.
- **values** - contains strings.
- **raw** - contains data files.

Most developers are already familiar with the use of **dpi** suffixes on the **drawable** directory to provide multiple versions of an image, letting Android choose the correct version for each device. The same mechanism is used to provide multiple language translations by suffixing resource directories with language and culture identifiers.



> **NOTE**
>
> When specifying a top-level language like `es` only two characters are required; however when specifying a full locale, the directory name format requires a dash and lowercase **r** to separate the two parts, for example **pt-rBR** or **zh-rCN**. Compare this to the value returned in code, which has an underscore (eg. `pt_BR`). Both of these are different to the value .NET `CultureInfo` class uses, which has a dash only (eg. `pt-BR`). Keep these differences in mind when working across Xamarin platforms.

Strings.xml file format

A localized **values** directory (eg. **values-es** or **values-pt-rBR**) should contain a file called **Strings.xml** that will contain the translated text for that locale.

Each translatable string is an XML element with the resource ID specified as the `name` attribute and the translated string as the value:

```
<string name="app_name">TaskyL10n</string>
```

You need to escape according to normal XML rules, and the `name` must be a valid Android resource ID (no spaces or dashes). Here is an example of the default (English) strings file for the example:

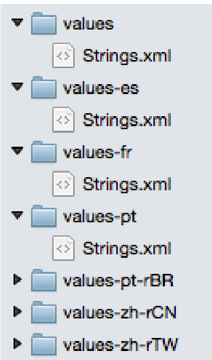
values/Strings.xml

```
<resources>
    <string name="app_name">TaskyL10n</string>
    <string name="taskadd">Add Task</string>
    <string name="taskname">Name</string>
    <string name="tasknotes">Notes</string>
    <string name="taskdone">Done</string>
    <string name="taskcancel">Cancel</string>
</resources>
```

The Spanish directory **values-es** contains a file with the same name (**Strings.xml**) that contains the translations:

values-es/Strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">TaskyLeon</string>
    <string name="taskadd">agregar tarea</string>
    <string name="taskname">Nombre</string>
    <string name="tasknotes">Notas</string>
    <string name="taskdone">Completo</string>
    <string name="taskcancel">Cancelar</string>
</resources>
```



With the strings files set-up, the translated values can be referenced in both layouts and code.

AXML Layout Files

To reference localized strings in layout files, use the `@string/id` syntax. This XML snippet from the sample shows `text` properties being set with localized resource IDs (some other attributes have been omitted):

```
<TextView
    android:id="@+id/NameLabel"
    android:text="@string/taskname"
    ... />
<CheckBox
    android:id="@+id/chkDone"
    android:text="@string/taskdone"
    ... />
```

GetText Method

To retrieve translated strings in code, use the `GetText` method and pass the resource ID:

```
var cancelText = Resources.GetText (Resource.String.taskcancel);
```

Quantity Strings

Android string resources also let you create *quantity strings* which allow translators to provide different translations for different quantities, such as:

- "There is 1 task left."
- "There are 2 tasks still to do."

(rather than a generic "There are n task(s) left").

In the **Strings.xml**

```
<plurals name="numberOfTasks">
   <!--
      As a developer, you should always supply "one" and "other"
      strings. Your translators will know which strings are actually
      needed for their language.
    -->
   <item quantity="one">There is %d task left.</item>
   <item quantity="other">There are %d tasks still to do.</item>
 </plurals>
```

To render the complete string use the `GetQuantityString` method, passing the resource ID and the value to be displayed (which is passed twice). The second parameter is used by Android to determine *which* `quantity` string to use, the third parameter is the value actually substituted into the string (both are required).

```
var translated = Resources.GetQuantityString (
                    Resource.Plurals.numberOfTasks, taskcount, taskcount);`
```

Valid `quantity` switches are:

- zero
- one
- two
- few
- many
- other

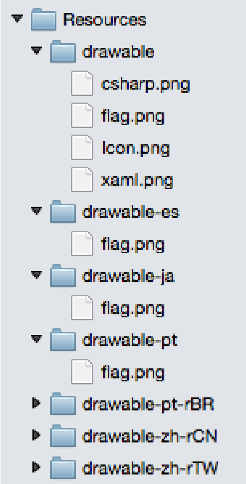
They're described in more detail in the Android docs. If a given language does not require 'special' handling, those `quantity` strings will be ignored (for example, English only uses `one` and `other`; specifying a `zero` string will have no effect, it will not be used).

Images

Localized images follow the same rules as strings files: all images referenced in the application should be placed in **drawable** directories so there is a fallback.

Locale-specific images should then be placed in qualified drawable folders such as **drawable-es** or **drawable-ja** (dpi specifiers can also be added).

In this screenshot, four images are saved in the **drawable** directory, but only one, **flag.png**, has localized copies in other directories.



Other Resource Types

You can also provide other types of alternative, language-specific resources including layouts, animations, and raw files. This means you could provide a specific screen layout for one or more of your target languages, for example you could create a layout specifically for German that allows for very long text labels.

Android 4.2 introduced support for right to left (RTL) languages if you set the application setting `android:supportsRtl="true"`. The resource qualifier `"ldrtl"` can be included in a directory name to contain custom layouts that are designed for RTL display.

For more information on resource directory naming and fallback, refer to the Android docs for providing alternative resources.

App name

The application name is easy to localize by using a `@string/id` in for the `MainLauncher` activity:

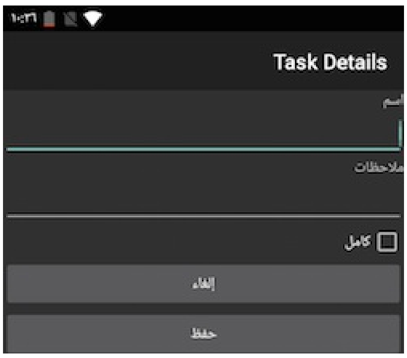
```
[Activity (Label = "@string/app_name", MainLauncher = true, Icon="@drawable/launcher",
    ConfigurationChanges =  ConfigChanges.Orientation | ConfigChanges.Locale)]
```

Right-to-Left (RTL) Languages

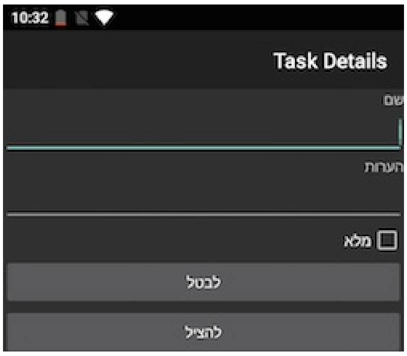
Android 4.2 and newer provides full support for RTL layouts, described in detail in the Native RTL Support blog.

When using Android 4.2 (API level 17) and newer, alignment values can be specified with `start` and `end` instead of `left` and `right` (for example `android:paddingStart`). There are also new APIs like `LayoutDirection`, `TextDirection`, and `TextAlignment` to help build screens that adapt for RTL readers.

The following screenshot shows the localized **Tasky** sample in Arabic:



The next screenshot shows the localized **Tasky** sample in Hebrew:



RTL text is localized using **Strings.xml** files in the same way as LTR text.

Testing

Make sure to thoroughly test the default locale. Your application will crash if the default resources cannot be loaded for some reason (i.e. they are missing).

Emulator Testing

Refer to Google's Testing on an Android Emulator section for instructions on how to set an emulator to a specific locale using the ADB shell.

```
adb shell setprop persist.sys.locale fr-CA;stop;sleep 5;start
```

Device Testing

To test on a device, change the language in the **Settings** app.

> **TIP**
>
> Make a note of the icons and location of the menu items so that you can revert the language to the original setting.

Summary

This article covers the basics of localizing Android applications using the built-in resource handling. You can learn more about i18n and L10n for iOS, Android and cross-platform (including Xamarin.Forms) apps in this cross-platform guide.

Related Links

- Tasky (localized in code) (sample)
- Android Localizing with Resources
- Cross-Platform Localization Overview
- Xamarin.Forms Localization
- iOS Localization

Permissions In Xamarin.Android

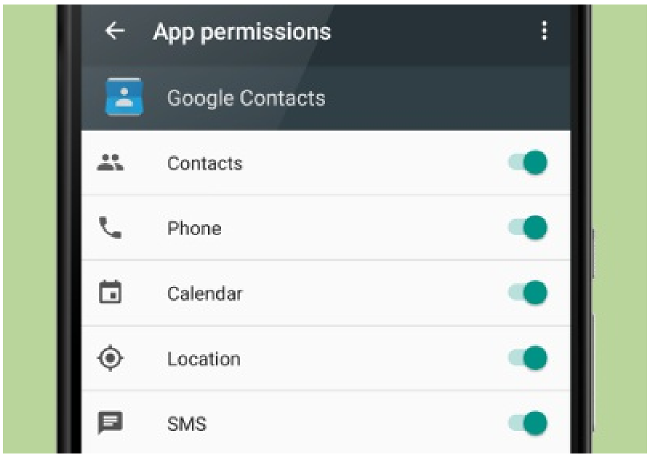
7/8/2021 • 9 minutes to read • Edit Online

Overview

Android applications run in their own sandbox and for security reasons do not have access to certain system resources or hardware on the device. The user must explicitly grant permission to the app before it may use these resources. For example, an application cannot access the GPS on a device without explicit permission from the user. Android will throw a `Java.Lang.SecurityException` if an app tries to access a protected resource without permission.

Permissions are declared in the **AndroidManifest.xml** by the application developer when the app is developed. Android has two different workflows for obtaining the user's consent for those permissions:

- For apps that targeted Android 5.1 (API level 22) or lower, the permission request occurred when the app was installed. If the user did not grant the permissions, then the app would not be installed. Once the app is installed, there is no way to revoke the permissions except by uninstalling the app.
- Starting in Android 6.0 (API level 23), users were given more control over permissions; they can grant or revoke permissions as long as the app is installed on the device. This screenshot shows the permission settings for the Google Contacts app. It lists the various permissions and allows the user to enable or disable permissions:



Android apps must check at run-time to see if they have permission to access a protected resource. If the app does not have permission, then it must make requests using the new APIs provided by the Android SDK for the user to grant the permissions. Permissions are divided into two categories:

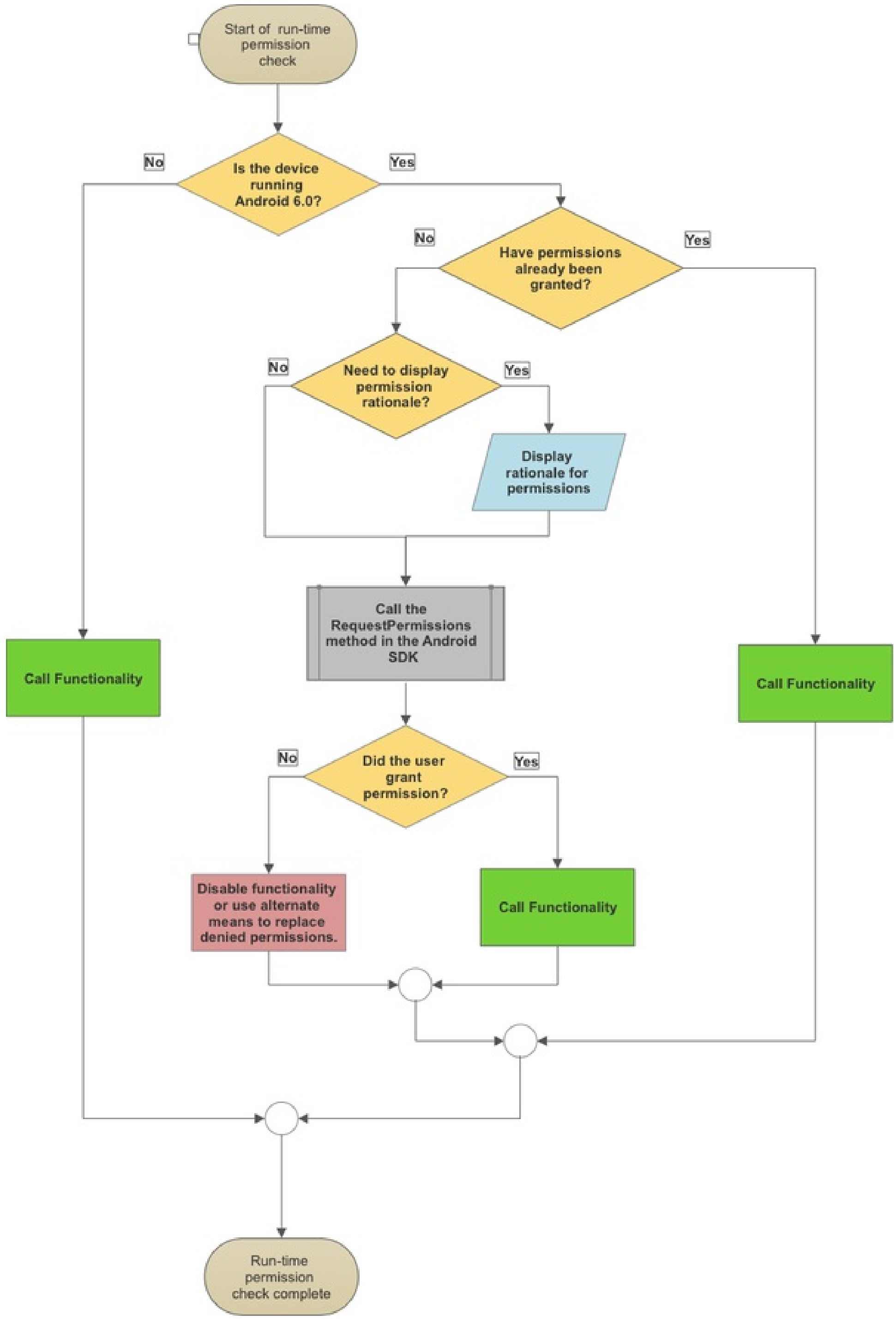
- **Normal Permissions** – These are permissions which pose little security risk to the user's security or privacy. Android 6.0 will automatically grant normal permissions at the time of installation. Please consult the Android documentation for a complete list of normal permissions.
- **Dangerous Permissions** – In contrast to normal permissions, dangerous permissions are those that protect the user's security or privacy. These must be explicitly granted by the user. Sending or receiving an SMS message is an example of an action requiring a dangerous permission.

> **IMPORTANT**
>
> The category that a permission belongs to may change over time. It is possible that a permission which was categorized as a "normal" permission may be elevated in future API levels to a dangerous permission.

Dangerous permissions are further sub-divided into *permission groups*. A permission group will hold permissions that are logically related. When the user grants permission to one member of a permission group, Android automatically grants permission to all members of that group. For example, the `STORAGE` permission group holds both the `WRITE_EXTERNAL_STORAGE` and `READ_EXTERNAL_STORAGE` permissions. If the user grants permission to `READ_EXTERNAL_STORAGE`, then the `WRITE_EXTERNAL_STORAGE` permission is automatically granted at the same time.

Before requesting one or more permissions, it is a best practice to provide a rationale as to why the app requires the permission before requesting the permission. Once the user understands the rationale, the app can request permission from the user. By understanding the rationale, the user can make an informed decision if they wish to grant the permission and understand the repercussions if they do not.

The whole workflow of checking and requesting permissions is known as a *run-time permissions* check, and can be summarized in the following diagram:



The Android Support Library backports some of the new APIs for permissions to older versions of Android. These backported APIs will automatically check the version of Android on the device so it is not necessary to perform an API level check each time.

This document will discuss how to add permissions to a Xamarin.Android application and how apps that target Android 6.0 (API level 23) or higher should perform a run-time permission check.

> **NOTE**
>
> It is possible that permissions for hardware may affect how the app is filtered by Google Play. For example, if the app requires permission for the camera, then Google Play will not show the app in the Google Play Store on a device that does not have a camera installed.

Requirements

It is strongly recommended that Xamarin.Android projects include the Xamarin.Android.Support.Compat NuGet package. This package will backport permission specific APIs to older versions of Android, providing one common interface without the need to constantly check the version of Android that the app is running on.

Requesting System Permissions

The first step in working with Android permissions is to declare the permissions in the Android manifest file. This must be done regardless of the API level that the app is targetting.

Apps that target Android 6.0 or higher cannot assume that because the user granted permission at some point in the past, that the permission will be valid the next time. An app that targets Android 6.0 must always perform a runtime permission check. Apps that target Android 5.1 or lower do not need to perform a run-time permission check.

> **NOTE**
>
> Applications should only request the permissions that they require.

Declaring Permissions in the Manifest

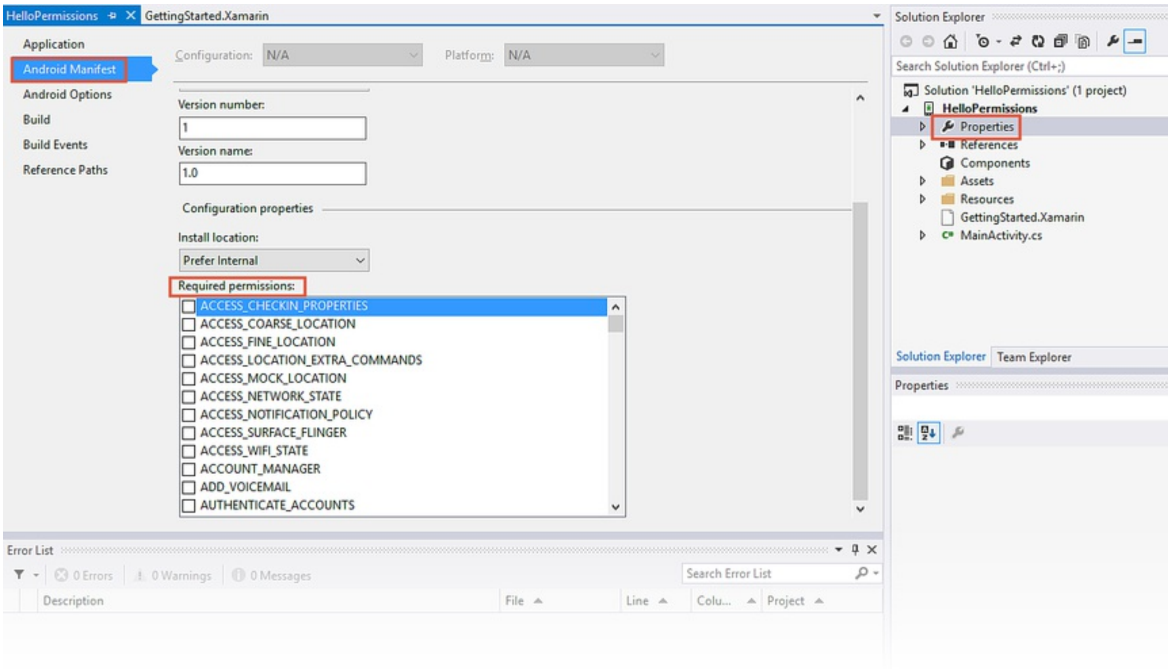
Permissions are added to the **AndroidManifest.xml** with the `uses-permission` element. For example, if an application is to locate the position of the device, it requires fine and course location permissions. The following two elements are added to the manifest:

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

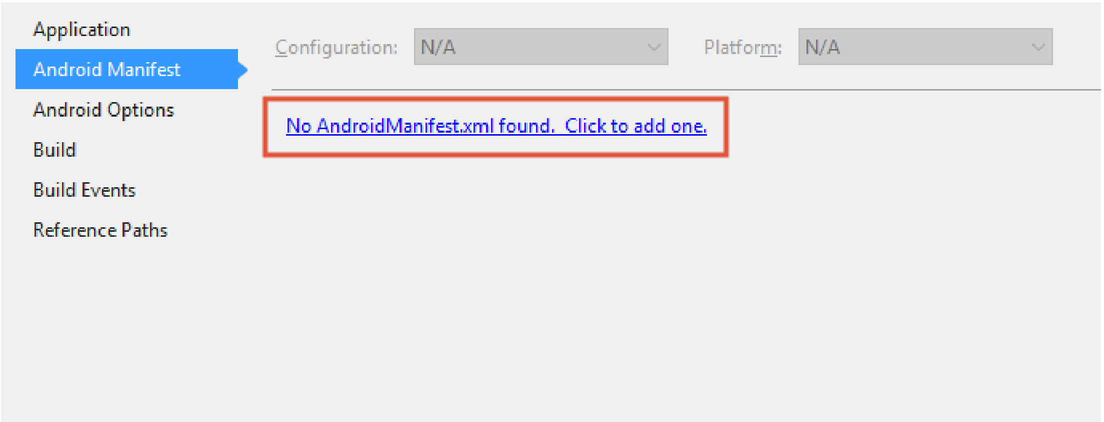
- Visual Studio
- Visual Studio for Mac

It is possible to declare the permissions using the tool support built into Visual Studio:

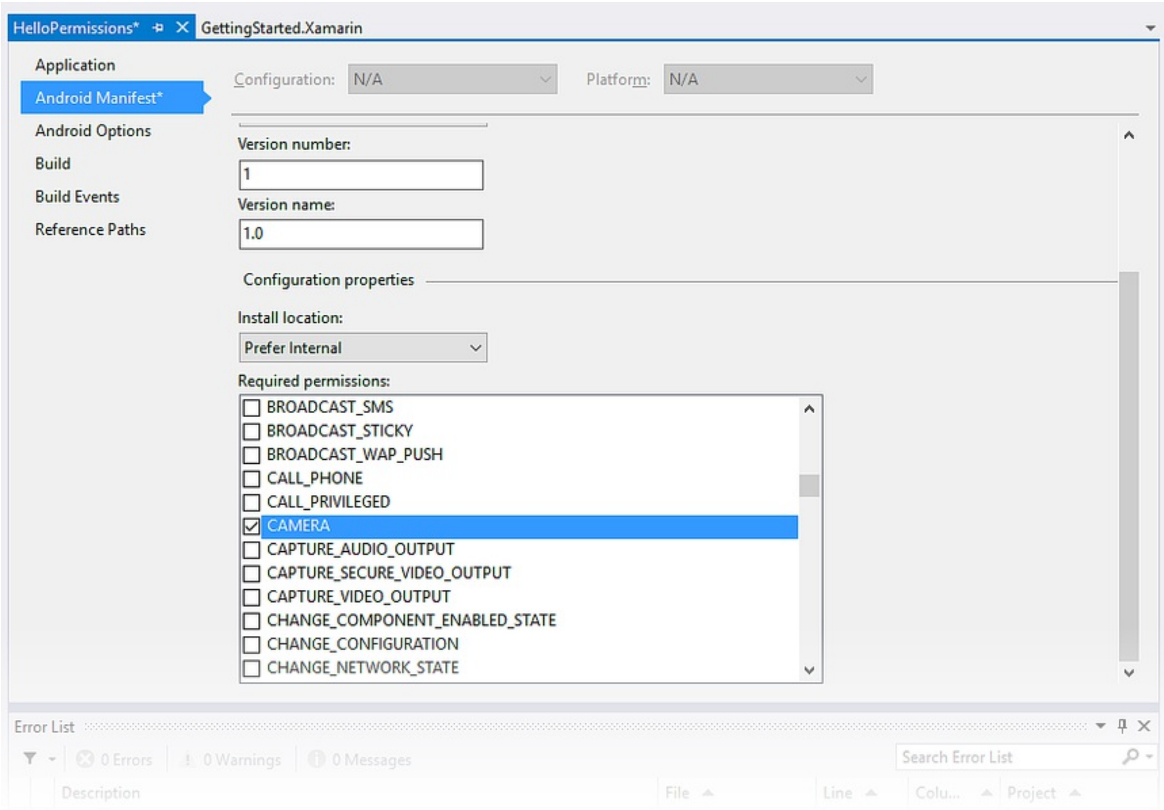
1. Double-click **Properties** in the **Solution Explorer** and select the **Android Manifest** tab in the Properties window:



2. If the application does not already have an AndroidManifest.xml, click **No AndroidManifest.xml found. Click to add one** as shown below:



3. Select any permissions your application needs from the **Required permissions** list and save:



Xamarin.Android will automatically add some permissions at build time to Debug builds. This will make debugging the application easier. In particular, two notable permissions are `INTERNET` and `READ_EXTERNAL_STORAGE`. These automatically-set permissions will not appear to be enabled in the **Required permissions** list. Release builds, however, use only the permissions that are explicitly set in the **Required permissions** list.

For apps that target Android 5.1(API level 22) or lower, there is nothing more that needs to be done. Apps that will run on Android 6.0 (API 23 level 23) or higher should proceed on to the next section on how to perform run time permission checks.

Runtime Permission Checks in Android 6.0

The `ContextCompat.CheckSelfPermission` method (available with the Android Support Library) is used to check if a specific permission has been granted. This method will return a `Android.Content.PM.Permission` enum which has one of two values:

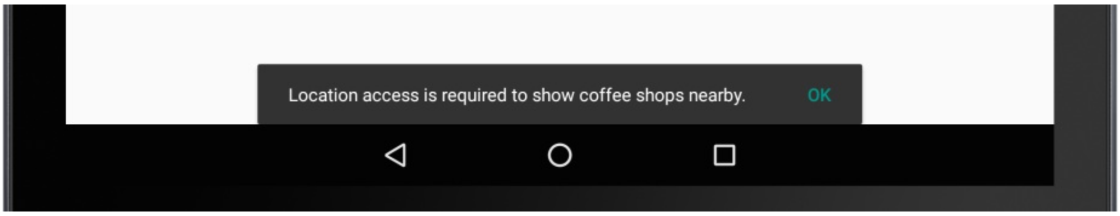
- `Permission.Granted` – The specified permission has been granted.
- `Permission.Denied` – The specified permission has not been granted.

This code snippet is an example of how to check for the Camera permission in an Activity:

```
if (ContextCompat.CheckSelfPermission(this, Manifest.Permission.Camera) == (int)Permission.Granted)
{
    // We have permission, go ahead and use the camera.
}
else
{
    // Camera permission is not granted. If necessary display rationale & request.
}
```

It is a best practice to inform the user as to why a permission is necessary for an application so that an informed decision can be made to grant the permission. An example of this would be an app that takes photos and geo-tags them. It is clear to the user that the camera permission is necessary, but it might not be clear why the app also needs the location of the device. The rationale should display a message to help the user understand why the location permission is desirable and that the camera permission is required.

The `ActivityCompat.ShouldShowRequestPermissionRationale` method is used to determine if the rationale should be shown to the user. This method will return `true` if the rationale for a given permission should be displayed. This screenshot shows an example of a Snackbar displayed by an application that explains why the app needs to know the location of the device:



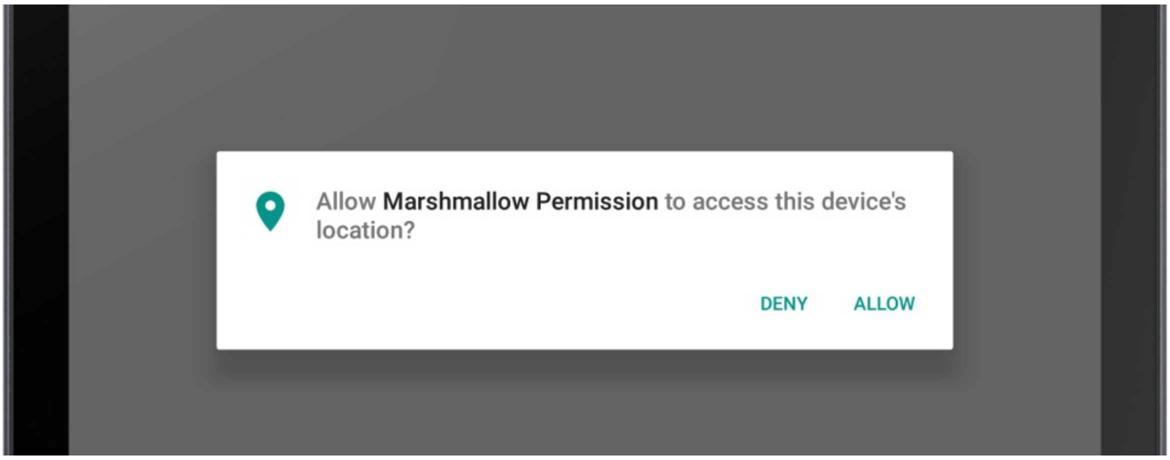
If the user grants the permission, the `ActivityCompat.RequestPermissions(Activity activity, string[] permissions, int requestCode)` method should be called. This method requires the following parameters:

- **activity** – This is the activity that is requesting the permissions and is to be informed by Android of the results.
- **permissions** – A list of the permissions that are being requested.
- **requestCode** – An integer value that is used to match the results of the permission request to a `RequestPermissions` call. This value should be greater than zero.

This code snippet is an example of the two methods that were discussed. First, a check is made to determine if the permission rationale should be shown. If the rationale is to be shown, then a Snackbar is displayed with the rationale. If the user clicks **OK** in the Snackbar, then the app will request the permissions. If the user does not accept the rationale, then the app should not proceed to request permissions. If the rationale is not shown, then the Activity will request the permission:

```
if (ActivityCompat.ShouldShowRequestPermissionRationale(this, Manifest.Permission.AccessFineLocation))
{
    // Provide an additional rationale to the user if the permission was not granted
    // and the user would benefit from additional context for the use of the permission.
    // For example if the user has previously denied the permission.
    Log.Info(TAG, "Displaying camera permission rationale to provide additional context.");

    var requiredPermissions = new String[] { Manifest.Permission.AccessFineLocation };
    Snackbar.Make(layout,
                  Resource.String.permission_location_rationale,
                  Snackbar.LengthIndefinite)
            .SetAction(Resource.String.ok,
                       new Action<View>(delegate(View obj) {
                           ActivityCompat.RequestPermissions(this, requiredPermissions, REQUEST_LOCATION);
                       }
            )
    ).Show();
}
else
{
    ActivityCompat.RequestPermissions(this, new String[] { Manifest.Permission.Camera }, REQUEST_LOCATION);
}
```

`RequestPermission` can be called even if the user has already granted permission. Subsequent calls are not necessary, but they provide the user with the opportunity to confirm (or revoke) the permission. When `RequestPermission` is called, control is handed off to the operating system, which will display a UI for accepting the permissions:



After the user is finished, Android will return the results to the Activity via a callback method, `OnRequestPermissionResult`. This method is a part of the interface `ActivityCompat.IOnRequestPermissionsResultCallback` which must be implemented by the Activity. This interface has a single method, `OnRequestPermissionsResult`, which will be invoked by Android to inform the Activity of the user's choices. If the user has granted the permission, then the app can go ahead and use the protected resource. An example of how to implement `OnRequestPermissionResult` is shown below:

```
public override void OnRequestPermissionsResult(int requestCode, string[] permissions, Permission[]
grantResults)
{
    if (requestCode == REQUEST_LOCATION)
    {
        // Received permission result for camera permission.
        Log.Info(TAG, "Received response for Location permission request.");

        // Check if the only required permission has been granted
        if ((grantResults.Length == 1) && (grantResults[0] == Permission.Granted)) {
            // Location permission has been granted, okay to retrieve the location of the device.
            Log.Info(TAG, "Location permission has now been granted.");
            Snackbar.Make(layout, Resource.String.permission_available_camera, Snackbar.LengthShort).Show();
        }
        else
        {
            Log.Info(TAG, "Location permission was NOT granted.");
            Snackbar.Make(layout, Resource.String.permissions_not_granted, Snackbar.LengthShort).Show();
        }
    }
    else
    {
        base.OnRequestPermissionsResult(requestCode, permissions, grantResults);
    }
}
```

Summary

This guide discussed how to add and check for permissions in an Android device. The differences in how permissions work between old Android apps (API level < 23) and new Android apps (API level > 22). It discussed how to perform run-time permission checks in Android 6.0.

Related Links

- List of Normal Permissions
- Runtime Permissions Sample App
- Handling Permissions in Xamarin.Android

Android Graphics and Animation

7/8/2021 • 17 minutes to read • Edit Online

Android provides a very rich and diverse framework for supporting 2D graphics and animations. This topic introduces these frameworks and discusses how to create custom graphics and animations for use in a Xamarin.Android application.

Overview

Despite running on devices that are traditionally of limited power, the highest rated mobile applications often have a sophisticated User Experience (UX), complete with high quality graphics and animations that provide an intuitive, responsive, dynamic feel. As mobile applications get more and more sophisticated, users have begun to expect more and more from applications.

Luckily for us, modern mobile platforms have very powerful frameworks for creating sophisticated animations and custom graphics while retaining ease of use. This enables developers to add rich interactivity with very little effort.

UI API frameworks in Android can roughly be split into two categories: Graphics and Animation.

Graphics are further split into different approaches for doing 2D and 3D graphics. 3D graphics are available via a number of built in frameworks such as OpenGL ES (a mobile specific version of OpenGL), and third-party frameworks such as MonoGame (a cross platform toolkit compatible with the XNA toolkit). Although 3D graphics are not within the scope of this article, we will examine the built-in 2D drawing techniques.

Android provides two different API's for creating 2D graphics. One is a high level declarative approach and the other a programmatic low-level API:

- **Drawable Resources** – These are used to create custom graphics either programmatically or (more typically) by embedding drawing instructions in XML files. Drawable resources are typically defined as XML files that contain instructions or actions for Android to render a 2D graphic.
- **Canvas** – this is a low level API that involves drawing directly on an underlying bitmap. It provides very fine-grained control over what is displayed.

In addition to these 2D graphics techniques, Android also provides several different ways to create animations:

- **Drawable Animations** – Android also supports frame-by-frame animations known as *Drawable Animation*. This is the simplest animation API. Android sequentially loads and displays Drawable resources in sequence (much like a cartoon).
- **View Animations** – *View Animations* are the original animation API's in Android and are available in all versions of Android. This API is limited in that it will only work with View objects and can only perform simple transformations on those Views. View animations are typically defined in XML files found in the `/Resources/anim` folder.
- **Property Animations** – Android 3.0 introduced a new set of animation API's known as *Property Animations*. These new API's introduced an extensible and flexible system that can be used to animate the properties of any object, not just View objects. This flexibility allows animations to be encapsulated in distinct classes that will make code sharing easier.

View Animations are more suitable for applications that must support the older pre-Android 3.0 API's (API level 11). Otherwise applications should use the newer Property Animation API's for the reasons that were mentioned above.

All of these frameworks are viable options, however where possible, preference should be given to Property Animations, as it is a more flexible API to work with. Property Animations allow for animation logic to be encapsulated in distinct classes that makes code sharing easier and simplifies code maintenance.

Accessibility

Graphics and animations help to make Android apps attractive and fun to use; however, it is important to remember that some interactions occur via screenreaders, alternate input devices, or with assisted zoom. Also, some interactions may occur without audio capabilities.

Apps are more usable in these situations if they have been designed with accessibility in mind: providing hints and navigation assistance in the user-interface, and ensuring there is text-content or descriptions for pictorial elements of the UI.

Refer to Google's Accessibility Guide for more information on how to utilize Android's accessibility APIs.

2D Graphics

Drawable Resources are a popular technique in Android applications. As with other resources, Drawable Resources are declarative – they're defined in XML files. This approach allows for a clean separation of code from resources. This can simplify development and maintenance because it is not necessary to change code to update or change the graphics in an Android application. However, while Drawable Resources are useful for many simple and common graphic requirements, they lack the power and control of the Canvas API.

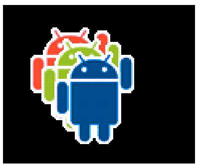
The other technique, using the Canvas object, is very similar to other traditional API frameworks such as System.Drawing or iOS's Core Drawing. Using the Canvas object provides the most control of how 2D graphics are created. It is appropriate for situations where a Drawable Resource will not work or will be difficult to work with. For example, it may be necessary to draw a custom slider control whose appearance will change based on calculations related to the value of the slider.

Let's examine Drawable Resources first. They are simpler and cover the most common custom drawing cases.

Drawable Resources

Drawable Resources are defined in an XML file in the directory `/Resources/drawable`. Unlike embedding PNG or JPEG's, it is not necessary to provide density-specific versions of Drawable Resources. At runtime, an Android application will load these resources and use the instructions contained in these XML files to create 2D graphics. Android defines several different types of Drawable Resources:

- ShapeDrawable – This is a Drawable object that draws a primitive geometric shape and applies a limited set of graphical effects on that shape. They are very useful for things such as customizing Buttons or setting the background of TextViews. We will see an example of how to use a `ShapeDrawable` later in this article.
- StateListDrawable – This is a Drawable Resource that will change appearance based on the state of a widget/control. For example, a button may change its appearance depending on whether it is pressed or not.
- LayerDrawable – This Drawable Resource that will stack several other drawables one on top of another. An example of a *LayerDrawable* is shown in the following screenshot:



- TransitionDrawable – This is a *LayerDrawable* but with one difference. A *TransitionDrawable* is able to animate one layer showing up over top another.
- LevelListDrawable – This is very similar to a *StateListDrawable* in that it will display an image based on certain conditions. However, unlike a *StateListDrawable*, the *LevelListDrawable* displays an image based on an integer value. An example of a *LevelListDrawable* would be to display the strength of a WiFi signal. As the strength of the WiFi signal changes, the drawable that is displayed will change accordingly.
- ScaleDrawable/ClipDrawable – As their name implies, these Drawables provide both scaling and clipping functionality. The *ScaleDrawable* will scale another Drawable, while the *ClipDrawable* will clip another Drawable.
- InsetDrawable – This Drawable will apply insets on the sides of another Drawable resource. It is used when a View needs a background that is smaller than the View's actual bounds.
- XML BitmapDrawable – This file is a set of instructions, in XML, that are to be performed on an actual bitmap. Some actions that Android can perform are tiling, dithering, and anti-aliasing. One of the very common uses of this is to tile a bitmap across the background of a layout.

Drawable Example

Let's look at a quick example of how to create a 2D graphic using a `ShapeDrawable`. A `ShapeDrawable` can define one of the four basic shapes: rectangle, oval, line, and ring. It is also possible to apply basic effects, such as gradient, colour, and size. The following XML is a `ShapeDrawable` that may be found in the *AnimationsDemo* companion project (in the file `Resources/drawable/shape_rounded_blue_rect.xml`). It defines a rectangle with a purple gradient background and rounded corners:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android" android:shape="rectangle">
<!-- Specify a gradient for the background -->
<gradient android:angle="45"
          android:startColor="#55000066"
          android:centerColor="#00000000"
          android:endColor="#00000000"
          android:centerX="0.75" />

<padding android:left="5dp"
          android:right="5dp"
          android:top="5dp"
          android:bottom="5dp" />

<corners android:topLeftRadius="10dp"
          android:topRightRadius="10dp"
          android:bottomLeftRadius="10dp"
          android:bottomRightRadius="10dp" />
</shape>
```

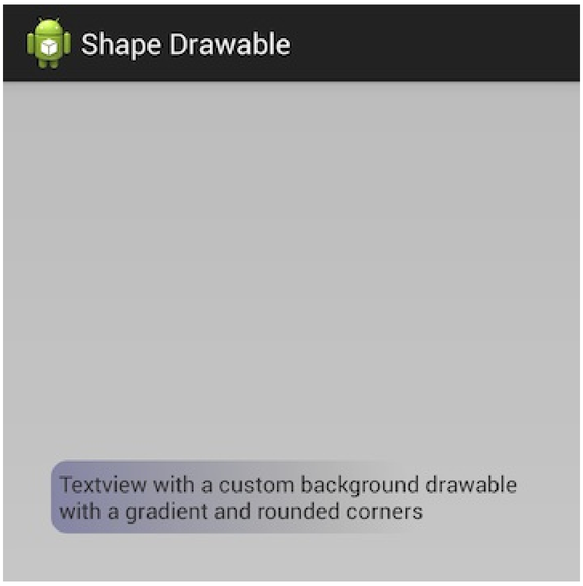
We can reference this Drawable Resource declaratively in a Layout or other Drawable as shown in the following XML:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#33000000">
    <TextView android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:layout_centerInParent="true"
              android:background="@drawable/shape_rounded_blue_rect"
              android:text="@string/message_shapedrawable" />
</RelativeLayout>
```

Drawable Resources can also be applied programmatically. The following code snippet shows how to programmatically set the background of a TextView:

```
TextView tv = FindViewById<TextView>(Resource.Id.shapeDrawableTextView);
tv.SetBackgroundResource(Resource.Drawable.shape_rounded_blue_rect);
```

To see what this would look like, run the *AnimationsDemo* project and select the Shape Drawable item from the main menu. We should see something similar to the following screenshot:



For more details about the XML elements and syntax of Drawable Resources, consult Google's documentation.

Using the Canvas Drawing API

Drawables are powerful but have their limitations. Certain things are either not possible or very complex (for example: applying a filter to a picture that was taken by a camera on the device). It would be very difficult to apply red-eye reduction by using a Drawable Resource. Instead, the Canvas API allows an application to have very fine-grained control to selectively change colors in a specific part of the picture.

One class that is commonly used with the Canvas is the Paint class. This class holds colour and style information about how to draw. It is used to provide things such a color and transparency.

The Canvas API uses the *painter's model* to draw 2D graphics. Operations are applied in successive layers on top of each other. Each operation will cover some area of the underlying bitmap. When the area overlaps a previously painted area, the new paint will partially or completely obscure the old. This is the same way that many other drawing APIs such as System.Drawing and iOS's Core Graphics work.

There are two ways to obtain a `Canvas` object. The first way involves defining a Bitmap object, and then instantiating a `Canvas` object with it. For example, the following code snippet creates a new canvas with an underlying bitmap:

```
Bitmap bitmap = Bitmap.CreateBitmap(100, 100, Bitmap.Config.Argb8888);
Canvas canvas = new Canvas(b);
```

The other way to obtain a `Canvas` object is by the OnDraw callback method that is provided the View base class. Android calls this method when it decides a View needs to draw itself and passes in a `Canvas` object for the View to work with.

The Canvas class exposes methods to programmatically provide the draw instructions. For example:

- Canvas.DrawPaint – Fills the entire canvas's bitmap with the specified paint.

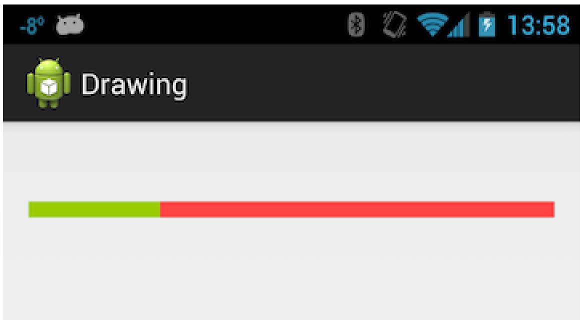
- Canvas.DrawPath – Draws the specified geometric shape using the specified paint.
- Canvas.DrawText – Draws the text on the canvas with the specified colour. The text is drawn at location `x,y` .

Drawing with the Canvas API

Here's an example of the Canvas API in action. The following code snippet shows how to draw a view:

```
public class MyView : View
{
    protected override void OnDraw(Canvas canvas)
    {
        base.OnDraw(canvas);
        Paint green = new Paint {
            AntiAlias = true,
            Color = Color.Rgb(0x99, 0xcc, 0),
        };
        green.SetStyle(Paint.Style.FillAndStroke);

        Paint red = new Paint {
            AntiAlias = true,
            Color = Color.Rgb(0xff, 0x44, 0x44)
        };
        red.SetStyle(Paint.Style.FillAndStroke);

        float middle = canvas.Width * 0.25f;
        canvas.DrawPaint(red);
        canvas.DrawRect(0, 0, middle, canvas.Height, green);
    }
}
```

This code above first creates a red paint and a green paint object. It fills the content of the canvas with red, and then instructs the canvas to draw a green rectangle that is 25% of the width of the canvas. An example of this can be seen by in `AnimationsDemo` project that is included with the source code for this article. By starting up the application and selecting the Drawing item from the main menu, we should a screen similar to the following:



Animation

Users like things that move in their applications. Animations are a great way to improve the user experience of an application and help it stand out. The best animations are the ones that users don't notice because they feel natural. Android provides the following three API's for animations:

- **View Animation** – This is the original API. These animations are tied to a specific View and can perform simple transformations on the contents of the View. Because of it's simplicity, this API still useful for things like alpha animations, rotations, and so forth.
- **Property Animation** – Property animations were introduced in Android 3.0. They enable an application to animate almost anything. Property animations can be used to change any property of any object, even if that object is not visible on the screen.

- **Drawable Animation** – This a special Drawable resource that is used to apply a very simple animation effect to layouts.

In general, property animation is the preferred system to use as it is more flexible and offers more features.

View Animations

View animations are limited to Views and can only perform animations on values such as start and end points, size, rotation, and transparency. These types of animations are typically referred to as *tween animations*. View animations can be defined two ways – programmatically in code or by using XML files. XML files are the preferred way to declare view animations, as they are more readable and easier to maintain.

The animation XML files will be stored in the `/Resources/anim` directory of a Xamarin.Android project. This file must have one of the following elements as the root element :

- `alpha` – A fade-in or fade-out animation.
- `rotate` – A rotation animation.
- `scale` – A resizing animation.
- `translate` – A horizontal and/or vertical motion.
- `set` – A container that may hold one or more of the other animation elements.

By default, all animations in an XML file will be applied simultaneously. To make animations occur sequentially, set the `android:startOffset` attribute on one of the elements defined above.

It is possible to affect the rate of change in an animation by using an *interpolator*. An interpolator makes it possible for animation effects to be accelerated, repeated, or decelerated. The Android framework provides several interpolators out of the box, such as (but not limited to):

- `AccelerateInterpolator` / `DecelerateInterpolator` – these interpolators increase or decrease the rate of change in an animation.
- `BounceInterpolator` – the change bounces at the end.
- `LinearInterpolator` – the rate of changes is constant.

The following XML shows an example of an animation file that combines some of these elements:

```xml
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android=http://schemas.android.com/apk/res/android
     android:shareInterpolator="false">

    <scale android:interpolator="@android:anim/accelerate_decelerate_interpolator"
           android:fromXScale="1.0"
           android:toXScale="1.4"
           android:fromYScale="1.0"
           android:toYScale="0.6"
           android:pivotX="50%"
           android:pivotY="50%"
           android:fillEnabled="true"
           android:fillAfter="false"
           android:duration="700" />

    <set android:interpolator="@android:anim/accelerate_interpolator">
        <scale android:fromXScale="1.4"
               android:toXScale="0.0"
               android:fromYScale="0.6"
               android:toYScale="0.0"
               android:pivotX="50%"
               android:pivotY="50%"
               android:fillEnabled="true"
               android:fillBefore="false"
               android:fillAfter="true"
               android:startOffset="700"
               android:duration="400" />

        <rotate android:fromDegrees="0"
                android:toDegrees="-45"
                android:toYScale="0.0"
                android:pivotX="50%"
                android:pivotY="50%"
                android:fillEnabled="true"
                android:fillBefore="false"
                android:fillAfter="true"
                android:startOffset="700"
                android:duration="400" />
    </set>
</set>
```

This animation will perform all of the animations simultaneously. The first scale animation will stretch the image horizontally and shrink it vertically, and then the image will simultaneously be rotated 45 degrees counter-clockwise and shrink, disappearing from the screen.

The animation can be programmatically applied to a View by inflating the animation and then applying it to a View. Android provides the helper class `Android.Views.Animations.AnimationUtils` that will inflate an animation resource and return an instance of `Android.Views.Animations.Animation`. This object is applied to a View by calling `StartAnimation` and passing the `Animation` object. The following code snippet shows an example of this:

```csharp
Animation myAnimation = AnimationUtils.LoadAnimation(Resource.Animation.MyAnimation);
ImageView myImage = FindViewById<ImageView>(Resource.Id.imageView1);
myImage.StartAnimation(myAnimation);
```

Now that we have a fundamental understanding of how View Animations work, lets move to Property Animations.

Property Animations

Property animators are a new API that was introduced in Android 3.0. They provide a more extensible API that can be used to animate any property on any object.

All property animations are created by instances of the Animator subclass. Applications do not directly use this class, instead they use one of it's subclasses:

- ValueAnimator – This class is the most important class in the entire property animation API. It calculates the values of properties that need to be changed. The `ViewAnimator` does not directly update those values; instead, it raises events that can be used to update animated objects.
- ObjectAnimator – This class is a subclass of `ValueAnimator` . It is meant to simplify the process of animating objects by accepting a target object and property to update.
- AnimationSet – This class is responsible for orchestrating how animations run in relation to one another. Animations may run simultaneously, sequentially, or with a specified delay between them.

Evaluators are special classes that are used by animators to calculate the new values during an animation. Out of the box, Android provides the following evaluators:

- IntEvaluator – Calculates values for integer properties.
- FloatEvaluator – Calculates values for float properties.
- ArgbEvaluator – Calculates values for colour properties.

If the property that is being animated is not a `float` , `int` or colour, applications may create their own evaluator by implementing the `ITypeEvaluator` interface. (Implementing custom evaluators is beyond the scope of this topic.)

Using the ValueAnimator

There are two parts to any animation: calculating animated values and then setting those values on properties on some object. ValueAnimator will only calculate the values, but it will not operate on objects directly. Instead, objects will be updated inside event handlers that will be invoked during the animation lifespan. This design allows several properties to be updated from one animated value.

You obtain an instance of `ValueAnimator` by calling one of the following factory methods:

- `ValueAnimator.OfInt`
- `ValueAnimator.OfFloat`
- `ValueAnimator.OfObject`

Once that is done, the `ValueAnimator` instance must have its duration set, and then it can be started. The following example shows how to animate a value from 0 to 1 over the span of 1000 milliseconds:

```
ValueAnimator animator = ValueAnimator.OfInt(0, 100);
animator.SetDuration(1000);
animator.Start();
```

But itself, the code snippet above is not very useful – the animator will run but there is no target for the updated value. The `Animator` class will raise the Update event when it decides that it is necessary to inform listeners of a new value. Applications may provide an event handler to respond to this event as shown in the following code snippet:

```
MyCustomObject myObj = new MyCustomObject();
myObj.SomeIntegerValue = -1;

animator.Update += (object sender, ValueAnimator.AnimatorUpdateEventArgs e) =>
{
    int newValue = (int) e.Animation.AnimatedValue;
    // Apply this new value to the object being animated.
    myObj.SomeIntegerValue = newValue;
};
```

Now that we have an understanding of `ValueAnimator`, lets learn more about the `ObjectAnimator`.

Using the ObjectAnimator

ObjectAnimator is a subclass of `ViewAnimator` that combines the timing engine and value computation of the `ValueAnimator` with the logic required to wire up event handlers. The `ValueAnimator` requires applications to explicitly wire up an event handler – `ObjectAnimator` will take care of this step for us.

The API for `ObjectAnimator` is very similar to the API for `ViewAnimator`, but requires that you provide the object and the name of the property to update. The following example shows an example of using `ObjectAnimator`:

```
MyCustomObject myObj = new MyCustomObject();
myObj.SomeIntegerValue = -1;

ObjectAnimator animator = ObjectAnimator.OfFloat(myObj, "SomeIntegerValue", 0, 100);
animator.SetDuration(1000);
animator.Start();
```

As you can see from the previous code snippet, `ObjectAnimator` can reduce and simplify the code that is necessary to animate an object.

Drawable Animations

The final animation API is the Drawable Animation API. Drawable animations load a series of Drawable resources one after the other and display them sequentially, similar to a flip-it cartoon.

Drawable resources are defined in an XML file that has an `<animation-list>` element as the root element and a series of `<item>` elements that define each frame in the animation. This XML file is stored in the `/Resource/drawable` folder of the application. The following XML is an example of a drawable animation:

```
<animation-list xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:drawable="@drawable/asteroid01" android:duration="100" />
  <item android:drawable="@drawable/asteroid02" android:duration="100" />
  <item android:drawable="@drawable/asteroid03" android:duration="100" />
  <item android:drawable="@drawable/asteroid04" android:duration="100" />
  <item android:drawable="@drawable/asteroid05" android:duration="100" />
  <item android:drawable="@drawable/asteroid06" android:duration="100" />
</animation-list>
```

This animation will run through six frames. The `android:duration` attribute declares how long each frame will be displayed. The next code snippet shows an example of creating a Drawable animation and starting it when the user clicks a button on the screen:

```
AnimationDrawable _asteroidDrawable;

protected override void OnCreate(Bundle bundle)
{
    base.OnCreate(bundle);
    SetContentView(Resource.Layout.Main);

    _asteroidDrawable = (Android.Graphics.Drawables.AnimationDrawable)
    Resources.GetDrawable(Resource.Drawable.spinning_asteroid);

    ImageView asteroidImage = FindViewById<ImageView>(Resource.Id.imageView2);
    asteroidImage.SetImageDrawable((Android.Graphics.Drawables.Drawable) _asteroidDrawable);

    Button asteroidButton = FindViewById<Button>(Resource.Id.spinAsteroid);
    asteroidButton.Click += (sender, e) =>
    {
        _asteroidDrawable.Start();
    };
}
```

At this point we have covered the foundations of the animation APIs available in an Android application.

Summary

This article introduced a lot of new concepts and API's to help add some graphics to an Android application. First it discussed the various 2D graphics API's and demonstrated how Android allows applications to draw directly to the screen using a Canvas object. We also saw some alternate techniques that allow graphics to be declaratively created using XML files. Then we went on to discuss the old and new API's for creating animations in Android.

Related Links

- Animation Demo (sample)
- Animation and Graphics
- Using Animations to Bring your Mobile Apps to Life
- AnimationDrawable
- Canvas
- Object Animator
- Value Animator

CPU Architectures

7/8/2021 • 3 minutes to read • Edit Online

Xamarin.Android supports several CPU architectures, including 32-bit and 64-bit devices. This article explains how to target an app to one or more Android-supported CPU architectures.

CPU Architectures Overview

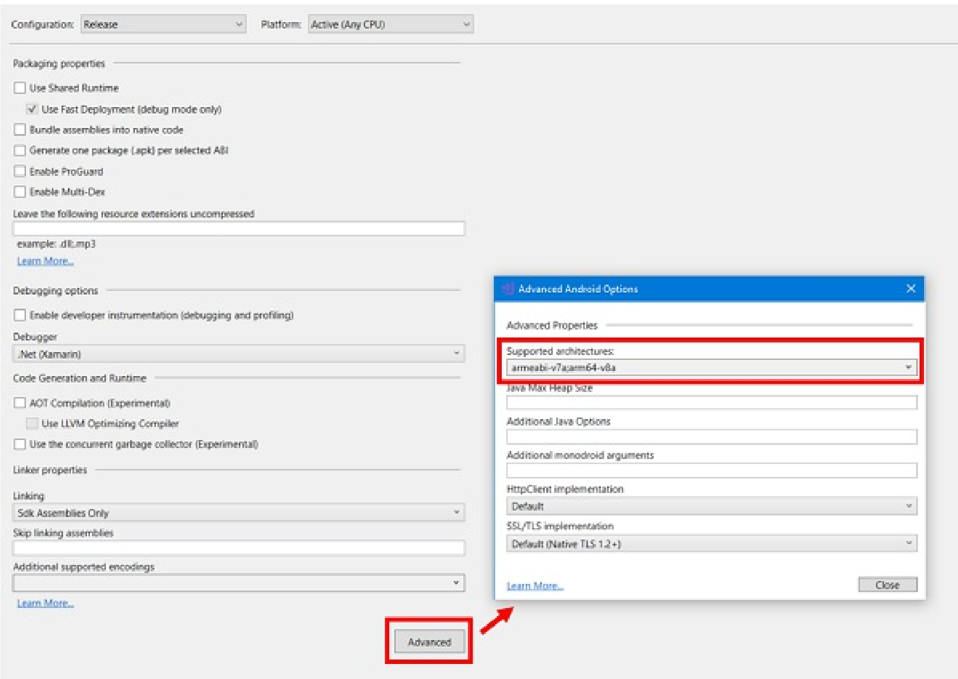
When you prepare your app for release, you must specify which platform CPU architectures your app supports. A single APK can contain machine code to support multiple, different architectures. Each collection of architecture-specific code is associated with an *Application Binary Interface* (ABI). Each ABI defines how this machine code is expected to interact with Android at run time. For more information about how this works, see Multi-Core Devices & Xamarin.Android.

How to Specify Supported Architectures

- Visual Studio
- Visual Studio for Mac

Typically, you explicitly select an architecture (or architectures) when your app is configured for **Release**. When your app is configured for **Debug**, the **Use Shared Runtime** and **Use Fast Deployment** options are enabled, which disable explicit architecture selection.

In Visual Studio, right-click on your project under the **Solution Explorer** and select **Properties**. Under the **Android Options** page check the **Packaging properties** section and verify that **Use Shared Runtime** is disabled (turning this off allows you to explicitly select which ABIs to support). Click the **Advanced** button and, under **Supported architectures**, check the architectures that you want to support:



Xamarin.Android supports the following architectures:

- **armeabi** – ARM-based CPUs that support at least the ARMv5TE instruction set. Note that `armeabi` is not thread-safe and should not be used on multi-CPU devices.

> **NOTE**
>
> As of Xamarin.Android 9.2, `armeabi` is no longer supported.

- **armeabi-v7a** – ARM-based CPUs with hardware floating-point operations and multiple CPU (SMP) devices. Note that `armeabi-v7a` machine code will not run on ARMv5 devices.
- **arm64-v8a** – CPUs based on the 64-bit ARMv8 architecture.
- **x86** – CPUs that support the x86 (or IA-32) instruction set. This instruction set is equivalent to that of the Pentium Pro, including MMX, SSE, SSE2, and SSE3 instructions.
- **x86_64** CPUs that support the 64-bit x86 (also referred as *x64* and *AMD64*) instruction set.

Xamarin.Android defaults to `armeabi-v7a` for **Release** builds. This setting provides significantly better performance than `armeabi`. If you are targeting a 64-bit ARM platform (such as the Nexus 9), select `arm64-v8a`. If you are deploying your app to an x86 device, select `x86`. If the target x86 device uses a 64-bit CPU architecture, select `x86_64`.

Targeting Multiple Platforms

To target multiple CPU architectures, you can select more than one ABI (at the expense of larger APK file size). You can use the **Generate one package (.apk) per selected ABI** option (described in Set Packaging Properties) to create a separate APK for each supported architecture.

You do not have to select **arm64-v8a** or **x86_64** to target 64-bit devices; 64-bit support is not required to run your app on 64-bit hardware. For example, 64-bit ARM devices (such as the Nexus 9) can run apps configured for `armeabi-v7a`. The primary advantage of enabling 64-bit support is to make it possible for your app to address more memory.

> **NOTE**
>
> From August 2018 new apps will be required to target API level 26, and from August 2019 apps will be required to provide 64-bit versions in addition to the 32-bit version.

Additional Information

In some situations, you may need to create a separate APK for each architecture (to reduce the size of your APK, or because your app has shared libraries that are specific to a particular CPU architecture). For more information about this approach, see Build ABI-Specific APKs.

Handling Rotation

7/8/2021 • 8 minutes to read • Edit Online

This topic describes how to handle device orientation changes in Xamarin.Android. It covers how to work with the Android resource system to automatically load resources for a particular device orientation as well as how to programmatically handle orientation changes.

Overview

Because mobile devices are easily rotated, built-in rotation is a standard feature in mobile OSes. Android provides a sophisticated framework for dealing with rotation within applications, whether the user interface is created declaratively in XML or programmatically in code. When automatically handling declarative layout changes on a rotated device, an application can benefit from the tight integration with the Android resource system. For programmatic layout, changes must be handled manually. This allows finer control at runtime, but at the expense of more work for the developer. An application can also choose to opt out of the Activity restart and take manual control of orientation changes.

This guide examines the following orientation topics:

- **Declarative Layout Rotation** – How to use the Android resource system to build orientation-aware applications, including how to load both layouts and drawables for particular orientations.
- **Programmatic Layout Rotation** – How to add controls programmatically as well as how to handle orientation changes manually.

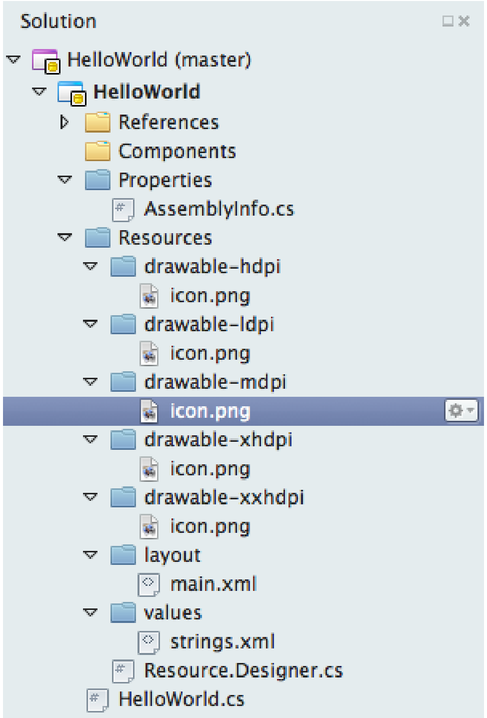
Handling Rotation Declaratively with Layouts

By including files in folders that follow naming conventions, Android automatically loads the appropriate files when the orientation changes. This includes support for:

- *Layout Resources* – Specifying which layout files are inflated for each orientation.
- *Drawable Resources* – Specifying which drawables are loaded for each orientation.

Layout Resources

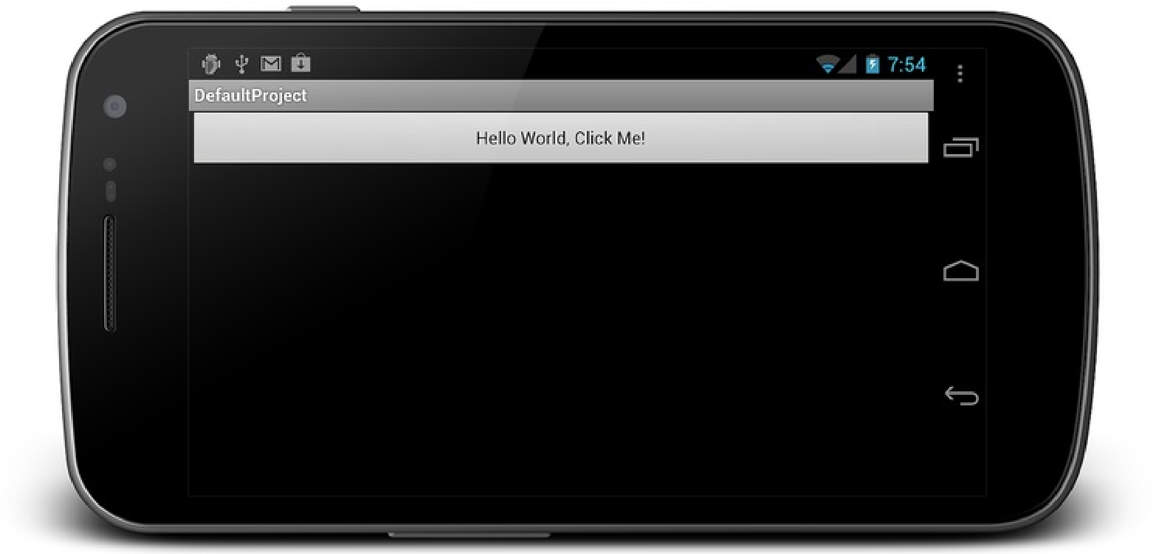
By default, Android XML (AXML) files included in the **Resources/layout** folder are used for rendering views for an Activity. This folder's resources are used for both portrait and landscape orientation if no additional layout resources are provided specifically for landscape. Consider the project structure created by the default project template:



This project creates a single **Main.axml** file in the **Resources/layout** folder. When the Activity's `OnCreate` method is called, it inflates the view defined in **Main.axml,** which declares a button as shown in the XML below:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
<Button
  android:id="@+id/myButton"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:text="@string/hello"/>
</LinearLayout>
```

If the device is rotated to landscape orientation, the Activity's `OnCreate` method is called again and the same **Main.axml** file is inflated, as shown in the screenshot below:



Orientation-Specific Layouts

In addition to the layout folder (which defaults to portrait and can also be explicitly named *layout-port* by including a folder named `layout-land`), an application can define the views it needs when in landscape without any code changes.

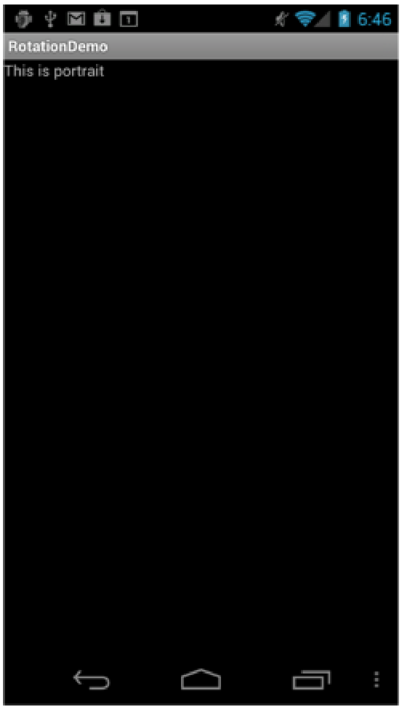
Suppose the **Main.axml** file contained the following XML:

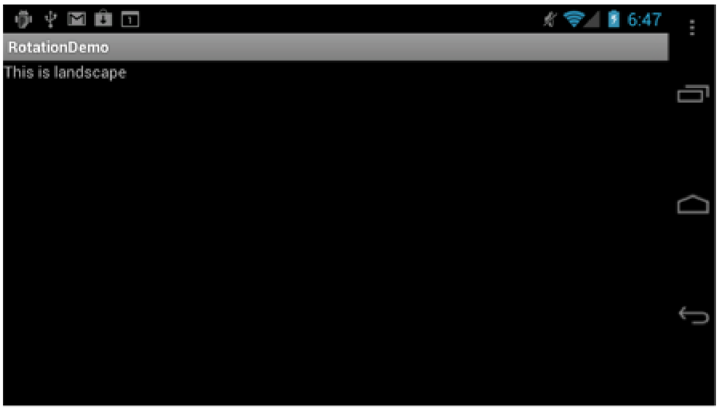
```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TextView
    android:text="This is portrait"
    android:layout_height="wrap_content"
    android:layout_width="fill_parent" />
</RelativeLayout>
```

If a folder named layout-land that contains an additional **Main.axml** file is added to the project, inflating the layout when in landscape will now result in Android loading the newly added **Main.axml**. Consider the landscape version of the **Main.axml** file that contains the following code (for simplicity, this XML is similar to the default portrait version of the code, but uses a different string in the `TextView`):

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TextView
    android:text="This is landscape"
    android:layout_height="wrap_content"
    android:layout_width="fill_parent" />
</RelativeLayout>
```

Running this code and rotating the device from portrait to landscape demonstrates the new XML loading, as shown below:





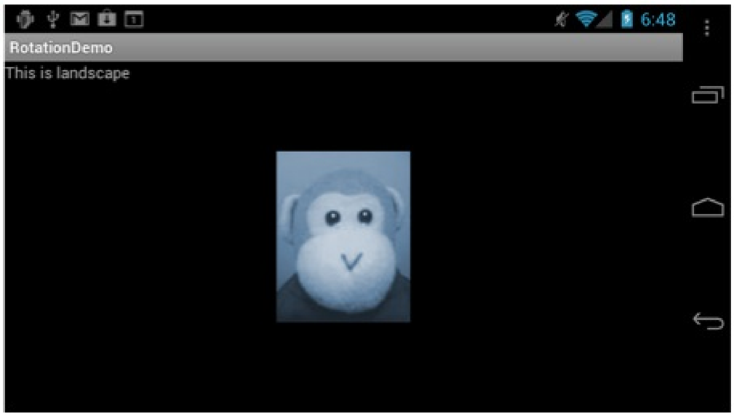
Drawable Resources

During rotation, Android treats drawable resources similarly to layout resources. In this case, the system gets the drawables from the **Resources/drawable** and **Resources/drawable-land** folders, respectively.

For example, say the project includes an image named Monkey.png in the **Resources/drawable** folder, where the drawable is referenced from an `ImageView` in XML like this:

```
<ImageView
  android:layout_height="wrap_content"
  android:layout_width="wrap_content"
  android:src="@drawable/monkey"
  android:layout_centerVertical="true"
  android:layout_centerHorizontal="true" />
```

Let's further assume that a different version of **Monkey.png** is included under **Resources/drawable-land**. Just like with the layout files, when the device is rotated, the drawable changes for the given orientation, as shown below:



Handling Rotation Programmatically

Sometimes we define layouts in code. This can happen for a variety of reasons, including technical limitations, developer preference, etc. When we add controls programmatically, an application must manually account for device orientation, which is handled automatically when we use XML resources.

Adding Controls in Code

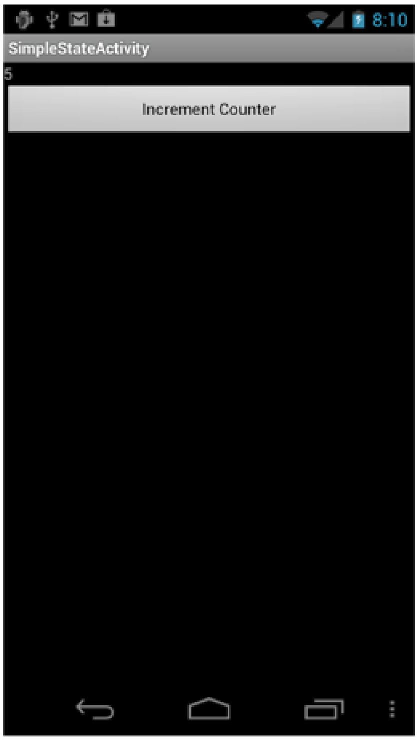
To add controls programmatically, an application needs to perform the following steps:

- Create a layout.
- Set layout parameters.
- Create controls.
- Set control layout parameters.
- Add controls to the layout.
- Set the layout as the content view.

For example, consider a user interface consisting of a single `TextView` control added to a `RelativeLayout`, as shown in the following code.

```
protected override void OnCreate (Bundle bundle)
{
  base.OnCreate (bundle);

  // create a layout
  var rl = new RelativeLayout (this);

  // set layout parameters
  var layoutParams = new RelativeLayout.LayoutParams (ViewGroup.LayoutParams.FillParent,
ViewGroup.LayoutParams.FillParent);
  rl.LayoutParameters = layoutParams;

  // create TextView control
  var tv = new TextView (this);

  // set TextView's LayoutParameters
  tv.LayoutParameters = layoutParams;
  tv.Text = "Programmatic layout";

  // add TextView to the layout
  rl.AddView (tv);

  // set the layout as the content view
  SetContentView (rl);
}
```

This code creates an instance of a `RelativeLayout` class and sets its `LayoutParameters` property. The `LayoutParams` class is Android's way of encapsulating how controls are positioned in a reusable way. Once an instance of a layout is created, controls can be created and added to it. Controls also have `LayoutParameters`, such as the `TextView` in this example. After the `TextView` is created, adding it to the `RelativeLayout` and setting the `RelativeLayout` as the content view results in the application displaying the `TextView` as shown:

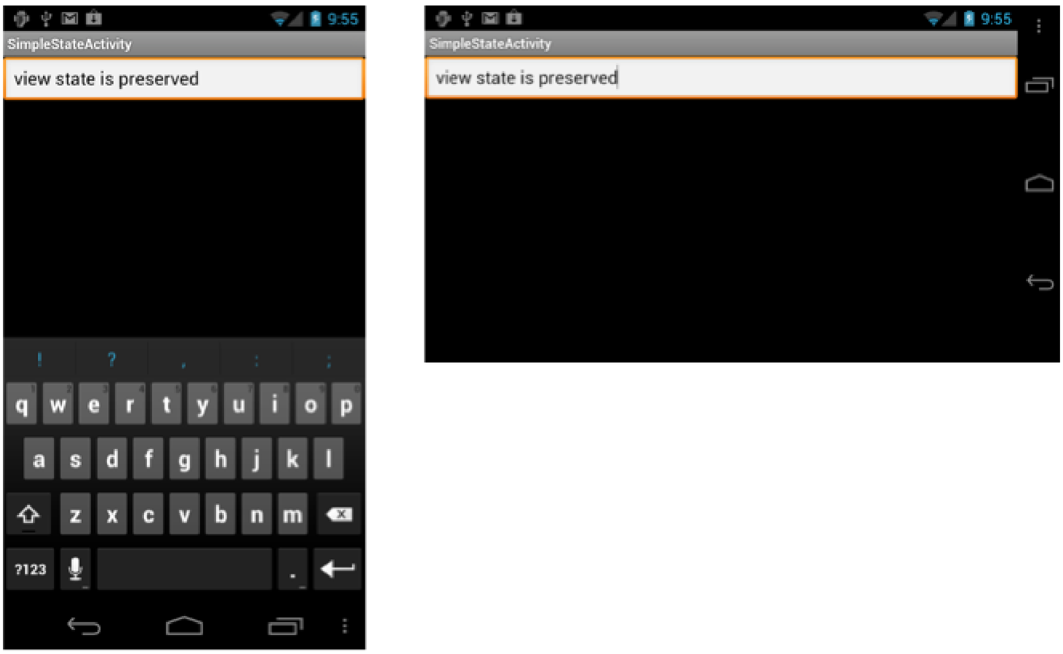


Detecting Orientation in Code

If an application tries to load a different user interface for each orientation when `OnCreate` is called (this will happen each time a device is rotated), it must detect the orientation, and then load the desired user interface code. Android has a class called the `WindowManager`, which can be used to determine the current device rotation via the `WindowManager.DefaultDisplay.Rotation` property, as shown below:

```
protected override void OnCreate (Bundle bundle)
{
  base.OnCreate (bundle);

  // create a layout
  var rl = new RelativeLayout (this);

  // set layout parameters
  var layoutParams = new RelativeLayout.LayoutParams (ViewGroup.LayoutParams.FillParent,
ViewGroup.LayoutParams.FillParent);
  rl.LayoutParameters = layoutParams;

  // get the initial orientation
  var surfaceOrientation = WindowManager.DefaultDisplay.Rotation;
  // create layout based upon orientation
  RelativeLayout.LayoutParams tvLayoutParams;

  if (surfaceOrientation == SurfaceOrientation.Rotation0 || surfaceOrientation ==
SurfaceOrientation.Rotation180) {
    tvLayoutParams = new RelativeLayout.LayoutParams (ViewGroup.LayoutParams.FillParent,
ViewGroup.LayoutParams.WrapContent);
  } else {
    tvLayoutParams = new RelativeLayout.LayoutParams (ViewGroup.LayoutParams.FillParent,
ViewGroup.LayoutParams.WrapContent);
    tvLayoutParams.LeftMargin = 100;
    tvLayoutParams.TopMargin = 100;
  }

  // create TextView control
  var tv = new TextView (this);
  tv.LayoutParameters = tvLayoutParams;
  tv.Text = "Programmatic layout";

  // add TextView to the layout
  rl.AddView (tv);

  // set the layout as the content view
  SetContentView (rl);
}
```

This code sets the `TextView` to be positioned 100 pixels from the top left of the screen, automatically animating to the new layout, when rotated to landscape, as shown here:



Preventing Activity Restart

In addition to handling everything in `OnCreate`, an application can also prevent an Activity from being restarted when the orientation changes by setting `ConfigurationChanges` in the `ActivityAttribute` as follows:

```
[Activity (Label = "CodeLayoutActivity", ConfigurationChanges=Android.Content.PM.ConfigChanges.Orientation |
Android.Content.PM.ConfigChanges.ScreenSize)]
```

Now when the device is rotated, the Activity is not restarted. In order to manually handle the orientation change in this case, an Activity can override the `OnConfigurationChanged` method and determine the orientation from the `Configuration` object that is passed in, as in the new implementation of the Activity below:

```
[Activity (Label = "CodeLayoutActivity", ConfigurationChanges=Android.Content.PM.ConfigChanges.Orientation |
Android.Content.PM.ConfigChanges.ScreenSize)]
public class CodeLayoutActivity : Activity
{
  TextView _tv;
  RelativeLayout.LayoutParams _layoutParamsPortrait;
  RelativeLayout.LayoutParams _layoutParamsLandscape;

  protected override void OnCreate (Bundle bundle)
  {
    // create a layout
    // set layout parameters
    // get the initial orientation

    // create portrait and landscape layout for the TextView
    _layoutParamsPortrait = new RelativeLayout.LayoutParams (ViewGroup.LayoutParams.FillParent,
ViewGroup.LayoutParams.WrapContent);

    _layoutParamsLandscape = new RelativeLayout.LayoutParams (ViewGroup.LayoutParams.FillParent,
ViewGroup.LayoutParams.WrapContent);
    _layoutParamsLandscape.LeftMargin = 100;
    _layoutParamsLandscape.TopMargin = 100;

    _tv = new TextView (this);

    if (surfaceOrientation == SurfaceOrientation.Rotation0 || surfaceOrientation ==
SurfaceOrientation.Rotation180) {
      _tv.LayoutParameters = _layoutParamsPortrait;
    } else {
      _tv.LayoutParameters = _layoutParamsLandscape;
    }

    _tv.Text = "Programmatic layout";
    rl.AddView (_tv);
    SetContentView (rl);
  }

  public override void OnConfigurationChanged (Android.Content.Res.Configuration newConfig)
  {
    base.OnConfigurationChanged (newConfig);

    if (newConfig.Orientation == Android.Content.Res.Orientation.Portrait) {
      _tv.LayoutParameters = _layoutParamsPortrait;
      _tv.Text = "Changed to portrait";
    } else if (newConfig.Orientation == Android.Content.Res.Orientation.Landscape) {
      _tv.LayoutParameters = _layoutParamsLandscape;
      _tv.Text = "Changed to landscape";
    }
  }
}
```

Here the `TextView's` layout parameters are initialized for both landscape and portrait. Class variables hold the parameters, along with the `TextView` itself, since the Activity will not be re-created when orientation changes. The code still uses the `surfaceOrientartion` in `OnCreate` to set the initial layout for the `TextView`. After that, `OnConfigurationChanged` handles all subsequent layout changes.

When we run the application, Android loads the user interface changes as device rotation occurs, and does not restart the Activity.

Preventing Activity Restart for Declarative Layouts

Activity restarts caused by device rotation can also be prevented if we define the layout in XML. For example, we can use this approach if we want to prevent an Activity restart (for performance reasons, perhaps) and we don't need to load new resources for different orientations.

To do this, we follow the same procedure that we use with a programmatic layout. Simply set `ConfigurationChanges` in the `ActivityAttribute`, as we did in the `CodeLayoutActivity` earlier. Any code that does need to run for the orientation change can again be implemented in the `OnConfigurationChanged` method.

Maintaining State During Orientation Changes

Whether handling rotation declaratively or programmatically, all Android applications should implement the same techniques for managing state when device orientation changes. Managing state is important because the system restarts a running Activity when an Android device is rotated. Android does this to make it easy to load alternate resources, such as layouts and drawables that are designed specifically for a particular orientation. When it restarts, the Activity loses any transient state it may have stored in local class variables. Therefore, if an Activity is state reliant, it must persist its state at the application level. An application needs to handle saving and restoring any application state that it wants to preserve across orientation changes.

For more information on persisting state in Android, refer to the Activity Lifecycle guide.

Summary

This article covered how to use Android's built-in capabilities to work with rotation. First, it explained how to use the Android resource system to create orientation aware applications. Then it presented how to add controls in code as well as how to handle orientation changes manually.

Related Links

- Rotation Demo (sample)
- Activity Lifecycle
- Handling Runtime Changes
- Fast Screen Orientation Change

Android Audio

7/8/2021 • 10 minutes to read • Edit Online

The Android OS provides extensive support for multimedia, encompassing both audio and video. This guide focuses on audio in Android and covers playing and recording audio using the built-in audio player and recorder classes, as well as the low-level audio API. It also covers working with Audio events broadcast by other applications, so that developers can build well-behaved applications.

Overview

Modern mobile devices have adopted functionality that formerly would have required dedicated pieces of equipment – cameras, music players and video recorders. Because of this, multimedia frameworks have become a first-class feature in mobile APIs.

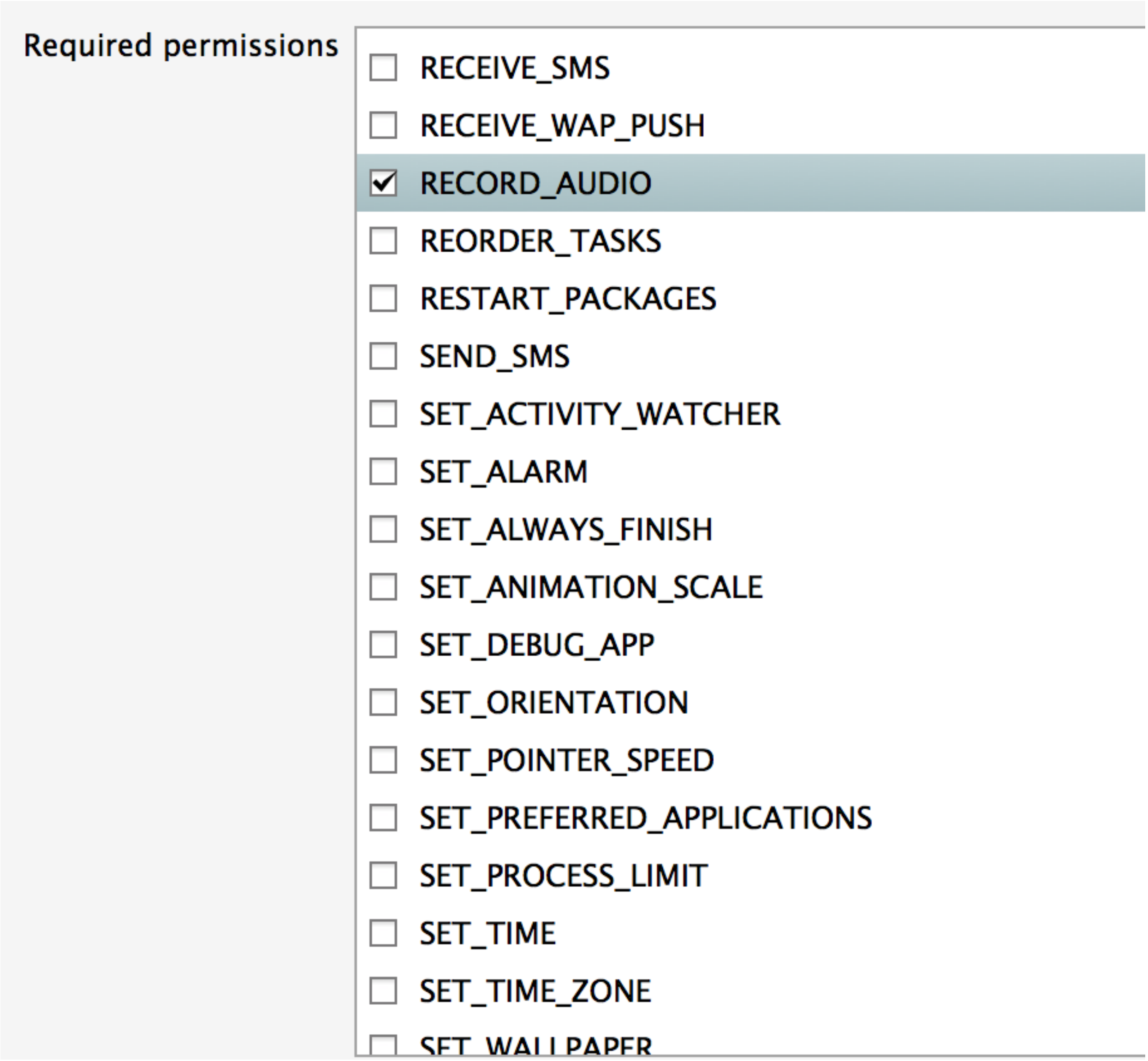
Android provides extensive support for multimedia. This article examines working with audio in Android, and covers the following topics

1. **Playing Audio with MediaPlayer** – Using the built-in `MediaPlayer` class to play audio, including local audio files and streamed audio files with the `AudioTrack` class.
2. **Recording Audio** – Using the built-in `MediaRecorder` class to record audio.
3. **Working with Audio Notifications** – Using audio notifications to create well-behaved applications that respond correctly to events (such as incoming phone calls) by suspending or canceling their audio outputs.
4. **Working with Low-Level Audio** – Playing audio using the `AudioTrack` class by writing directly to memory buffers. Recording audio using the `AudioRecord` class and reading directly from memory buffers.

Requirements

This guide requires Android 2.0 (API level 5) or higher. Please note that debugging audio on Android must be done on a device.

It is necessary to request the `RECORD_AUDIO` permissions in **AndroidManifest.XML**:



Playing Audio with the MediaPlayer Class

The simplest way to play audio in Android is with the built-in MediaPlayer class. `MediaPlayer` can play either local or remote files by passing in the file path. However, `MediaPlayer` is very state-sensitive and calling one of its methods in the wrong state will cause an exception to be thrown. It's important to interact with `MediaPlayer` in the order described below to avoid errors.

Initializing and Playing

Playing audio with `MediaPlayer` requires the following sequence:

1. Instantiate a new MediaPlayer object.

2. Configure the file to play via the SetDataSource method.

3. Call the Prepare method to initialize the player.

4. Call the Start method to start the audio playing.

The code sample below illustrates this usage:

```
protected MediaPlayer player;
public void StartPlayer(String  filePath)
{
  if (player == null) {
    player = new MediaPlayer();
  } else {
    player.Reset();
    player.SetDataSource(filePath);
    player.Prepare();
    player.Start();
  }
}
```

Suspending and Resuming Playback

The playback can be suspended by calling the Pause method:

```
player.Pause();
```

To resume paused playback, call the Start method. This will resume from the paused location in the playback:

```
player.Start();
```

Calling the Stop method on the player ends an ongoing playback:

```
player.Stop();
```

When the player is no longer needed, the resources must be released by calling the Release method:

```
player.Release();
```

Using the MediaRecorder Class to Record Audio

The corollary to `MediaPlayer` for recording audio in Android is the MediaRecorder class. Like the `MediaPlayer`, it is state-sensitive and transitions through several states to get to the point where it can start recording. In order to record audio, the `RECORD_AUDIO` permission must be set. For instructions on how to set application permissions see Working with AndroidManifest.xml.

Initializing and Recording

Recording audio with the `MediaRecorder` requires the following steps:

1. Instantiate a new MediaRecorder object.
2. Specify which hardware device to use to capture the audio input via the SetAudioSource method.
3. Set the output file audio format using the SetOutputFormat method. For a list of supported audio types see Android Supported Media Formats.
4. Call the SetAudioEncoder method to set the audio encoding type.
5. Call the SetOutputFile method to specify the name of the output file that the audio data is written to.
6. Call the Prepare method to initialize the recorder.
7. Call the Start method to start recording.

The following code sample illustrates this sequence:

```
protected MediaRecorder recorder;
void RecordAudio (String filePath)
{
  try {
    if (File.Exists (filePath)) {
      File.Delete (filePath);
    }
    if (recorder == null) {
      recorder = new MediaRecorder (); // Initial state.
    } else {
      recorder.Reset ();
      recorder.SetAudioSource (AudioSource.Mic);
      recorder.SetOutputFormat (OutputFormat.ThreeGpp);
      recorder.SetAudioEncoder (AudioEncoder.AmrNb);
      // Initialized state.
      recorder.SetOutputFile (filePath);
      // DataSourceConfigured state.
      recorder.Prepare (); // Prepared state
      recorder.Start (); // Recording state.
    }
  } catch (Exception ex) {
    Console.Out.WriteLine( ex.StackTrace);
  }
}
```

Stopping recording

To stop the recording, call the `Stop` method on the `MediaRecorder`:

```
recorder.Stop();
```

Cleaning up

Once the `MediaRecorder` has been stopped, call the Reset method to put it back into its idle state:

```
recorder.Reset();
```

When the `MediaRecorder` is no longer needed, its resources must be released by calling the Release method:

```
recorder.Release();
```

Managing Audio Notifications

The AudioManager Class

The AudioManager class provides access to audio notifications that let applications know when audio events occur. This service also provides access to other audio features, such as volume and ringer mode control. The `AudioManager` allows an application to handle audio notifications to control audio playback.

Managing Audio Focus

The audio resources of the device (the built-in player and recorder) are shared by all running applications.

Conceptually, this is similar to applications on a desktop computer where only one application has the keyboard focus: after selecting one of the running applications by mouse-clicking it, the keyboard input goes only to that application.

Audio focus is a similar idea and prevents more than one application from playing or recording audio at the same time. It is more complicated than keyboard focus because it is voluntary – the application can ignore that fact that it does not currently have audio focus and play regardless – and because there are different types of audio focus that can be requested. For example, if the requestor is only expected to play audio for a very short time, it may request transient focus.

Audio focus may be granted immediately, or initially denied and granted later. For example, if an application requests audio focus during a phone call, it will be denied, but focus may well be granted once the phone call is finished. In this case, a listener is registered in order to respond accordingly if audio focus is taken away. Requesting audio focus is used to determine whether or not it is OK to play or record audio.

For more information about audio focus, see Managing Audio Focus.

Registering the Callback for Audio Focus

Registering the `FocusChangeListener` callback from the `IOnAudioChangeListener` is an important part of obtaining and releasing audio focus. This is because the granting of audio focus may be deferred until a later time. For example, an application may request to play music while there is a phone call in progress. Audio focus will not be granted until the phone call is finished.

For this reason, the callback object is passed as a parameter into the `GetAudioFocus` method of the `AudioManager`, and it is this call that registers the callback. If audio focus is initially denied but later granted, the application is informed by invoking `OnAudioFocusChange` on the callback. The same method is used to tell the application that audio focus is being taken away.

When the application has finished using the audio resources, it calls the `AbandonFocus` method of the `AudioManager`, and again passes in the callback. This deregisters the callback and releases the audio resources, so that other applications may obtain audio focus.

Requesting Audio Focus

The steps required to request the audio resources of the device are as follow:

1. Obtain a handle to the `AudioManager` system service.

2. Create an instance of the callback class.

3. Request the audio resources of the device by calling the `RequestAudioFocus` method on the `AudioManager`. The parameters are the callback object, the stream type (music, voice call, ring etc.) and the type of the access right being requested (the audio resources can be requested momentarily or for an indefinite period, for example).

4. If the request is granted, the `playMusic` method is invoked immediately, and the audio starts to play back.

5. If the request is denied, no further action is taken. In this case, the audio will only play if the request is granted at a later time.

The code sample below shows these steps:

```
Boolean RequestAudioResources(INotificationReceiver parent)
{
  AudioManager audioMan = (AudioManager) GetSystemService(Context.AudioService);
  AudioManager.IOnAudioFocusChangeListener listener  = new MyAudioListener(this);
  var ret = audioMan.RequestAudioFocus (listener, Stream.Music, AudioFocus.Gain );
  if (ret == AudioFocusRequest.Granted) {
    playMusic();
    return (true);
  } else if (ret == AudioFocusRequest.Failed) {
    return (false);
  }
  return (false);
}
```

Releasing Audio Focus

When the playback of the track is complete, the `AbandonFocus` method on `AudioManager` is invoked. This allows another application to gain the audio resources of the device. Other applications will receive a notification of this audio focus change if they have registered their own listeners.

Low Level Audio API

The low-level audio APIs provide a greater control over audio playing and recording because they interact directly with memory buffers instead of using file URIs. There are some scenarios where this approach is preferable. Such scenarios include:

1. When playing from encrypted audio files.

2. When playing a succession of short clips.

3. Audio streaming.

AudioTrack Class

The AudioTrack class uses the low-level audio APIs for recording, and is the low-level equivalent of the `MediaPlayer` class.

Initializing and Playing

To play audio, a new instance of `AudioTrack` must be instantiated. The argument list passed into the constructor specifies how to play the audio sample contained in the buffer. The arguments are:

1. Stream type – Voice, ringtone, music, system or alarm.

2. Frequency – The sampling rate expressed in Hz.

3. Channel Configuration – Mono or stereo.

4. Audio format – 8 bit or 16 bit encoding.

5. Buffer size – in bytes.

6. Buffer mode – streaming or static.

After construction, the Play method of `AudioTrack` is invoked, to set it up to start playing. Writing the audio buffer to the `AudioTrack` starts the playback:

```
void PlayAudioTrack(byte[] audioBuffer)
{
  AudioTrack audioTrack = new AudioTrack(
    // Stream type
    Stream.Music,
    // Frequency
    11025,
    // Mono or stereo
    ChannelOut.Mono,
    // Audio encoding
    Android.Media.Encoding.Pcm16bit,
    // Length of the audio clip.
    audioBuffer.Length,
    // Mode. Stream or static.
    AudioTrackMode.Stream);

    audioTrack.Play();
    audioTrack.Write(audioBuffer, 0, audioBuffer.Length);
}
```

Pausing and Stopping the Playback

Call the Pause method to pause the playback:

```
audioTrack.Pause();
```

Calling the Stop method will terminate the playback permanently:

```
audioTrack.Stop();
```

Cleanup

When the `AudioTrack` is no longer needed, its resources must be released by calling Release:

```
audioTrack.Release();
```

The AudioRecord Class

The AudioRecord class is the equivalent of `AudioTrack` on the recording side. Like `AudioTrack`, it uses memory buffers directly, in place of files and URIs. It requires that the `RECORD_AUDIO` permission be set in the manifest.

Initializing and Recording

The first step is to construct a new AudioRecord object. The argument list passed into the constructor provides all the information required for recording. Unlike in `AudioTrack`, where the arguments are largely enumerations, the equivalent arguments in `AudioRecord` are integers. These include:

1. Hardware audio input source such as microphone.
2. Stream type – Voice, ringtone, music, system or alarm.
3. Frequency – The sampling rate expressed in Hz.
4. Channel Configuration – Mono or stereo.
5. Audio format – 8 bit or 16 bit encoding.
6. Buffer size-in bytes

Once the `AudioRecord` is constructed, its StartRecording method is invoked. It is now ready to begin recording. The `AudioRecord` continuously reads the audio buffer for input, and writes this input out to an audio file.

```
void RecordAudio()
{
  byte[] audioBuffer = new byte[100000];
  var audRecorder = new AudioRecord(
    // Hardware source of recording.
    AudioSource.Mic,
    // Frequency
    11025,
    // Mono or stereo
    ChannelIn.Mono,
    // Audio encoding
    Android.Media.Encoding.Pcm16bit,
    // Length of the audio clip.
    audioBuffer.Length
  );
  audRecorder.StartRecording();
  while (true) {
    try
    {
      // Keep reading the buffer while there is audio input.
      audRecorder.Read(audioBuffer, 0, audioBuffer.Length);
      // Write out the audio file.
    } catch (Exception ex) {
      Console.Out.WriteLine(ex.Message);
      break;
    }
  }
}
```

Stopping the Recording

Calling the Stop method terminates the recording:

```
audRecorder.Stop();
```

Cleanup

When the `AudioRecord` object is no longer needed, calling its Release method releases all resources associated with it:

```
audRecorder.Release();
```

Summary

The Android OS provides a powerful framework for playing, recording and managing audio. This article covered how to play and record audio using the high-level `MediaPlayer` and `MediaRecorder` classes. Next, it explored how to use audio notifications to share the audio resources of the device between different applications. Finally, it dealt with how to playback and record audio using the low-level APIs, which interface directly with memory buffers.

Related Links

- Working With Audio (sample)
- Media Player
- Media Recorder
- Audio Manager
- Audio Track
- Audio Recorder

Notifications in Xamarin.Android

11/2/2020 • 2 minutes to read • Edit Online

This section explains how to implement notifications in Xamarin.Android. It describes the various UI elements of an Android notification and discusses the API's involved with creating and displaying a notification.

Local notifications in Android

This section explains how to implement local notifications in Xamarin.Android. It describes the various UI elements of an Android notification and discuss the APIs involved with creating and displaying a notification.

Walkthrough - using local notifications in Xamarin.Android

This walkthrough covers how to use local notifications in a Xamarin.Android application. It demonstrates the basics of creating and publishing a notification. When the user clicks on the notification in the notification drawer it starts up a second Activity.

Further reading

Firebase Cloud Messaging – Firebase Cloud Messaging (FCM) is a service that facilitates messaging between mobile apps and server applications. Firebase Cloud Messaging can be used to implement remote notifications (also called push notifications) in Xamarin.Android applications.

Notifications – This Android Developer topic is the definitive guide for Android notifications. It includes a design considerations section that helps you design your notifications so that they conform to the guidelines of the Android user interface. It provides more background information about preserving navigation when starting an Activity, and it explains how to display progress in a notification and control media playback on the Lock Screen.

NotificationListenerService – This Android service makes it possible for your app to listen to (and interact with) all notifications posted on the Android device, not just the notifications that your app is registered to receive. Note that the user must explicitly give permission to your app for it to be able to listen for notifications on the device.

Related links

- Local Notifications (sample)

Local notifications on Android

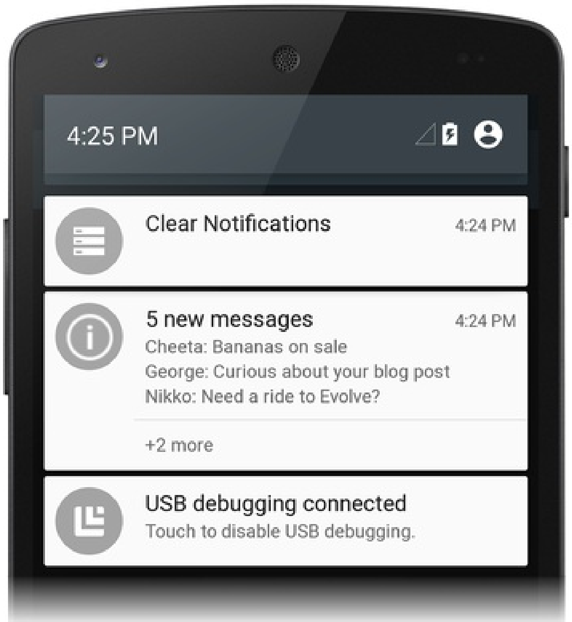
7/8/2021 • 29 minutes to read • Edit Online

This section shows how to implement local notifications in Xamarin.Android. It explains the various UI elements of an Android notification and discusses the API's involved with creating and displaying a notification.

Local notifications overview

Android provides two system-controlled areas for displaying notification icons and notification information to the user. When a notification is first published, its icon is displayed in the *notification area*, as shown in the following screenshot:

To obtain details about the notification, the user can open the notification drawer (which expands each notification icon to reveal notification content) and perform any actions associated with the notifications. The following screenshot shows a *notification drawer* that corresponds to the notification area displayed above:



Android notifications use two types of layouts:

- ***Base layout*** – a compact, fixed presentation format.
- ***Expanded layout*** – a presentation format that can expand to a larger size to reveal more information.

Each of these layout types (and how to create them) is explained in the following sections.

> **NOTE**
>
> This guide focuses on the NotificationCompat APIs from the Android support library. These APIs will ensure maximum backwards compatibility to Android 4.0 (API level 14).

Base layout

All Android notifications are built on the base layout format, which, at a minimum, includes the following elements:

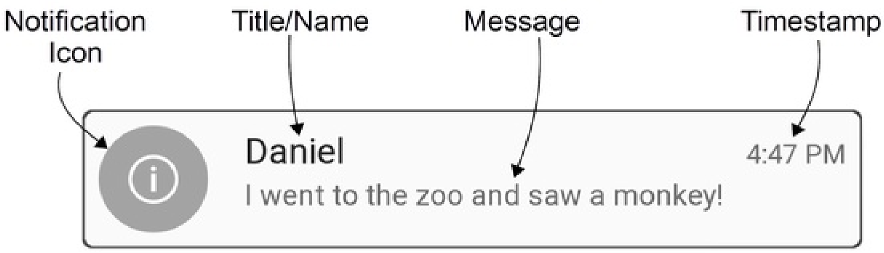
1. A *notification icon*, which represents the originating app, or the notification type if the app supports different types of notifications.

2. The notification *title*, or the name of the sender if the notification is a personal message.

3. The notification message.

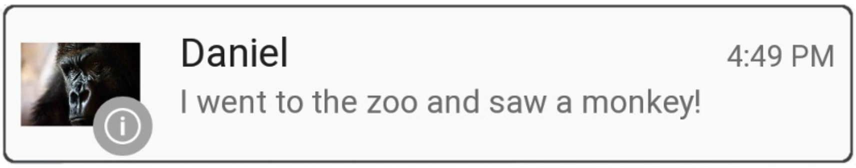
4. A *timestamp*.

These elements are displayed as illustrated in the following diagram:

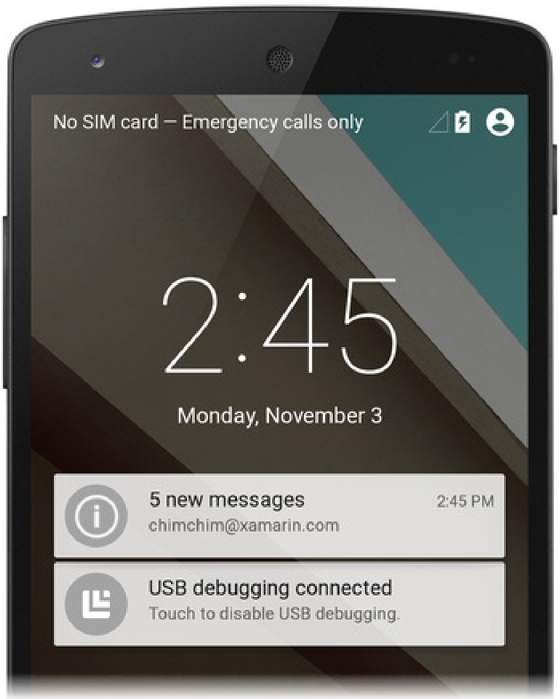


Base layouts are limited to 64 density-independent pixels (dp) in height. Android creates this basic notification style by default.

Optionally, notifications can display a large icon that represents the application or the sender's photo. When a large icon is used in a notification in Android 5.0 and later, the small notification icon is displayed as a badge over the large icon:

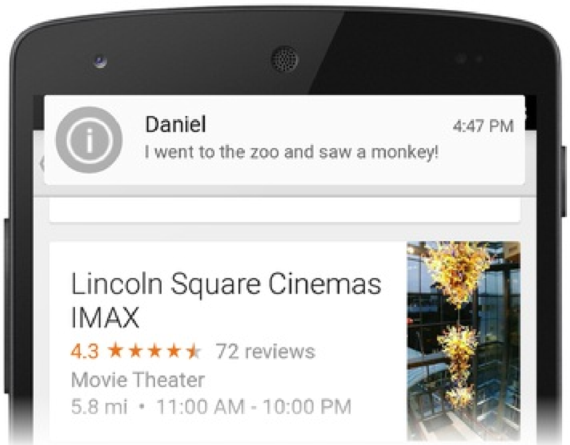


Beginning with Android 5.0, notifications can also appear on the lock screen:



The user can double-tap the lock screen notification to unlock the device and jump to the app that originated that notification, or swipe to dismiss the notification. Apps can set the visibility level of a notification to control what is shown on the lock screen, and users can choose whether to allow sensitive content to be shown in lock screen notifications.

Android 5.0 introduced a high-priority notification presentation format called *Heads-up*. Heads-up notifications slide down from the top of the screen for a few seconds and then retreat back up to the notification area:



Heads-up notifications make it possible for the system UI to put important information in front of the user without disrupting the state of the currently running activity.

Android includes support for notification metadata so that notifications can be sorted and displayed intelligently. Notification metadata also controls how notifications are presented on the lock screen and in Heads-up format. Applications can set the following types of notification metadata:

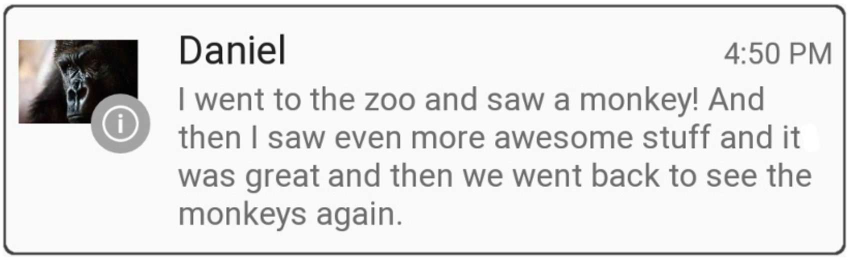
- **Priority** – The priority level determines how and when notifications are presented. For example, In Android 5.0, high-priority notifications are displayed as Heads-up notifications.
- **Visibility** – Specifies how much notification content is to be displayed when the notification appears on the lock screen.
- **Category** – Informs the system how to handle the notification in various circumstances, such as when the device is in *Do Not Disturb* mode.

> **NOTE**
>
> **Visibility** and **Category** were introduced in Android 5.0 and are not available in earlier versions of Android. Beginning with Android 8.0, notification channels are used to control how notifications are presented to the user.

Expanded layouts

Beginning with Android 4.1, notifications can be configured with expanded layout styles that allow the user to expand the height of the notification to view more content. For example, the following example illustrates an expanded layout notification in contracted mode:

When this notification is expanded, it reveals the entire message:



Android supports three expanded layout styles for single-event notifications:

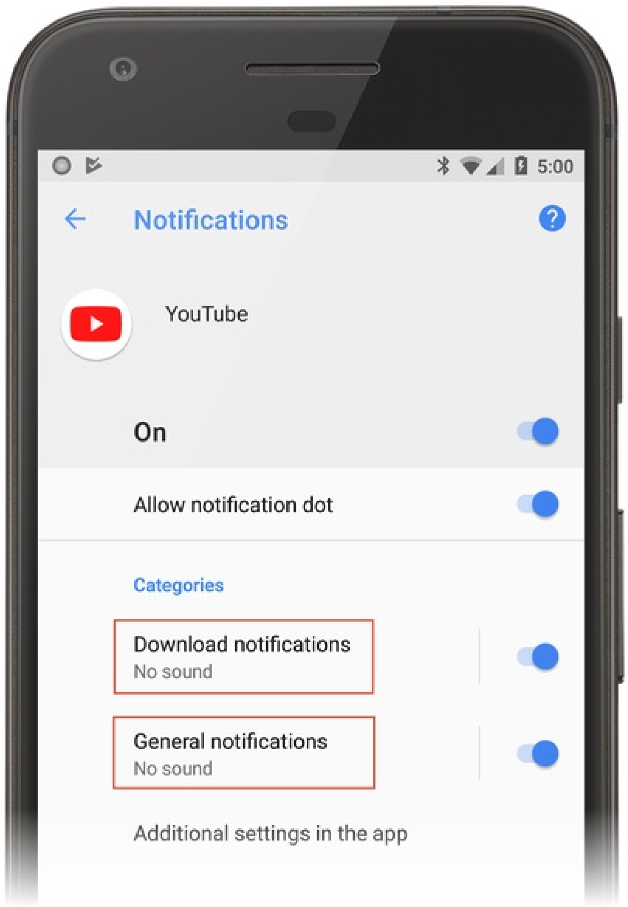
- *Big Text* – In contracted mode, displays an excerpt of the first line of the message followed by two periods. In expanded mode, displays the entire message (as seen in the above example).
- *Inbox* – In contracted mode, displays the number of new messages. In expanded mode, displays the first email message or a list of the messages in the inbox.
- *Image* – In contracted mode, displays only the message text. In expanded mode, displays the text and an image.

Beyond the Basic Notification (later in this article) explains how to create *Big Text*, *Inbox*, and *Image* notifications.

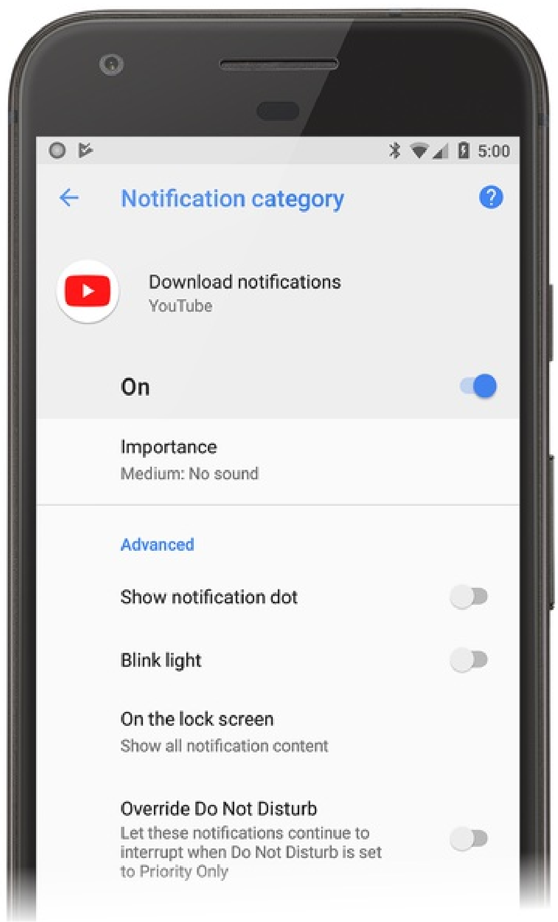
Notification channels

Beginning with Android 8.0 (Oreo), you can use the *notification channels* feature to create a user-customizable channel for each type of notification that you want to display. Notification channels make it possible for you to group notifications so that all notifications posted to a channel exhibit the same behavior. For example, you might have a notification channel that is intended for notifications that require immediate attention, and a separate "quieter" channel that is used for informational messages.

The **YouTube** app that is installed with Android Oreo lists two notification categories: **Download notifications** and **General notifications**:



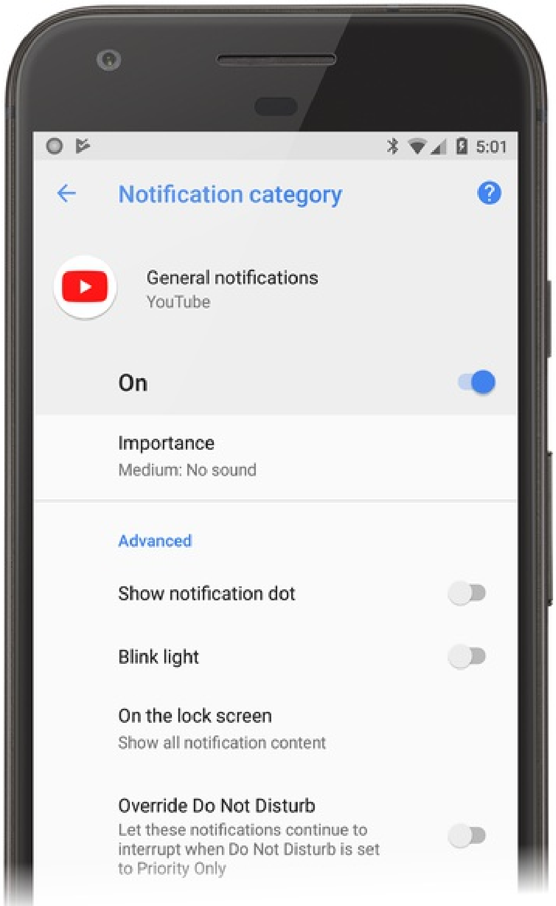
Each of these categories corresponds to a notification channel. The YouTube app implements a **Download Notifications** channel and a **General Notifications** channel. The user can tap **Download notifications**, which displays the settings screen for the app's download notifications channel:



In this screen, the user can modify the behavior of the **Download** notifications channel by doing the following:

- Set the Importance level to **Urgent**, **High**, **Medium**, or **Low**, which configures the level of sound and visual interruption.
- Turn the notification dot on or off.
- Turn the blinking light on or off.
- Show or hide notifications on the lock screen.
- Override the **Do Not Disturb** setting.

The **General Notifications** channel has similar settings:



Notice that you do not have absolute control over how your notification channels interact with the user – the user can modify the settings for any notification channel on the device as seen in the screenshots above. However, you can configure default values (as will be described below). As these examples illustrate, the new notification channels feature makes it possible for you to give users fine-grained control over different kinds of notifications.

Notification creation

To create a notification in Android, you use the NotificationCompat.Builder class from the Xamarin.Android.Support.v4 NuGet package. This class makes it possible to create and publish notifications on older versions of Android. `NotificationCompat.Builder` is also discussed.

`NotificationCompat.Builder` provides methods for setting the various options in a notification, such as:

- The content, including the title, the message text, and the notification icon.
- The style of the notification, such as *Big Text*, *Inbox*, or *Image* style.

- The priority of the notification: minimum, low, default, high, or maximum. On Android 8.0 and higher, the priority is set via a *notification channel*.

- The visibility of the notification on the lock screen: public, private, or secret.

- Category metadata that helps Android classify and filter the notification.

- An optional intent that indicates an activity to launch when the notification is tapped.

- The ID of the notification channel that the notification will be published on (Android 8.0 and higher).

After you set these options in the builder, you generate a notification object that contains the settings. To publish the notification, you pass this notification object to the *Notification Manager*. Android provides the NotificationManager class, which is responsible for publishing notifications and displaying them to the user. A reference to this class can be obtained from any context, such as an activity or a service.

Creating a notification channel

Apps that are running on Android 8.0 must create a notification channel for their notifications. A notification channel requires the following three pieces of information:

- An ID string that is unique to the package that will identify the channel.
- The name of the channel that will be displayed to the user. The name must be between one and 40 characters.
- The importance of the channel.

Apps will need to check the version of Android that they are running. Devices running versions older than Android 8.0 should not create a notification channel. The following method is one example of how to create a notification channel in an activity:

```
void CreateNotificationChannel()
{
    if (Build.VERSION.SdkInt < BuildVersionCodes.O)
    {
        // Notification channels are new in API 26 (and not a part of the
        // support library). There is no need to create a notification
        // channel on older versions of Android.
        return;
    }

    var channelName = Resources.GetString(Resource.String.channel_name);
    var channelDescription = GetString(Resource.String.channel_description);
    var channel = new NotificationChannel(CHANNEL_ID, channelName, NotificationImportance.Default)
              {
                  Description = channelDescription
              };

    var notificationManager = (NotificationManager) GetSystemService(NotificationService);
    notificationManager.CreateNotificationChannel(channel);
}
```

The notification channel should be created each time the activity is created. For the `CreateNotificationChannel` method, it should be called in the `OnCreate` method of an activity.

Creating and publishing a notification

To generate a notification in Android, follow these steps:

1. Instantiate a `NotificationCompat.Builder` object.

2. Call various methods on the `NotificationCompat.Builder` object to set notification options.

3. Call the Build method of the `NotificationCompat.Builder` object to instantiate a notification object.

4. Call the Notify method of the notification manager to publish the notification.

You must provide at least the following information for each notification:

- A small icon (24x24 dp in size)
- A short title
- The text of the notification

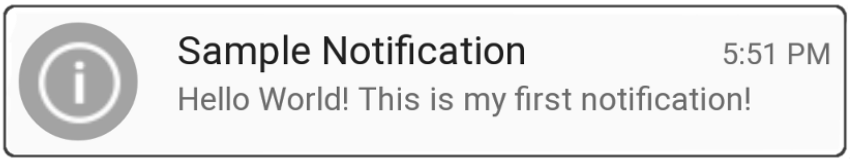
The following code example illustrates how to use `NotificationCompat.Builder` to generate a basic notification. Notice that `NotificationCompat.Builder` methods support method chaining; that is, each method returns the builder object so you can use the result of the last method call to invoke the next method call:

```
// Instantiate the builder and set notification elements:
NotificationCompat.Builder builder = new NotificationCompat.Builder(this, CHANNEL_ID)
    .SetContentTitle ("Sample Notification")
    .SetContentText ("Hello World! This is my first notification!")
    .SetSmallIcon (Resource.Drawable.ic_notification);

// Build the notification:
Notification notification = builder.Build();

// Get the notification manager:
NotificationManager notificationManager =
    GetSystemService (Context.NotificationService) as NotificationManager;

// Publish the notification:
const int notificationId = 0;
notificationManager.Notify (notificationId, notification);
```

In this example, a new `NotificationCompat.Builder` object called `builder` is instantiated, along with the ID of the notification channel to be used. The title and text of the notification are set, and the notification icon is loaded from **Resources/drawable/ic_notification.png**. The call to the notification builder's `Build` method creates a notification object with these settings. The next step is to call the `Notify` method of the notification manager. To locate the notification manager, you call `GetSystemService`, as shown above.

The `Notify` method accepts two parameters: the notification identifier and the notification object. The notification identifier is a unique integer that identifies the notification to your application. In this example, the notification identifier is set to zero (0); however, in a production application, you will want to give each notification a unique identifier. Reusing the previous identifier value in a call to `Notify` causes the last notification to be overwritten.

When this code runs on an Android 5.0 device, it generates a notification that looks like the following example:



The notification icon is displayed on the left hand side of the notification – this image of a circled "i" has an alpha channel so that Android can draw a gray circular background behind it. You can also supply an icon without an alpha channel. To display a photographic image as an icon, see Large Icon Format later in this topic.

The timestamp is set automatically, but you can override this setting by calling the SetWhen method of the notification builder. For example, the following code example sets the timestamp to the current time:

```
builder.SetWhen (Java.Lang.JavaSystem.CurrentTimeMillis());
```

Enabling sound and vibration

If you want your notification to also play a sound, you can call the notification builder's SetDefaults method and pass in the `NotificationDefaults.Sound` flag:

```
// Instantiate the notification builder and enable sound:
NotificationCompat.Builder builder = new NotificationCompat.Builder(this, CHANNEL_ID)
    .SetContentTitle ("Sample Notification")
    .SetContentText ("Hello World! This is my first notification!")
    .SetDefaults (NotificationDefaults.Sound)
    .SetSmallIcon (Resource.Drawable.ic_notification);
```

This call to `SetDefaults` will cause the device to play a sound when the notification is published. If you want the device to vibrate rather than play a sound, you can pass `NotificationDefaults.Vibrate` to `SetDefaults.` If you want the device to play a sound and vibrate the device, you can pass both flags to `SetDefaults`:

```
builder.SetDefaults (NotificationDefaults.Sound | NotificationDefaults.Vibrate);
```

If you enable sound without specifying a sound to play, Android uses the default system notification sound. However, you can change the sound that will be played by calling the notification builder's SetSound method. For example, to play the alarm sound with your notification (instead of the default notification sound), you can get the URI for the alarm sound from the RingtoneManager and pass it to `SetSound`:

```
builder.SetSound (RingtoneManager.GetDefaultUri(RingtoneType.Alarm));
```

Alternatively, you can use the system default ringtone sound for your notification:

```
builder.SetSound (RingtoneManager.GetDefaultUri(RingtoneType.Ringtone));
```

After you create a notification object, it's possible to set notification properties on the notification object (rather than configure them in advance through `NotificationCompat.Builder` methods). For example, instead of calling the `SetDefaults` method to enable vibration on a notification, you can directly modify the bit flag of the notification's Defaults property:

```
// Build the notification:
Notification notification = builder.Build();

// Turn on vibrate:
notification.Defaults |= NotificationDefaults.Vibrate;
```

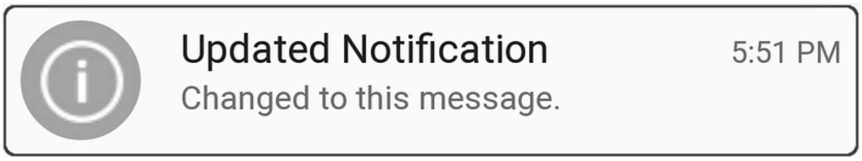
This example causes the device to vibrate when the notification is published.

Updating a notification

If you want to update the content of a notification after it has been published, you can reuse the existing `NotificationCompat.Builder` object to create a new notification object and publish this notification with the identifier of the last notification. For example:

```
// Update the existing notification builder content:
builder.SetContentTitle ("Updated Notification");
builder.SetContentText ("Changed to this message.");

// Build a notification object with updated content:
notification = builder.Build();

// Publish the new notification with the existing ID:
notificationManager.Notify (notificationId, notification);
```

In this example, the existing `NotificationCompat.Builder` object is used to create a new notification object with a different title and message. The new notification object is published using the identifier of the previous notification, and this updates the content of the previously-published notification:



The body of the previous notification is reused – only the title and the text of the notification changes while the notification is displayed in the notification drawer. The title text changes from "Sample Notification" to "Updated Notification" and the message text changes from "Hello World! This is my first notification!" to "Changed to this message."

A notification remains visible until one of three things happens:

- The user dismisses the notification (or taps *Clear All*).
- The application makes a call to `NotificationManager.Cancel`, passing in the unique notification ID that was assigned when the notification was published.
- The application calls `NotificationManager.CancelAll`.

For more about updating Android notifications, see Modify a Notification.

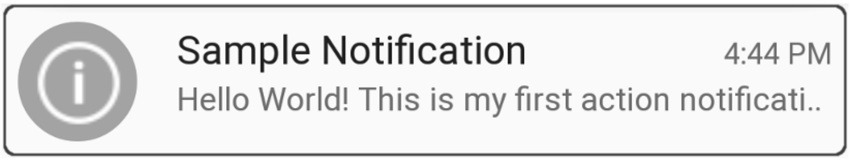
Starting an activity from a notification

In Android, it's common for a notification to be associated with an *action* – an activity that's launched when the user taps the notification. This activity can reside in another application or even in another task. To add an action to a notification, you create a PendingIntent object and associate the `PendingIntent` with the notification. A `PendingIntent` is a special type of intent that allows the recipient application to run a predefined piece of code with the permissions of the sending application. When the user taps the notification, Android starts up the activity specified by the `PendingIntent`.

The following code snippet illustrates how to create a notification with a `PendingIntent` that will launch the activity of the originating app, `MainActivity`:

```
// Set up an intent so that tapping the notifications returns to this app:
Intent intent = new Intent (this, typeof(MainActivity));

// Create a PendingIntent; we're only using one PendingIntent (ID = 0):
const int pendingIntentId = 0;
PendingIntent pendingIntent =
    PendingIntent.GetActivity (this, pendingIntentId, intent, PendingIntentFlags.OneShot);

// Instantiate the builder and set notification elements, including pending intent:
NotificationCompat.Builder builder = new NotificationCompat.Builder(this, CHANNEL_ID)
    .SetContentIntent (pendingIntent)
    .SetContentTitle ("Sample Notification")
    .SetContentText ("Hello World! This is my first action notification!")
    .SetSmallIcon (Resource.Drawable.ic_notification);

// Build the notification:
Notification notification = builder.Build();

// Get the notification manager:
NotificationManager notificationManager =
    GetSystemService (Context.NotificationService) as NotificationManager;

// Publish the notification:
const int notificationId = 0;
notificationManager.Notify (notificationId, notification);
```

This code is very similar to the notification code in the previous section, except that a `PendingIntent` is added to the notification object. In this example, the `PendingIntent` is associated with the activity of the originating app before it is passed to the notification builder's SetContentIntent method. The `PendingIntentFlags.OneShot` flag is passed to the `PendingIntent.GetActivity` method so that the `PendingIntent` is used only once. When this code runs, the following notification is displayed:



Tapping this notification takes the user back to the originating activity.

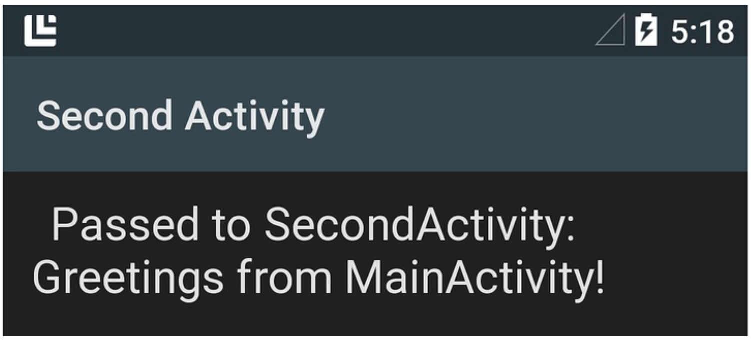
In a production app, your app must handle the *back stack* when the user presses the **Back** button within the notification activity (if you are not familiar with Android tasks and the back stack, see Tasks and Back Stack). In most cases, navigating backward out of the notification activity should return the user out of the app and back to Home screen. To manage the back stack, your app uses the TaskStackBuilder class to create a `PendingIntent` with a back stack.

Another real-world consideration is that the originating activity may need to send data to the notification activity. For example, the notification may indicate that a text message has arrived, and the notification activity (a message viewing screen), requires the ID of the message to display the message to the user. The activity that creates the `PendingIntent` can use the Intent.PutExtra method to add data (for example, a string) to the intent so that this data is passed to the notification activity.

The following code sample illustrates how to use `TaskStackBuilder` to manage the back stack, and it includes an example of how to send a single message string to a notification activity called `SecondActivity`:

```
// Setup an intent for SecondActivity:
Intent secondIntent = new Intent (this, typeof(SecondActivity));

// Pass some information to SecondActivity:
secondIntent.PutExtra ("message", "Greetings from MainActivity!");

// Create a task stack builder to manage the back stack:
TaskStackBuilder stackBuilder = TaskStackBuilder.Create(this);

// Add all parents of SecondActivity to the stack:
stackBuilder.AddParentStack (Java.Lang.Class.FromType (typeof (SecondActivity)));

// Push the intent that starts SecondActivity onto the stack:
stackBuilder.AddNextIntent (secondIntent);

// Obtain the PendingIntent for launching the task constructed by
// stackbuilder. The pending intent can be used only once (one shot):
const int pendingIntentId = 0;
PendingIntent pendingIntent =
    stackBuilder.GetPendingIntent (pendingIntentId, PendingIntentFlags.OneShot);

// Instantiate the builder and set notification elements, including
// the pending intent:
NotificationCompat.Builder builder = new NotificationCompat.Builder(this, CHANNEL_ID)
    .SetContentIntent (pendingIntent)
    .SetContentTitle ("Sample Notification")
    .SetContentText ("Hello World! This is my second action notification!")
    .SetSmallIcon (Resource.Drawable.ic_notification);

// Build the notification:
Notification notification = builder.Build();

// Get the notification manager:
NotificationManager notificationManager =
    GetSystemService (Context.NotificationService) as NotificationManager;

// Publish the notification:
const int notificationId = 0;
notificationManager.Notify (notificationId, notification);
```

In this code example, the app consists of two activities: `MainActivity` (which contains the notification code above), and `SecondActivity`, the screen the user sees after tapping the notification. When this code is run, a simple notification (similar to the previous example) is presented. Tapping on the notification takes the user to the `SecondActivity` screen:



The string message (passed into the intent's `PutExtra` method) is retrieved in `SecondActivity` via this line of code:

```
// Get the message from the intent:
string message = Intent.Extras.GetString ("message", "");
```

This retrieved message, "Greetings from MainActivity!," is displayed in the `SecondActivity` screen, as shown in the above screenshot. When the user presses the **Back** button while in `SecondActivity`, navigation leads out of the app and back to the screen preceding the launch of the app.

For more information about creating pending intents, see PendingIntent.

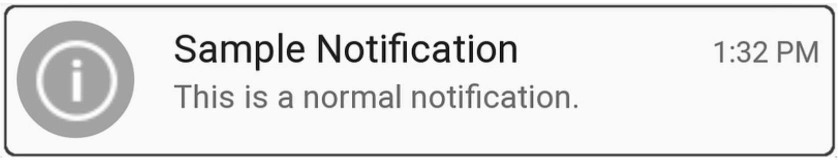
Beyond the basic notification

Notifications default to a simple base layout format in Android, but you can enhance this basic format by making additional `NotificationCompat.Builder` method calls. In this section, you'll learn how to add a large photo icon to your notification, and you'll see examples of how to create expanded layout notifications.

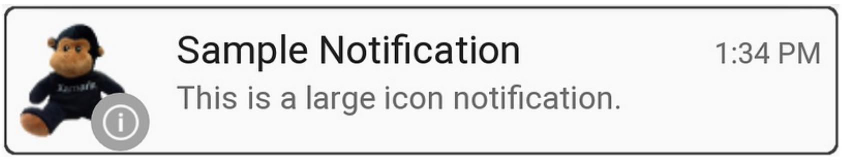
Large icon format

Android notifications typically display the icon of the originating app (on the left side of the notification). However, notifications can display an image or a photo (a *large icon*) instead of the standard small icon. For example, a messaging app could display a photo of the sender rather than the app icon.

Here is an example of a basic Android 5.0 notification – it displays only the small app icon:



And here is a screenshot of the notification after modifying it to display a large icon – it uses an icon created from an image of a Xamarin code monkey:



Notice that when a notification is presented in large icon format, the small app icon is displayed as a badge on the lower right corner of the large icon.

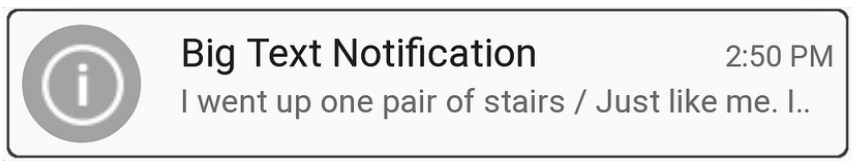
To use an image as a large icon in a notification, you call the notification builder's SetLargeIcon method and pass in a bitmap of the image. Unlike `SetSmallIcon`, `SetLargeIcon` only accepts a bitmap. To convert an image file into a bitmap, you use the BitmapFactory class. For example:

```
builder.SetLargeIcon (BitmapFactory.DecodeResource (Resources, Resource.Drawable.monkey_icon));
```

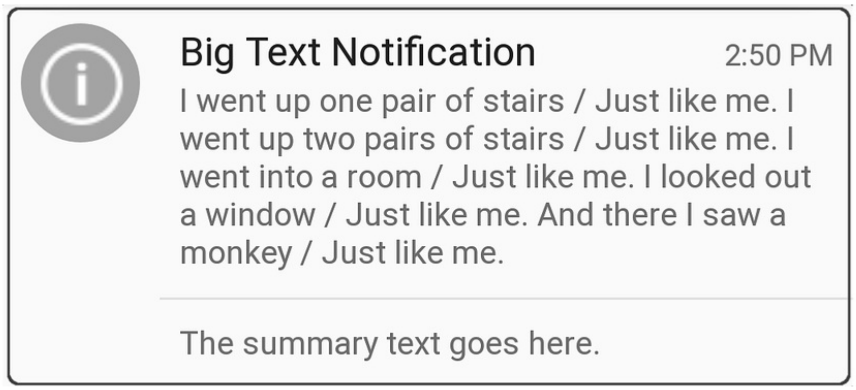
This example code opens the image file at **Resources/drawable/monkey_icon.png**, converts it to a bitmap, and passes the resulting bitmap to `NotificationCompat.Builder`. Typically, the source image resolution is larger than the small icon – but not much larger. An image that is too large might cause unnecessary resizing operations that could delay the posting of the notification.

Big text style

The *Big Text* style is an expanded layout template that you use for displaying long messages in notifications. Like all expanded layout notifications, the Big Text notification is initially displayed in a compact presentation format:



In this format, only an excerpt of the message is shown, terminated by two periods. When the user drags down on the notification, it expands to reveal the entire notification message:



This expanded layout format also includes summary text at the bottom of the notification. The maximum height of the *Big Text* notification is 256 dp.

To create a *Big Text* notification, you instantiate a `NotificationCompat.Builder` object, as before, and then instantiate and add a BigTextStyle object to the `NotificationCompat.Builder` object. Here is an example:

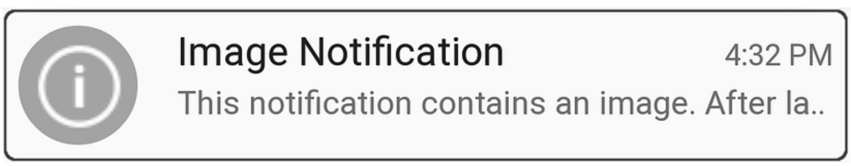
```
// Instantiate the Big Text style:
Notification.BigTextStyle textStyle = new Notification.BigTextStyle();

// Fill it with text:
string longTextMessage = "I went up one pair of stairs.";
longTextMessage += " / Just like me. ";
//...
textStyle.BigText (longTextMessage);

// Set the summary text:
textStyle.SetSummaryText ("The summary text goes here.");

// Plug this style into the builder:
builder.SetStyle (textStyle);

// Create the notification and publish it ...
```

In this example, the message text and summary text are stored in the `BigTextStyle` object (`textStyle`) before it is passed to `NotificationCompat.Builder.`

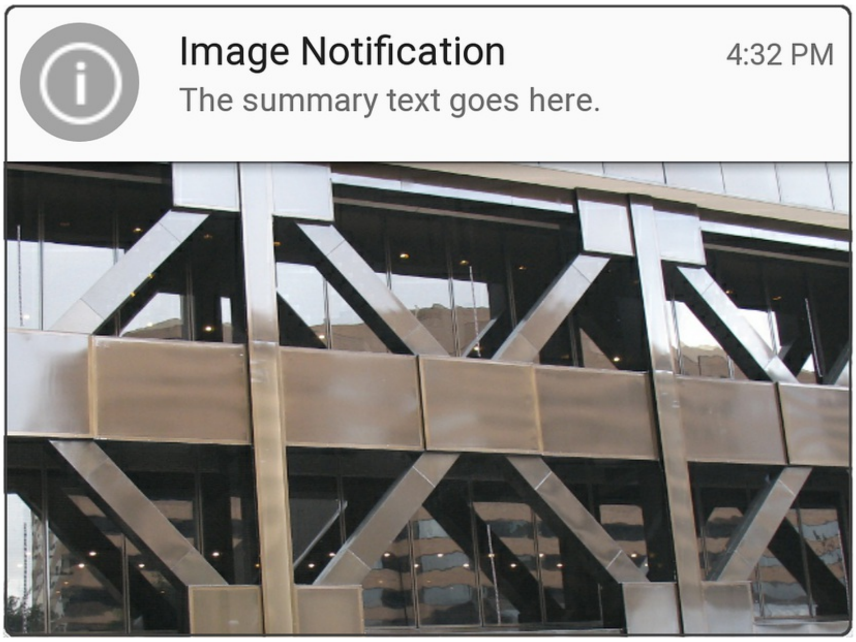
Image style

The *Image* style (also called the *Big Picture* style) is an expanded notification format that you can use to display an image in the body of a notification. For example, a screenshot app or a photo app can use the *Image* notification style to provide the user with a notification of the last image that was captured. Note that the maximum height of the *Image* notification is 256 dp – Android will resize the image to fit into this maximum height restriction, within the limits of available memory.

Like all expanded layout notifications, *Image* notifications are first displayed in a compact format that displays an excerpt of the accompanying message text:



When the user drags down on the *Image* notification, it expands to reveal an image. For example, here is the expanded version of the previous notification:



Notice that when the notification is displayed in compact format, it displays notification text (the text that is passed to the notification builder's `SetContentText` method, as shown earlier). However, when the notification is expanded to reveal the image, it displays summary text above the image.

To create an *Image* notification, you instantiate a `NotificationCompat.Builder` object as before, and then create and insert a BigPictureStyle object into the `NotificationCompat.Builder` object. For example:

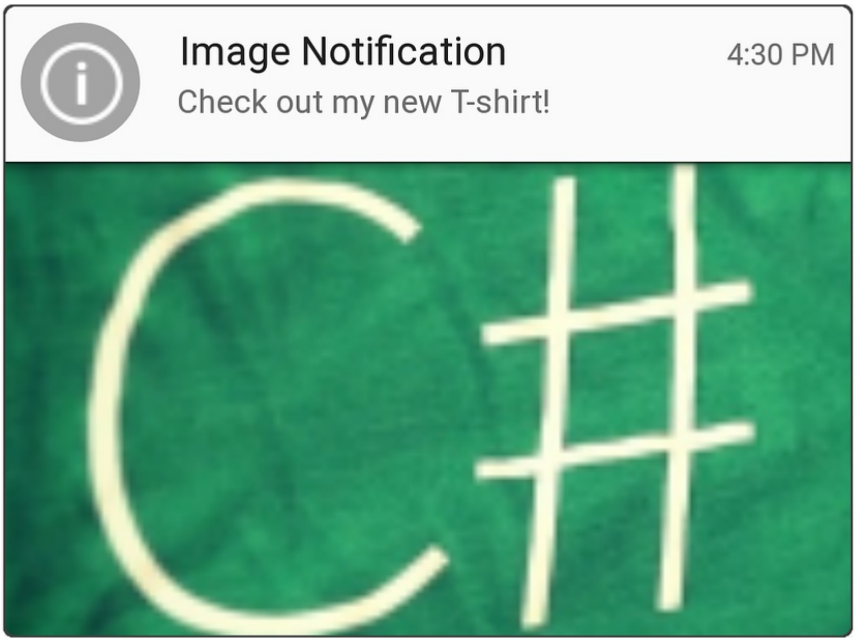
```
// Instantiate the Image (Big Picture) style:
Notification.BigPictureStyle picStyle = new Notification.BigPictureStyle();

// Convert the image to a bitmap before passing it into the style:
picStyle.BigPicture (BitmapFactory.DecodeResource (Resources, Resource.Drawable.x_bldg));

// Set the summary text that will appear with the image:
picStyle.SetSummaryText ("The summary text goes here.");

// Plug this style into the builder:
builder.SetStyle (picStyle);

// Create the notification and publish it ...
```

Like the `SetLargeIcon` method of `NotificationCompat.Builder`, the BigPicture method of `BigPictureStyle` requires a bitmap of the image that you want to display in the body of the notification. In this example, the DecodeResource method of `BitmapFactory` reads the image file located at **Resources/drawable/x_bldg.png** and converts it into a bitmap.

You can also display images that are not packaged as a resource. For example, the following sample code loads an image from the local SD card and displays it in an *Image* notification:

```
// Using the Image (Big Picture) style:
Notification.BigPictureStyle picStyle = new Notification.BigPictureStyle();

// Read an image from the SD card, subsample to half size:
BitmapFactory.Options options = new BitmapFactory.Options();
options.InSampleSize = 2;
string imagePath = "/sdcard/Pictures/my-tshirt.jpg";
picStyle.BigPicture (BitmapFactory.DecodeFile (imagePath, options));

// Set the summary text that will appear with the image:
picStyle.SetSummaryText ("Check out my new T-Shirt!");

// Plug this style into the builder:
builder.SetStyle (picStyle);

// Create notification and publish it ...
```

In this example, the image file located at **/sdcard/Pictures/my-tshirt.jpg** is loaded, resized to half of its original size, and then converted to a bitmap for use in the notification:

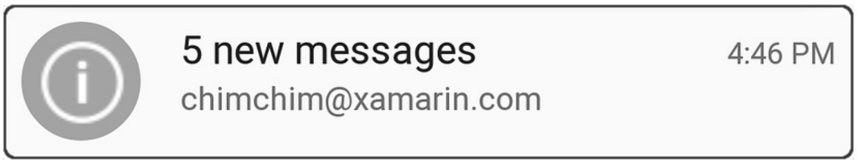


If you don't know the size of the image file in advance, it's a good idea to wrap the call to BitmapFactory.DecodeFile in an exception handler – an `OutOfMemoryError` exception might be thrown if the image is too large for Android to resize.

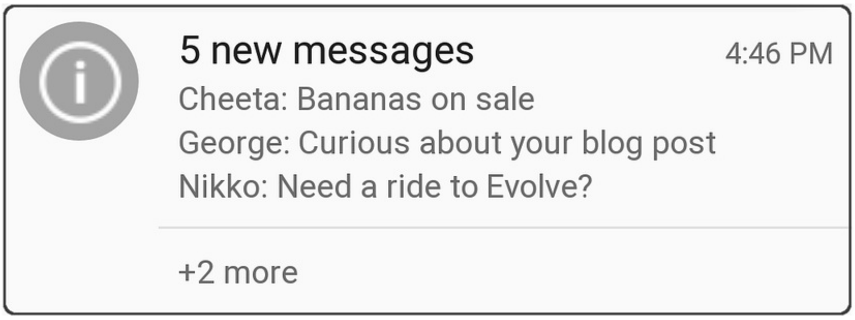
For more about loading and decoding large bitmap images, see Load Large Bitmaps Efficiently.

Inbox style

The *Inbox* format is an expanded layout template intended for displaying separate lines of text (such as an email inbox summary) in the body of the notification. The *Inbox* format notification is first displayed in a compact format:



When the user drags down on the notification, it expands to reveal an email summary as seen in the screenshot below:



To create an *Inbox* notification, you instantiate a `NotificationCompat.Builder` object, as before, and add an InboxStyle object to the `NotificationCompat.Builder`. Here is an example:

```
// Instantiate the Inbox style:
Notification.InboxStyle inboxStyle = new Notification.InboxStyle();

// Set the title and text of the notification:
builder.SetContentTitle ("5 new messages");
builder.SetContentText ("chimchim@xamarin.com");

// Generate a message summary for the body of the notification:
inboxStyle.AddLine ("Cheeta: Bananas on sale");
inboxStyle.AddLine ("George: Curious about your blog post");
inboxStyle.AddLine ("Nikko: Need a ride to Evolve?");
inboxStyle.SetSummaryText ("+2 more");

// Plug this style into the builder:
builder.SetStyle (inboxStyle);
```

To add new lines of text to the notification body, call the Addline method of the `InboxStyle` object (the maximum height of the *Inbox* notification is 256 dp). Note that, unlike *Big Text* style, the *Inbox* style supports individual lines of text in the notification body.

You can also use the *Inbox* style for any notification that needs to display individual lines of text in an expanded format. For example, the *Inbox* notification style can be used to combine multiple pending notifications into a summary notification – you can update a single *Inbox* style notification with new lines of notification content (see Updating a Notification above), rather than generate a continuous stream of new, mostly similar notifications.

Configuring metadata

`NotificationCompat.Builder` includes methods that you can call to set metadata about your notification, such as priority, visibility, and category. Android uses this information — along with user preference settings — to determine how and when to display notifications.

Priority settings

Apps running on Android 7.1 and lower need to set the priority directly on the notification itself. The priority setting of a notification determines two outcomes when the notification is published:

- Where the notification appears in relation to other notifications. For example, high priority notifications are presented above lower priority notifications in the notifications drawer, regardless of when each notification was published.
- Whether the notification is displayed in the Heads-up notification format (Android 5.0 and later). Only *high* and *maximum* priority notifications are displayed as Heads-up notifications.

Xamarin.Android defines the following enumerations for setting notification priority:

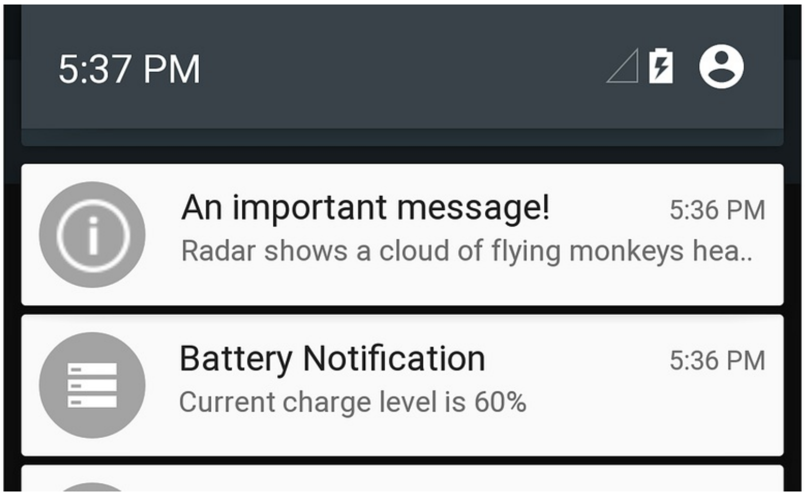
- `NotificationPriority.Max` – Alerts the user to an urgent or critical condition (for example, an incoming call, turn-by-turn directions, or an emergency alert). On Android 5.0 and later devices, maximum priority notifications are displayed in Heads-up format.

- `NotificationPriority.High` – Informs the user of important events (such as important emails or the arrival of real-time chat messages). On Android 5.0 and later devices, high priority notifications are displayed in Heads-up format.
- `NotificationPriority.Default` – Notifies the user of conditions that have a medium level of importance.
- `NotificationPriority.Low` – For non-urgent information that the user needs to be informed of (for example, software update reminders or social network updates).
- `NotificationPriority.Min` – For background information that the user notices only when viewing notifications (for example, location or weather information).

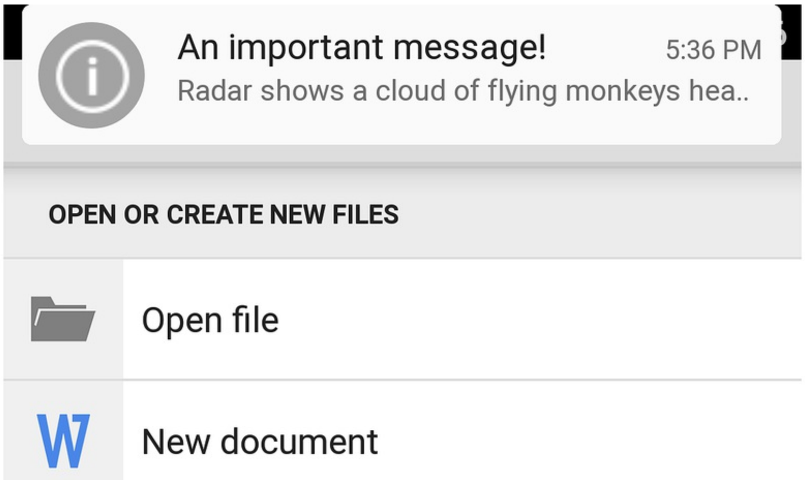
To set the priority of a notification, call the SetPriority method of the `NotificationCompat.Builder` object, passing in the priority level. For example:

```
builder.SetPriority (NotificationPriority.High);
```

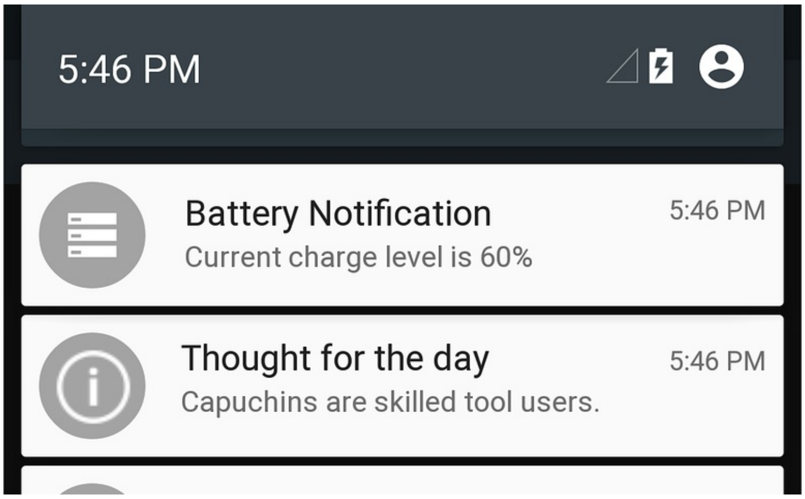
In the following example, the high priority notification, "An important message!" appears at the top of the notification drawer:



Because this is a high-priority notification, it is also displayed as a Heads-up notification above the user's current activity screen in Android 5.0:



In the next example, the low-priority "Thought for the day" notification is displayed under a higher-priority battery level notification:



Because the "Thought for the day" notification is a low-priority notification, Android will not display it in Heads-up format.

> **NOTE**
>
> On Android 8.0 and higher, the priority of the notification channel and user settings will determine the priority of the notification.

Visibility settings

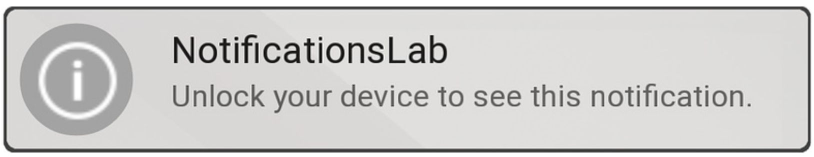
Beginning with Android 5.0, the *visibility* setting is available to control how much notification content appears on the secure lock screen. Xamarin.Android defines the following enumerations for setting notification visibility:

- `NotificationVisibility.Public` – The full content of the notification is displayed on the secure lock screen.
- `NotificationVisibility.Private` – Only essential information is displayed on the secure lock screen (such as the notification icon and the name of the app that posted it), but the rest of the notification's details are hidden. All notifications default to `NotificationVisibility.Private`.
- `NotificationVisibility.Secret` – Nothing is displayed on the secure lock screen, not even the notification icon. The notification content is available only after the user unlocks the device.

To set the visibility of a notification, apps call the `SetVisibility` method of the `NotificationCompat.Builder` object, passing in the visibility setting. For example, this call to `SetVisibility` makes the notification `Private`:

```
builder.SetVisibility (NotificationVisibility.Private);
```

When a `Private` notification is posted, only the name and icon of the app is displayed on the secure lock screen. Instead of the notification message, the user sees "Unlock your device to see this notification":



In this example, **NotificationsLab** is the name of the originating app. This redacted version of the notification appears only when the Lock screen is secure (i.e., secured via PIN, pattern, or password) – if the lock screen is not secure, the full content of the notification is available on the lock screen.

Category settings

Beginning with Android 5.0, predefined categories are available for ranking and filtering notifications.

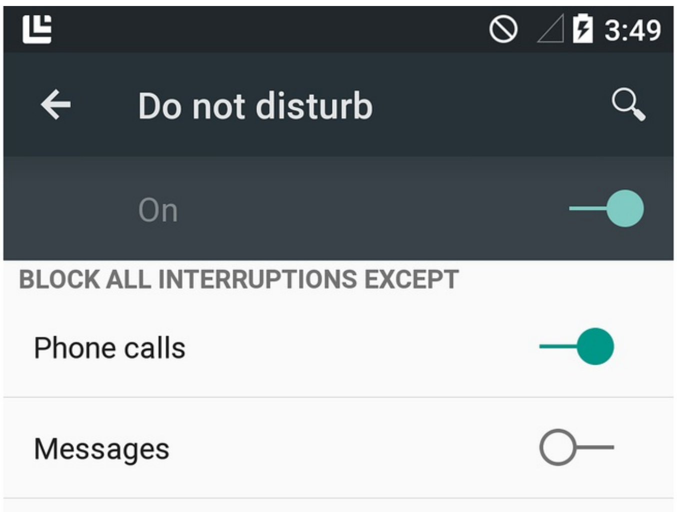
Xamarin.Android provides the following enumerations for these categories:

- `Notification.CategoryCall` – Incoming phone call.
- `Notification.CategoryMessage` – Incoming text message.
- `Notification.CategoryAlarm` – An alarm condition or timer expiration.
- `Notification.CategoryEmail` – Incoming email message.
- `Notification.CategoryEvent` – A calendar event.
- `Notification.CategoryPromo` – A promotional message or advertisement.
- `Notification.CategoryProgress` – The progress of a background operation.
- `Notification.CategorySocial` – Social networking update.
- `Notification.CategoryError` – Failure of a background operation or authentication process.
- `Notification.CategoryTransport` – Media playback update.
- `Notification.CategorySystem` – Reserved for system use (system or device status).
- `Notification.CategoryService` – Indicates that a background service is running.
- `Notification.CategoryRecommendation` – A recommendation message related to the currently running app.
- `Notification.CategoryStatus` – Information about the device.

When notifications are sorted, the notification's priority takes precedence over its category setting. For example, a high-priority notification will be displayed as Heads-up even if it belongs to the `Promo` category. To set the category of a notification, you call the `SetCategory` method of the `NotificationCompat.Builder` object, passing in the category setting. For example:

```
builder.SetCategory (Notification.CategoryCall);
```

The *Do not disturb* feature (new in Android 5.0) filters notifications based on categories. For example, the *Do not disturb* screen in **Settings** allows the user to exempt notifications for phone calls and messages:



When the user configures *Do not disturb* to block all interrupts except for phone calls (as illustrated in the above screenshot), Android allows notifications with a category setting of `Notification.CategoryCall` to be presented while the device is in *Do not disturb* mode. Note that `Notification.CategoryAlarm` notifications are never blocked in *Do not disturb* mode.

The LocalNotifications sample demonstrates how to use `NotificationCompat.Builder` to launch a second activity from a notification. This sample code is explained in the Using Local Notifications in Xamarin.Android walkthrough.

Notification styles

To create *Big Text*, *Image*, or *Inbox* style notifications with `NotificationCompat.Builder`, your app must use the compatibility versions of these styles. For example, to use the *Big Text* style, instantiate `NotificationCompat.BigTextstyle`:

```
NotificationCompat.BigTextStyle textStyle = new NotificationCompat.BigTextStyle();

// Plug this style into the builder:
builder.SetStyle (textStyle);
```

Similarly, your app can use `NotificationCompat.InboxStyle` and `NotificationCompat.BigPictureStyle` for *Inbox* and *Image* styles, respectively.

Notification priority and category

`NotificationCompat.Builder` supports the `SetPriority` method (available starting with Android 4.1). However, the `SetCategory` method is *not* supported by `NotificationCompat.Builder` because categories are part of the new notification metadata system that was introduced in Android 5.0.

To support older versions of Android, where `SetCategory` is not available, your code can check the API level at runtime to conditionally call `SetCategory` when the API level is equal to or greater than Android 5.0 (API level 21):

```
if (Android.OS.Build.VERSION.SdkInt >= Android.OS.BuildVersionCodes.Lollipop) {
    builder.SetCategory (Notification.CategoryEmail);
}
```

In this example, the app's **Target Framework** is set to Android 5.0 and the **Minimum Android Version** is set to **Android 4.1 (API Level 16)**. Because `SetCategory` is available in API level 21 and later, this example code will call `SetCategory` only when it is available – it will not call `SetCategory` when the API level is less than 21.

Lock screen visibility

Because Android did not support lock screen notifications before Android 5.0 (API level 21), `NotificationCompat.Builder` does not support the `SetVisibility` method. As explained above for `SetCategory`, your code can check the API level at runtime and call `SetVisiblity` only when it is available:

```
if (Android.OS.Build.VERSION.SdkInt >= Android.OS.BuildVersionCodes.Lollipop) {
    builder.SetVisibility (Notification.Public);
}
```

Summary

This article explained how to create local notifications in Android. It described the anatomy of a notification, it explained how to use `NotificationCompat.Builder` to create notifications, how to style notifications in large icon, *Big Text*, *Image* and *Inbox* formats, how to set notification metadata settings such as priority, visibility, and category, and how to launch an activity from a notification. This article also described how these notification settings work with the new Heads-up, lock screen, and *Do not disturb* features introduced in Android 5.0. Finally, you learned how to use `NotificationCompat.Builder` to maintain notification compatibility with earlier versions of Android.

For guidelines about designing notifications for Android, see Notifications.

Related Links

- NotificationsLab (sample)
- LocalNotifications (sample)
- Local Notifications In Android Walkthrough
- Notifying the User
- Notification
- NotificationManager
- NotificationCompat.Builder
- PendingIntent

Walkthrough - Using local notifications in Xamarin.Android

7/8/2021 • 5 minutes to read • Edit Online

This walkthrough demonstrates how to use local notifications in Xamarin.Android applications. It demonstrates the basics of creating and publishing a local notification. When the user clicks the notification in the notification area, it starts up a second Activity.

Overview

In this walkthrough, we will create an Android application that raises a notification when the user clicks a button in an Activity. When the user clicks the notification, it launches a second Activity that displays the number of times the user had clicked the button in the first Activity.

The following screenshots illustrate some examples of this application:

> **NOTE**
>
> This guide focuses on the NotificationCompat APIs from the Android support library. These APIs will ensure maximum backwards compatibility to Android 4.0 (API level 14).

Creating the project

To begin, let's create a new Android project using the **Android App** template. Let's call this project **LocalNotifications**. (If you are not familiar with creating Xamarin.Android projects, see Hello, Android.)

Edit the resource file **values/Strings.xml** so that it contains two extra string resources that will be used when it is time to create the notification channel:

```xml
<?xml version="1.0" encoding="utf-8"?>

<resources>
  <string name="Hello">Hello World, Click Me!</string>
  <string name="ApplicationName">Notifications</string>

  <string name="channel_name">Local Notifications</string>
  <string name="channel_description">The count from MainActivity.</string>
</resources>
```

Add the Android.Support.V4 NuGet package

In this walkthrough, we are using `NotificationCompat.Builder` to build our local notification. As explained in Local Notifications, we must include the Android Support Library v4 NuGet in our project to use `NotificationCompat.Builder`.

Next, let's edit **MainActivity.cs** and add the following `using` statement so that the types in `Android.Support.V4.App` are available to our code:

```
using Android.Support.V4.App;
```

Also, we must make it clear to the compiler that we are using the `Android.Support.V4.App` version of `TaskStackBuilder` rather than the `Android.App` version. Add the following `using` statement to resolve any ambiguity:

```
using TaskStackBuilder = Android.Support.V4.App.TaskStackBuilder;
```

Create the notification channel

Next, add a method to `MainActivity` that will create a notification channel (if necessary):

```
void CreateNotificationChannel()
{
    if (Build.VERSION.SdkInt < BuildVersionCodes.O)
    {
        // Notification channels are new in API 26 (and not a part of the
        // support library). There is no need to create a notification
        // channel on older versions of Android.
        return;
    }

    var name = Resources.GetString(Resource.String.channel_name);
    var description = GetString(Resource.String.channel_description);
    var channel = new NotificationChannel(CHANNEL_ID, name, NotificationImportance.Default)
                  {
                      Description = description
                  };

    var notificationManager = (NotificationManager) GetSystemService(NotificationService);
    notificationManager.CreateNotificationChannel(channel);
}
```

Update the `OnCreate` method to call this new method:

```
protected override void OnCreate(Bundle bundle)
{
    base.OnCreate(bundle);
    SetContentView(Resource.Layout.Main);

    CreateNotificationChannel();
}
```

Define the notification ID

We will need a unique ID for our notification and notification channel. Let's edit **MainActivity.cs** and add the following static instance variable to the `MainActivity` class:

```
static readonly int NOTIFICATION_ID = 1000;
static readonly string CHANNEL_ID = "location_notification";
internal static readonly string COUNT_KEY = "count";
```

Add code to generate the notification

Next, we need to create a new event handler for the button `Click` event. Add the following method to `MainActivity`:

```
void ButtonOnClick(object sender, EventArgs eventArgs)
{
    // Pass the current button press count value to the next activity:
    var valuesForActivity = new Bundle();
    valuesForActivity.PutInt(COUNT_KEY, count);

    // When the user clicks the notification, SecondActivity will start up.
    var resultIntent = new Intent(this, typeof(SecondActivity));

    // Pass some values to SecondActivity:
    resultIntent.PutExtras(valuesForActivity);

    // Construct a back stack for cross-task navigation:
    var stackBuilder = TaskStackBuilder.Create(this);
    stackBuilder.AddParentStack(Class.FromType(typeof(SecondActivity)));
    stackBuilder.AddNextIntent(resultIntent);

    // Create the PendingIntent with the back stack:
    var resultPendingIntent = stackBuilder.GetPendingIntent(0, (int) PendingIntentFlags.UpdateCurrent);

    // Build the notification:
    var builder = new NotificationCompat.Builder(this, CHANNEL_ID)
                  .SetAutoCancel(true) // Dismiss the notification from the notification area when the user
clicks on it
                  .SetContentIntent(resultPendingIntent) // Start up this activity when the user clicks the
intent.
                  .SetContentTitle("Button Clicked") // Set the title
                  .SetNumber(count) // Display the count in the Content Info
                  .SetSmallIcon(Resource.Drawable.ic_stat_button_click) // This is the icon to display
                  .SetContentText($"The button has been clicked {count} times."); // the message to display.

    // Finally, publish the notification:
    var notificationManager = NotificationManagerCompat.From(this);
    notificationManager.Notify(NOTIFICATION_ID, builder.Build());

    // Increment the button press count:
    count++;
}
```

The `OnCreate` method of MainActivity must make the call to create the notification channel and assign the

`ButtonOnClick` method to the `Click` event of the button (replace the delegate event handler provided by the template):

```
protected override void OnCreate(Bundle bundle)
{
    base.OnCreate(bundle);
    SetContentView(Resource.Layout.Main);

    CreateNotificationChannel();

    // Display the "Hello World, Click Me!" button and register its event handler:
    var button = FindViewById<Button>(Resource.Id.MyButton);
    button.Click += ButtonOnClick;
}
```

Create a second activity

Now we need to create another activity that Android will display when the user clicks our notification. Add another Android Activity to your project called **SecondActivity**. Open **SecondActivity.cs** and replace its contents with this code:

```
using System;
using Android.App;
using Android.OS;
using Android.Widget;

namespace LocalNotifications
{
    [Activity(Label = "Second Activity")]
    public class SecondActivity : Activity
    {
        protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);

            // Get the count value passed to us from MainActivity:
            var count = Intent.Extras.GetInt(MainActivity.COUNT_KEY, -1);

            // No count was passed? Then just return.
            if (count <= 0)
            {
                return;
            }

            // Display the count sent from the first activity:
            SetContentView(Resource.Layout.Second);
            var txtView = FindViewById<TextView>(Resource.Id.textView1);
            txtView.Text = $"You clicked the button {count} times in the previous activity.";
        }
    }
}
```

We must also create a resource layout for **SecondActivity**. Add a new **Android Layout** file to your project called **Second.axml**. Edit **Second.axml** and paste in the following layout code:

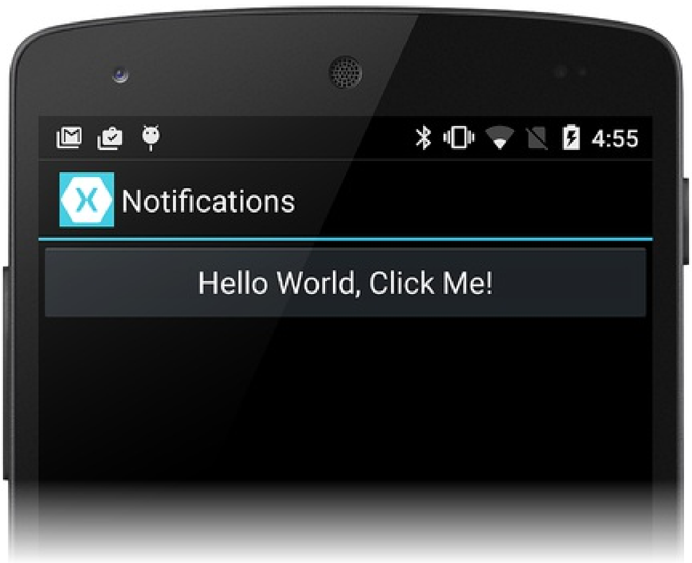
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:minWidth="25px"
    android:minHeight="25px">
    <TextView
        android:text=""
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/textView1" />
</LinearLayout>
```

Add a notification icon

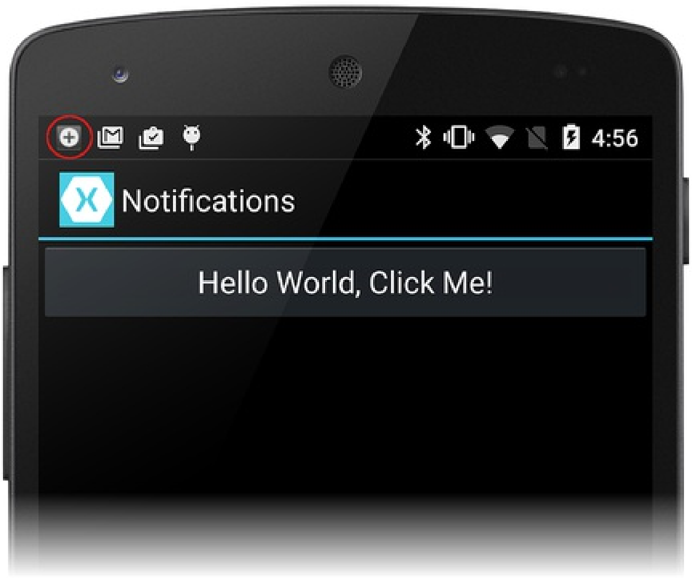
Finally, add a small icon that will appear in the notification area when the notification is launched. You can copy this icon to your project or create your own custom icon. Name the icon file **ic_stat_button_click.png** and copy it to the **Resources/drawable** folder. Remember to use **Add > Existing Item ...** to include this icon file in your project.

Run the application

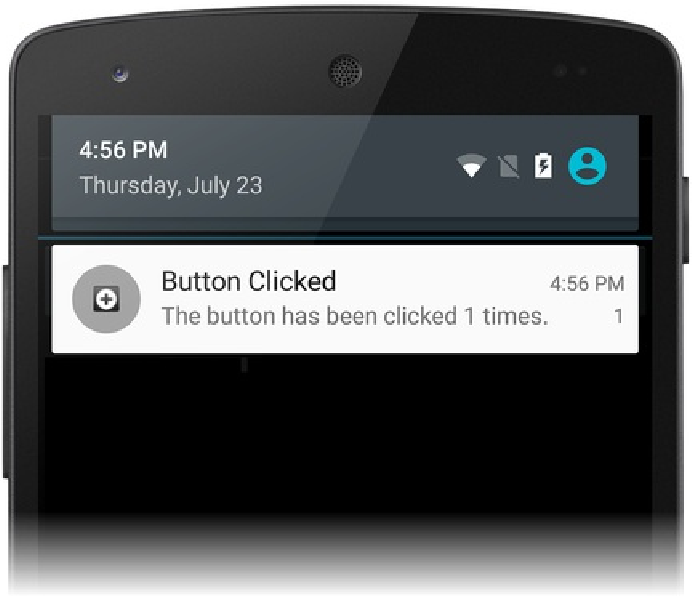
Build and run the application. You should be presented with the first activity, similar to the following screenshot:



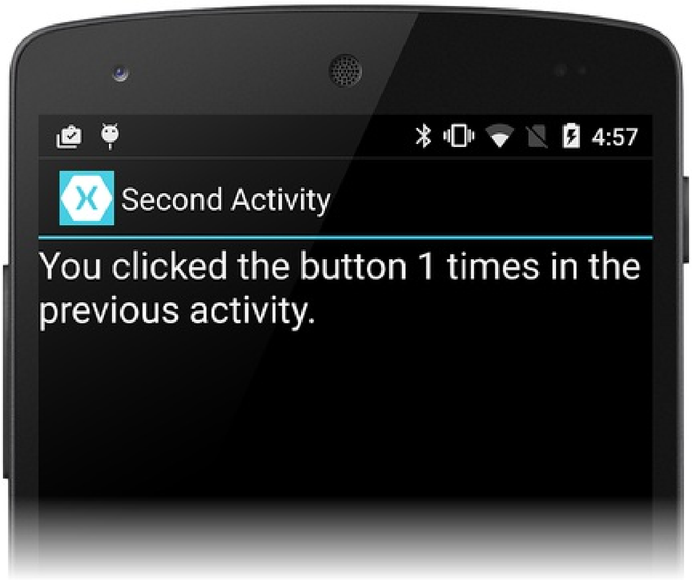
As you click the button, you should notice that the small icon for the notification appears in the notification area:



If you swipe down and expose the notification drawer, you should see the notification:



When you click the notification, it should disappear, and our other activity should be launched – looking somewhat like the following screenshot:



Congratulations! At this point you have completed the Android local notification walkthrough and you have a working sample that you can refer to. There is a lot more to notifications than we have shown here, so if you want more information, take a look at Google's documentation on notifications.

Summary

This walkthrough used `NotificationCompat.Builder` to create and display notifications. It showed a basic example of how to start up a second Activity as a way to respond to user interaction with the notification, and it demonstrated the transfer of data from the first Activity to the second Activity.

Related Links

- LocalNotifications (sample)
- Android Oreo Notification Channels
- Notification
- NotificationManager
- NotificationCompat.Builder
- PendingIntent

Touch and Gestures in Xamarin.Android

11/2/2020 • 2 minutes to read • Edit Online

Touch screens on many of today's devices allow users to quickly and efficiently interact with devices in a natural and intuitive way. This interaction is not limited just to simple touch detection - it is possible to use gestures as well. For example, the pinch-to-zoom gesture is a very common example of this by pinching a part of the screen with two fingers the user can zoom in or out. This guide examines touch and gestures in Android.

Touch Overview

iOS and Android are similar in the ways they handle touch. Both can support multi-touch - many points of contact on the screen - and complex gestures. This guide introduces some of the similarities in concepts, as well as the particularities of implementing touch and gestures on both platforms.

Android uses a `MotionEvent` object to encapsulate touch data, and methods on the View object to listen for touches.

In addition to capturing touch data, both iOS and Android provide means for interpreting patterns of touches into gestures. These gesture recognizers can in turn be used to interpret application-specific commands, such as a rotation of an image or a turn of a page. Android provides a handful of supported gestures, as well as resources to make adding complex custom gestures easy.

Whether you are working on Android or iOS, the choice between touches and gesture recognizers can be a confusing one. This guide recommends that in general, preference should be given to gesture recognizers. Gesture recognizers are implemented as discrete classes, which provide greater separation of concerns and better encapsulation. This makes it easy to share the logic between different views, minimizing the amount of code written.

This guide follows a similar format for each operating system: first, the platform's touch APIs are introduced and explained, as they are the foundation on which touch interactions are built. Then, we dive into the world of gesture recognizers – first by exploring some common gestures, and finishing up with creating custom gestures for applications. Finally, you'll see how to track individual fingers using low-level touch tracking to create a finger-paint program.

Sections

- Touch in Android
- Walkthrough: Using Touch in Android
- Multi-Touch tracking

Summary

In this guide we examined touch in Android. For both operating systems, we learned how to enable touch and how to respond to the touch events. Next, we learned about gestures and some of the gesture recognizers that both Android and iOS provide to handle some of the more common scenarios. We examined how to create custom gestures and implement them in applications. A walkthrough demonstrated the concepts and APIs for each operating system in action, and you also saw how to track individual fingers.

Related Links

- Android Touch Start (sample)

- Android Touch Final (sample)
- FingerPaint (sample)

Touch in Android

7/8/2021 • 6 minutes to read • Edit Online

Much like iOS, Android creates an object that holds data about the user's physical interaction with the screen – an `Android.View.MotionEvent` object. This object holds data such as what action is performed, where the touch took place, how much pressure was applied, etc. A `MotionEvent` object breaks down the movement into to the following values:

- An action code that describes the type of motion, such as the initial touch, the touch moving across the screen, or the touch ending.
- A set of axis values that describe the position of the `MotionEvent` and other movement properties such as where the touch is taking place, when the touch took place, and how much pressure was used. The axis values may be different depending on the device, so the previous list does not describe all axis values.

The `MotionEvent` object will be passed to an appropriate method in an application. There are three ways for a Xamarin.Android application to respond to a touch event:

- *Assign an event handler to* `View.Touch` - The `Android.Views.View` class has an `EventHandler<View.TouchEventArgs>` which applications can assign a handler to. This is typical .NET behavior.
- *Implementing* `View.IOnTouchListener` - Instances of this interface may be assigned to a view object using the View. `SetOnListener` method.This is functionally equivalent to assigning an event handler to the `View.Touch` event. If there is some common or shared logic that many different views may need when they are touched, it will be more efficient to create a class and implement this method than to assign each view its own event handler.
- *Override* `View.OnTouchEvent` - All views in Android subclass `Android.Views.View`. When a View is touched, Android will call the `OnTouchEvent` and pass it a `MotionEvent` object as a parameter.

> **NOTE**
>
> Not all Android devices support touch screens.

Adding the following tag to your manifest file causes Google Play to only display your app to those devices that are touch enabled:

```
<uses-configuration android:reqTouchScreen="finger" />
```

Gestures

A gesture is a hand-drawn shape on the touch screen. A gesture can have one or more strokes to it, each stroke consisting of a sequence of points created by a different point of contact with the screen. Android can support many different types of gestures, from a simple fling across the screen to complex gestures that involve multi-touch.

Android provides the `Android.Gestures` namespace specifically for managing and responding to gestures. At the heart of all gestures is a special class called `Android.Gestures.GestureDetector`. As the name implies, this class will listen for gestures and events based on `MotionEvents` supplied by the operating system.

To implement a gesture detector, an Activity must instantiate a `GestureDetector` class and provide an instance of `IOnGestureListener`, as illustrated by the following code snippet:

```
GestureOverlayView.IOnGestureListener myListener = new MyGestureListener();
_gestureDetector = new GestureDetector(this, myListener);
```

An Activity must also implement the OnTouchEvent and pass the MotionEvent to the gesture detector. The following code snippet shows an example of this:

```
public override bool OnTouchEvent(MotionEvent e)
{
    // This method is in an Activity
    return _gestureDetector.OnTouchEvent(e);
}
```

When an instance of `GestureDetector` identifies a gesture of interest, it will notify the activity or application either by raising an event or through a callback provided by `GestureDetector.IOnGestureListener`. This interface provides six methods for the various gestures:

- *OnDown* - Called when a tap occurs but is not released.
- *OnFling* - Called when a fling occurs and provides data on the start and end touch that triggered the event.
- *OnLongPress* - Called when a long press occurs.
- *OnScroll* - Called when a scroll event occurs.
- *OnShowPress* - Called after an OnDown has occurred and a move or up event has not been performed.
- *OnSingleTapUp* - Called when a single tap occurs.

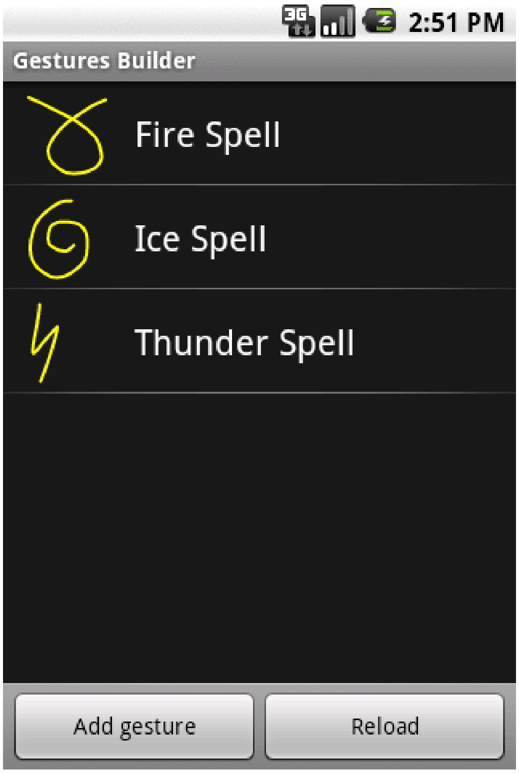
In many cases applications may only be interested in a subset of gestures. In this case, applications should extend the class GestureDetector.SimpleOnGestureListener and override the methods that correspond to the events that they are interested in.

Custom Gestures

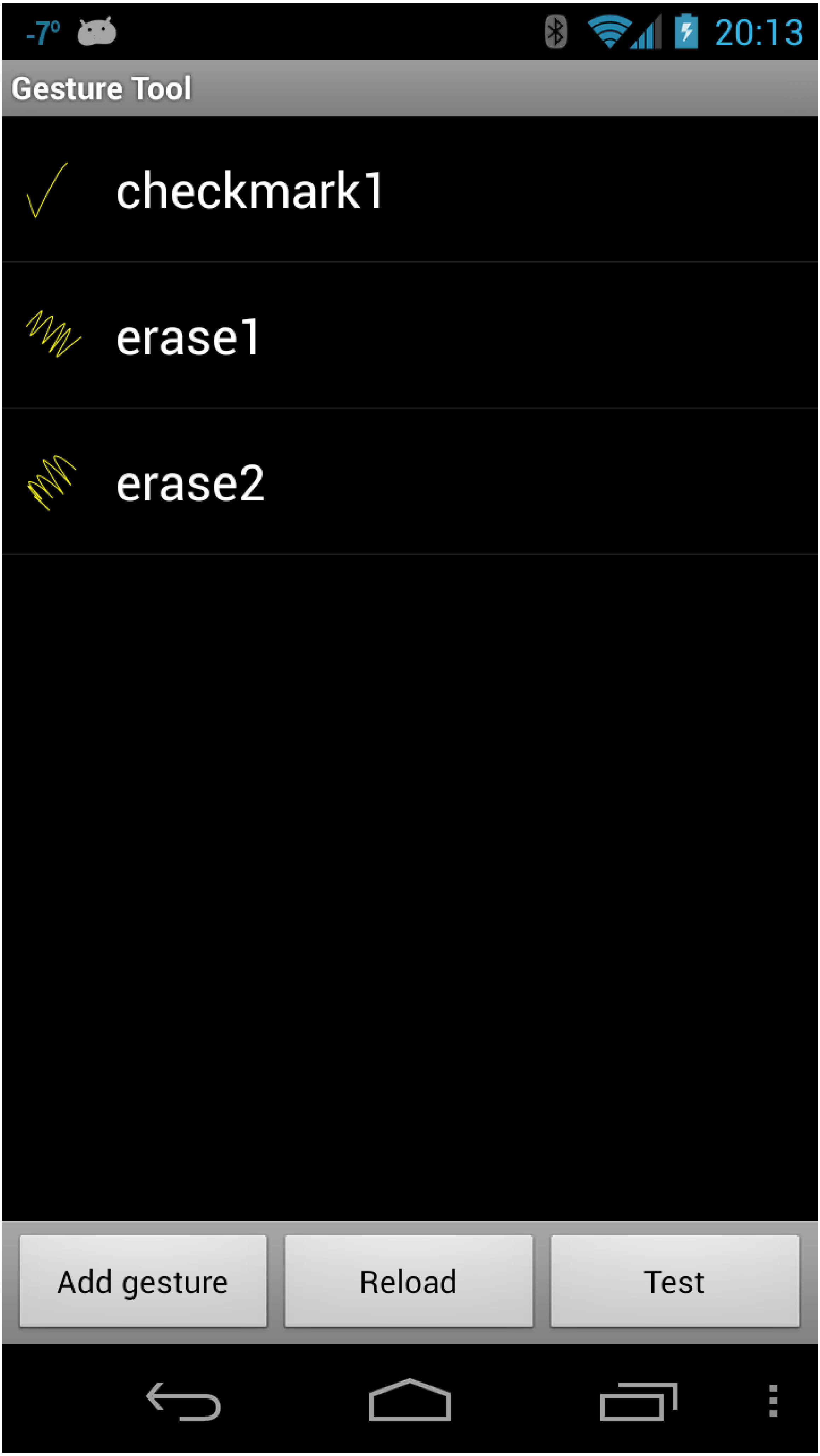
Gestures are a great way for users to interact with an application. The APIs we have seen so far would suffice for simple gestures, but might prove a bit onerous for more complicated gestures. To help with more complicated gestures, Android provides another set of API's in the Android.Gestures namespace that will ease some of the burden associated with custom gestures.

Creating Custom Gestures

Since Android 1.6, the Android SDK comes with an application pre-installed on the emulator called Gestures Builder. This application allows a developer to create pre-defined gestures that can be embedded in an application. The following screen shot shows an example of Gestures Builder:



An improved version of this application called Gesture Tool can be found Google Play. Gesture Tool is very much like Gestures Builder except that it allows you to test gestures after they have been created. This next screenshot shows Gestures Builder:


-7°
20:13
Gesture Tool
checkmark1
erase1
erase2
Add gesture
Reload
Test

Gesture Tool is a bit more useful for creating custom gestures as it allows the gestures to be tested as they are being created and is easily available through Google Play.

Gesture Tool allows you create a gesture by drawing on the screen and assigning a name. After the gestures are created they are saved in a binary file on the SD card of your device. This file needs to be retrieved from the device, and then packaged with an application in the folder /Resources/raw. This file can be retrieved from the emulator using the Android Debug Bridge. The following example shows copying the file from a Galaxy Nexus to the Resource directory of an application:

```
$ adb pull /storage/sdcard0/gestures <projectdirectory>/Resources/raw
```

Once you have retrieved the file it must be packaged with your application inside the directory /Resources/raw. The easiest way to use this gesture file is to load the file into a GestureLibrary, as shown in the following snippet:

```
GestureLibrary myGestures = GestureLibraries.FromRawResources(this, Resource.Raw.gestures);
if (!myGestures.Load())
{
    // The library didn't load, so close the activity.
    Finish();
}
```

Using Custom Gestures

To recognize custom gestures in an Activity, it must have an Android.Gesture.GestureOverlay object added to its layout. The following code snippet shows how to programmatically add a GestureOverlayView to an Activity:

```
GestureOverlayView gestureOverlayView = new GestureOverlayView(this);
gestureOverlayView.AddOnGesturePerformedListener(this);
SetContentView(gestureOverlayView);
```

The following XML snippet shows how to add a GestureOverlayView declaratively:

```
<android.gesture.GestureOverlayView
    android:id="@+id/gestures"
    android:layout_width="match_parent "
    android:layout_height="match_parent" />
```

The `GestureOverlayView` has several events that will be raised during the process of drawing a gesture. The most interesting event is `GesturePerformed`. This event is raised when the user has completed drawing their gesture.

When this event is raised, the Activity asks a `GestureLibrary` to try and match the gesture that the user with one of the gestures created by Gesture Tool. `GestureLibrary` will return a list of Prediction objects.

Each Prediction object holds a score and name of one of the gestures in the `GestureLibrary`. The higher the score, the more likely the gesture named in the Prediction matches the gesture drawn by the user. Generally speaking, scores lower than 1.0 are considered poor matches.

The following code shows an example of matching a gesture:

```
private void GestureOverlayViewOnGesturePerformed(object sender,
GestureOverlayView.GesturePerformedEventArgs gesturePerformedEventArgs)
{
    // In this example _gestureLibrary was instantiated in OnCreate
    IEnumerable<Prediction> predictions = from p in
_gestureLibrary.Recognize(gesturePerformedEventArgs.Gesture)
    orderby p.Score descending
    where p.Score > 1.0
    select p;
    Prediction prediction = predictions.FirstOrDefault();

    if (prediction == null)
    {
        Log.Debug(GetType().FullName, "Nothing matched the user's gesture.");
        return;
    }

    Toast.MakeText(this, prediction.Name, ToastLength.Short).Show();
}
```

With this done, you should have an understanding of how to use touch and gestures in a Xamarin.Android application. Let us now move on to a walkthrough and see all of the concepts in a working sample application.

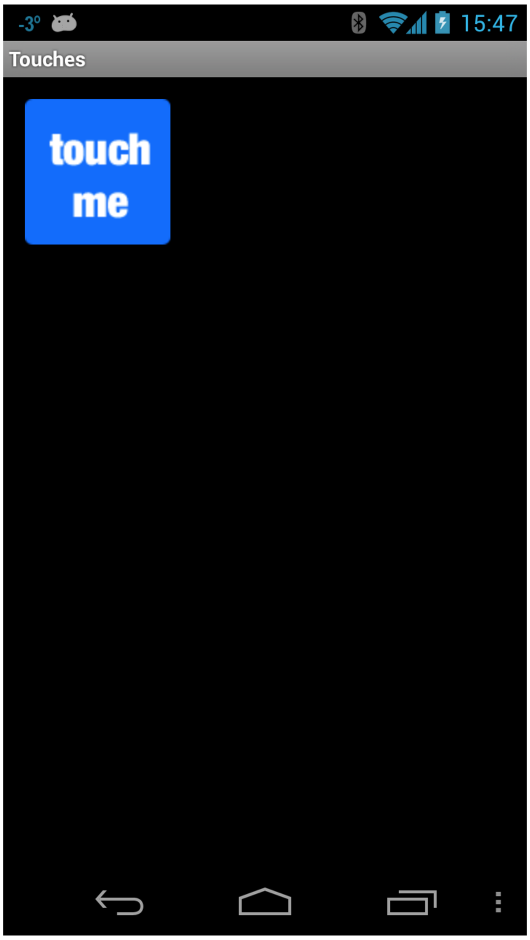
Related Links

- Android Touch Start (sample)
- Android Touch Final (sample)

Walkthrough - Using Touch in Android

7/8/2021 • 8 minutes to read • Edit Online

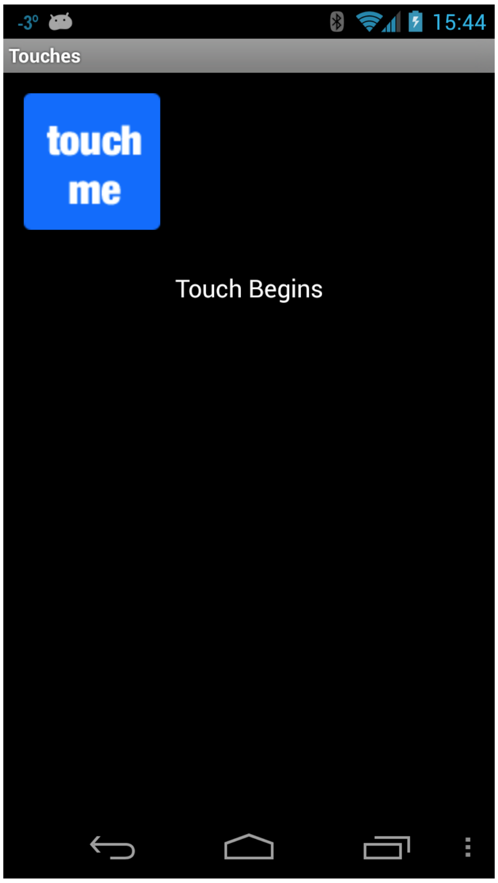
Let us see how to use the concepts from the previous section in a working application. We will create an application with four activities. The first activity will be a menu or a switchboard that will launch the other activities to demonstrate the various APIs. The following screenshot shows the main activity:



The first Activity, Touch Sample, will show how to use event handlers for touching the Views. The Gesture Recognizer activity will demonstrate how to subclass `Android.View.Views` and handle events as well as show how to handle pinch gestures. The third and final activity, **Custom Gesture**, will show how use custom gestures. To make things easier to follow and absorb, we'll break this walkthrough up into sections, with each section focusing on one of the Activities.

Touch Sample Activity

- Open the project **TouchWalkthrough_Start**. The **MainActivity** is all set to go – it is up to us to implement the touch behaviour in the activity. If you run the application and click **Touch Sample**, the following activity should start up:



- Now that we have confirmed that the Activity starts up, open the file **TouchActivity.cs** and add a handler for the `Touch` event of the `ImageView`:

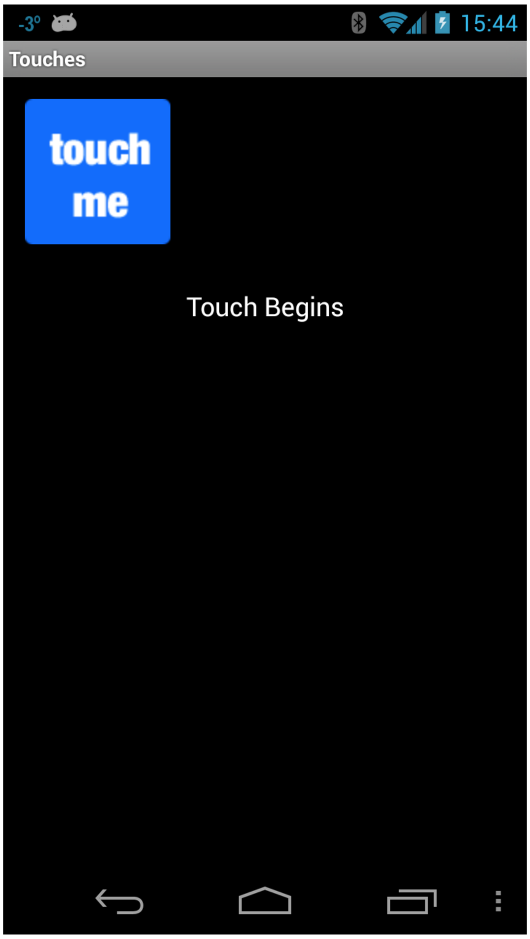
```
_touchMeImageView.Touch += TouchMeImageViewOnTouch;
```

- Next, add the following method to **TouchActivity.cs**:

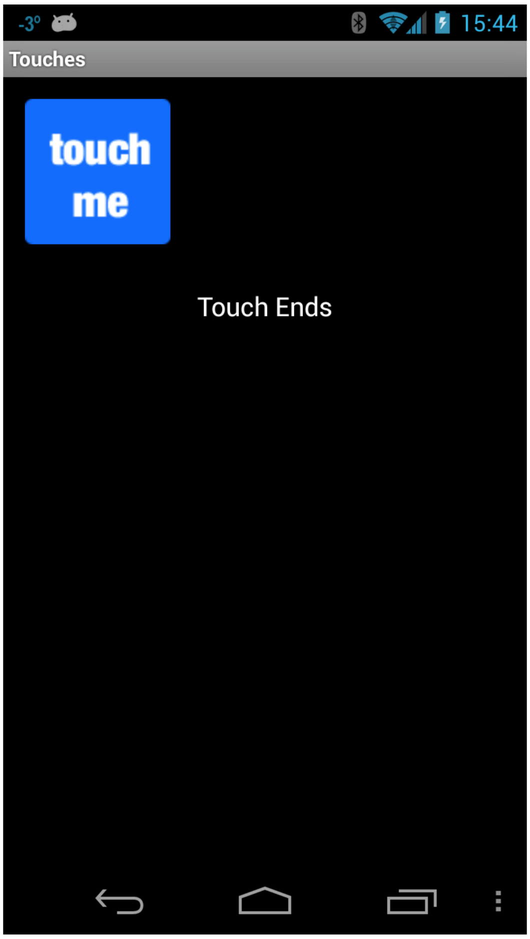
```
private void TouchMeImageViewOnTouch(object sender, View.TouchEventArgs touchEventArgs)
{
    string message;
    switch (touchEventArgs.Event.Action & MotionEventActions.Mask)
    {
        case MotionEventActions.Down:
        case MotionEventActions.Move:
        message = "Touch Begins";
        break;

        case MotionEventActions.Up:
        message = "Touch Ends";
        break;

        default:
        message = string.Empty;
        break;
    }

    _touchInfoTextView.Text = message;
}
```

Notice in the code above that we treat the `Move` and `Down` action as the same. This is because even though the user may not lift their finger off the `ImageView`, it may move around or the pressure exerted by the user may change. These types of changes will generate a `Move` action.

Each time the user touches the `ImageView`, the `Touch` event will be raised and our handler will display the message **Touch Begins** on the screen, as shown in the following screenshot:



As long as the user is touching the `ImageView`, **Touch Begins** will be displayed in the `TextView`. When the user is no longer touching the `ImageView`, the message **Touch Ends** will be displayed in the `TextView`, as shown in the following screenshot:



Gesture Recognizer Activity

Now lets implement the Gesture Recognizer activity. This activity will demonstrate how to drag a view around the screen and illustrate one way to implement pinch-to-zoom.

- Add a new Activity to the application called `GestureRecognizer`. Edit the code for this activity so that it resembles the following code:

```
public class GestureRecognizerActivity : Activity
{
    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);
        View v = new GestureRecognizerView(this);
        SetContentView(v);
    }
}
```

- Add a new Android view to the project, and name it `GestureRecognizerView`. Add the following variables to this class:

```
private static readonly int InvalidPointerId = -1;

private readonly Drawable _icon;
private readonly ScaleGestureDetector _scaleDetector;

private int _activePointerId = InvalidPointerId;
private float _lastTouchX;
private float _lastTouchY;
private float _posX;
private float _posY;
private float _scaleFactor = 1.0f;
```

- Add the following constructor to `GestureRecognizerView`. This constructor will add an `ImageView` to our activity. At this point the code still will not compile – we need to create the class `MyScaleListener` that will help with resizing the `ImageView` when the user pinches it:

```
public GestureRecognizerView(Context context): base(context, null, 0)
{
    _icon = context.Resources.GetDrawable(Resource.Drawable.Icon);
    _icon.SetBounds(0, 0, _icon.IntrinsicWidth, _icon.IntrinsicHeight);
    _scaleDetector = new ScaleGestureDetector(context, new MyScaleListener(this));
}
```

- To draw the image on our activity, we need to override the `OnDraw` method of the View class as shown in the following snippet. This code will move the `ImageView` to the position specified by `_posX` and `_posY` as well as resize the image according to the scaling factor:

```
protected override void OnDraw(Canvas canvas)
{
    base.OnDraw(canvas);
    canvas.Save();
    canvas.Translate(_posX, _posY);
    canvas.Scale(_scaleFactor, _scaleFactor);
    _icon.Draw(canvas);
    canvas.Restore();
}
```

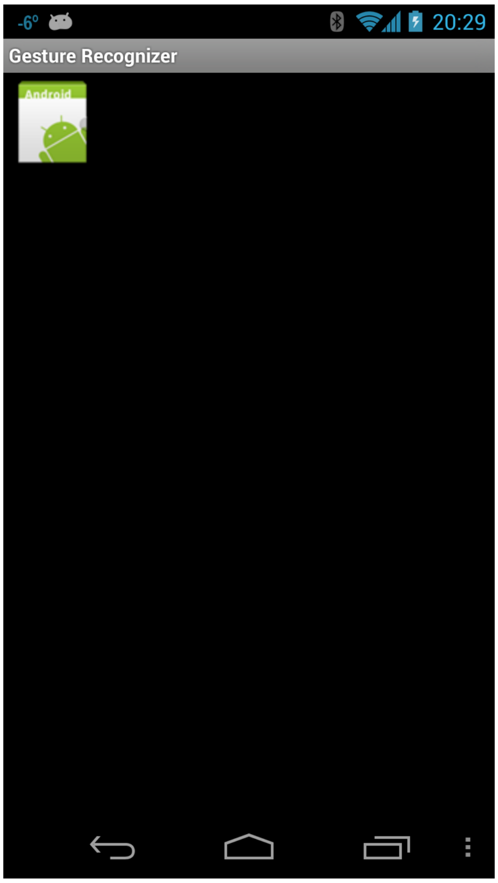
- Next we need to update the instance variable `_scaleFactor` as the user pinches the `ImageView`. We will add a class called `MyScaleListener`. This class will listen for the scale events that will be raised by Android when the user pinches the `ImageView`. Add the following inner class to `GestureRecognizerView`. This class is a `ScaleGesture.SimpleOnScaleGestureListener`. This class is a convenience class that listeners can subclass when you are interested in a subset of gestures:

```
private class MyScaleListener : ScaleGestureDetector.SimpleOnScaleGestureListener
{
    private readonly GestureRecognizerView _view;

    public MyScaleListener(GestureRecognizerView view)
    {
        _view = view;
    }

    public override bool OnScale(ScaleGestureDetector detector)
    {
        _view._scaleFactor *= detector.ScaleFactor;

        // put a limit on how small or big the image can get.
        if (_view._scaleFactor > 5.0f)
        {
            _view._scaleFactor = 5.0f;
        }
        if (_view._scaleFactor < 0.1f)
        {
            _view._scaleFactor = 0.1f;
        }

        _view.Invalidate();
        return true;
    }
}
```

- The next method we need to override in `GestureRecognizerView` is `OnTouchEvent`. The following code lists the full implementation of this method. There is a lot of code here, so lets take a minute and look what is going on here. The first thing this method does is scale the icon if necessary – this is handled by calling `_scaleDetector.OnTouchEvent`. Next we try to figure out what action called this method:
    - If the user touched the screen with, we record the X and Y positions and the ID of the first pointer that touched the screen.
    - If the user moved their touch on the screen, then we figure out how far the user moved the pointer.
    - If the user has lifted his finger off the screen, then we will stop tracking the gestures.

```
public override bool OnTouchEvent(MotionEvent ev)
{
    _scaleDetector.OnTouchEvent(ev);

    MotionEventActions action = ev.Action & MotionEventActions.Mask;
    int pointerIndex;

    switch (action)
    {
        case MotionEventActions.Down:
        _lastTouchX = ev.GetX();
        _lastTouchY = ev.GetY();
        _activePointerId = ev.GetPointerId(0);
        break;

        case MotionEventActions.Move:
        pointerIndex = ev.FindPointerIndex(_activePointerId);
        float x = ev.GetX(pointerIndex);
        float y = ev.GetY(pointerIndex);
        if (!_scaleDetector.IsInProgress)
        {
            // Only move the ScaleGestureDetector isn't already processing a gesture.
            float deltaX = x - _lastTouchX;
            float deltaY = y - _lastTouchY;
            _posX += deltaX;
            _posY += deltaY;
            Invalidate();
        }

        _lastTouchX = x;
        _lastTouchY = y;
        break;

        case MotionEventActions.Up:
        case MotionEventActions.Cancel:
        // We no longer need to keep track of the active pointer.
        _activePointerId = InvalidPointerId;
        break;

        case MotionEventActions.PointerUp:
        // check to make sure that the pointer that went up is for the gesture we're tracking.
        pointerIndex = (int) (ev.Action & MotionEventActions.PointerIndexMask) >> (int)
MotionEventActions.PointerIndexShift;
        int pointerId = ev.GetPointerId(pointerIndex);
        if (pointerId == _activePointerId)
        {
            // This was our active pointer going up. Choose a new
            // action pointer and adjust accordingly
            int newPointerIndex = pointerIndex == 0 ? 1 : 0;
            _lastTouchX = ev.GetX(newPointerIndex);
            _lastTouchY = ev.GetY(newPointerIndex);
            _activePointerId = ev.GetPointerId(newPointerIndex);
        }
        break;

    }
    return true;
}
```

- Now run the application, and start the Gesture Recognizer activity. When it starts the screen should look something like the screenshot below:



- Now touch the icon, and drag it around the screen. Try the pinch-to-zoom gesture. At some point your screen may look something like the following screen shot:

At this point you should give yourself a pat on the back: you have just implemented pinch-to-zoom in an Android application! Take a quick break and lets move on to the third and final Activity in this walkthrough – using custom gestures.

Custom Gesture Activity

The final screen in this walkthrough will use custom gestures.

For the purposes of this Walkthrough, the gestures library has already been created using Gesture Tool and added to the project in the file **Resources/raw/gestures**. With this bit of housekeeping out of the way, lets get on with the final Activity in the walkthrough.

- Add a layout file named **custom_gesture_layout.axml** to the project with the following contents. The project already has all the images in the **Resources** folder:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1" />
    <ImageView
        android:src="@drawable/check_me"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="3"
        android:id="@+id/imageView1"
        android:layout_gravity="center_vertical" />
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1" />
</LinearLayout>
```

- Next add a new Activity to the project and name it `CustomGestureRecognizerActivity.cs`. Add two instance variables to the class, as showing in the following two lines of code:

```
private GestureLibrary _gestureLibrary;
private ImageView _imageView;
```

- Edit the `OnCreate` method of the this Activity so that it resembles the following code. Lets take a minute to explain what is going on in this code. The first thing we do is instantiate a `GestureOverlayView` and set that as the root view of the Activity. We also assign an event handler to the `GesturePerformed` event of `GestureOverlayView`. Next we inflate the layout file that was created earlier, and add that as a child view of the `GestureOverlayView`. The final step is to initialize the variable `_gestureLibrary` and load the gestures file from the application resources. If the gestures file cannot be loaded for some reason, there is not much this Activity can do, so it is shutdown:

```
protected override void OnCreate(Bundle bundle)
{
    base.OnCreate(bundle);

    GestureOverlayView gestureOverlayView = new GestureOverlayView(this);
    SetContentView(gestureOverlayView);
    gestureOverlayView.GesturePerformed += GestureOverlayViewOnGesturePerformed;

    View view = LayoutInflater.Inflate(Resource.Layout.custom_gesture_layout, null);
    _imageView = view.FindViewById<ImageView>(Resource.Id.imageView1);
    gestureOverlayView.AddView(view);

    _gestureLibrary = GestureLibraries.FromRawResource(this, Resource.Raw.gestures);
    if (!_gestureLibrary.Load())
    {
        Log.Wtf(GetType().FullName, "There was a problem loading the gesture library.");
        Finish();
    }
}
```

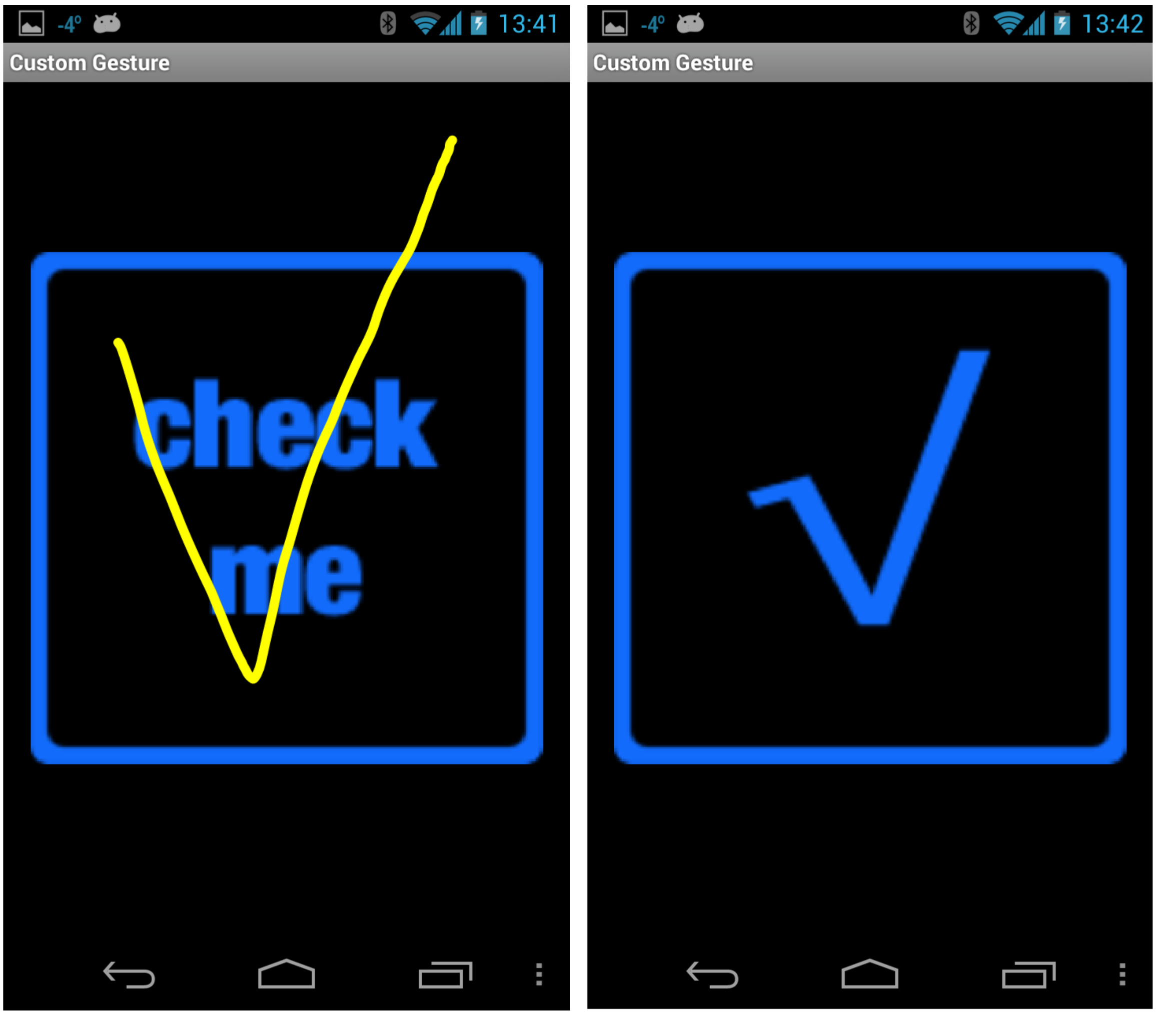
- The final thing we need to do implement the method `GestureOverlayViewOnGesturePerformed` as shown in the following code snippet. When the `GestureOverlayView` detects a gesture, it calls back to this method. The first thing we try to get an `IList<Prediction>` objects that match the gesture by calling `_gestureLibrary.Recognize()`. We use a bit of LINQ to get the `Prediction` that has the highest score for the gesture.

  If there was no matching gesture with a high enough score, then the event handler exits without doing anything. Otherwise we check the name of the prediction and change the image being displayed based on the name of the gesture:

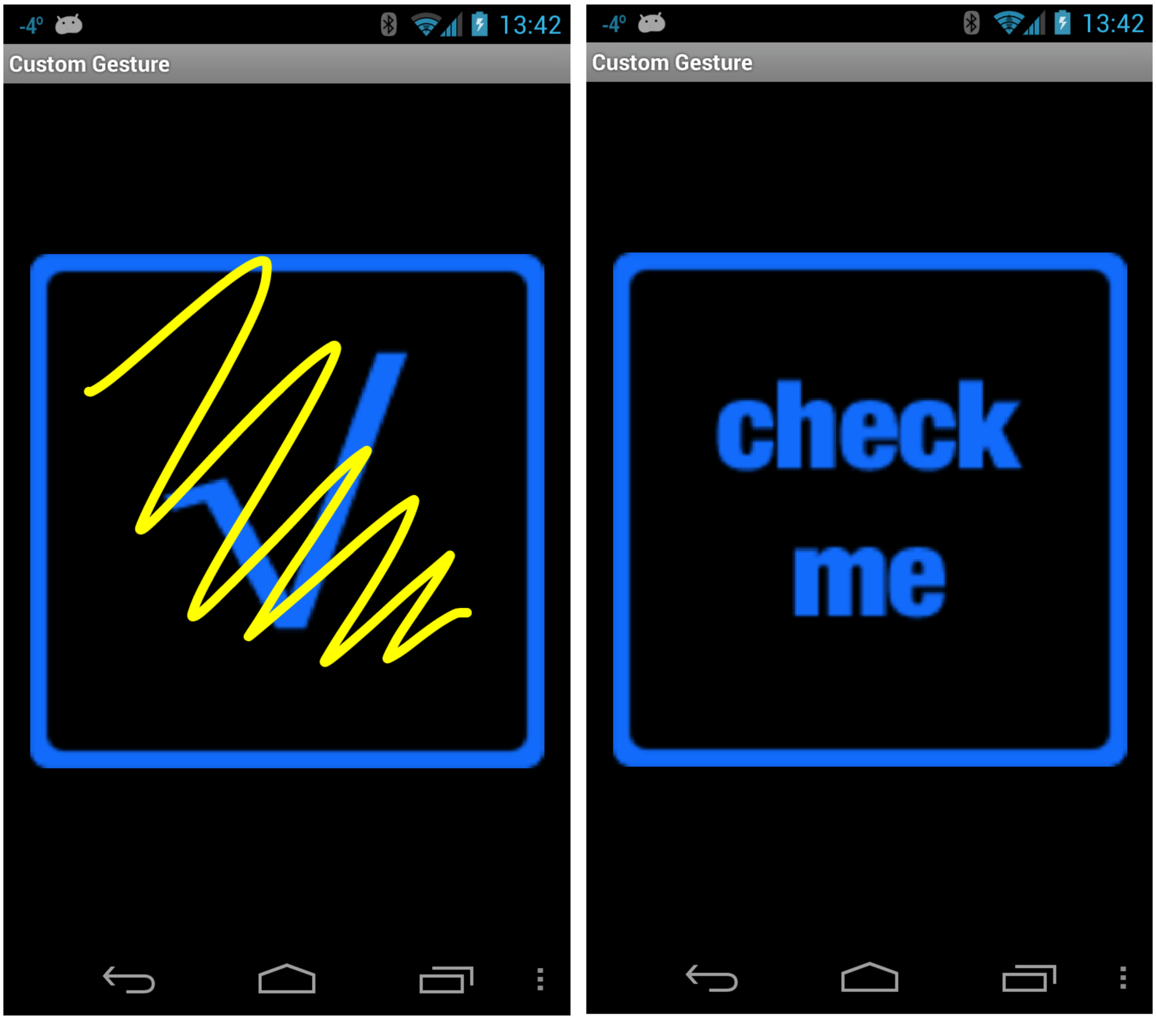
```
private void GestureOverlayViewOnGesturePerformed(object sender,
GestureOverlayView.GesturePerformedEventArgs gesturePerformedEventArgs)
{
    IEnumerable<Prediction> predictions = from p in
_gestureLibrary.Recognize(gesturePerformedEventArgs.Gesture)
    orderby p.Score descending
    where p.Score > 1.0
    select p;
    Prediction prediction = predictions.FirstOrDefault();

    if (prediction == null)
    {
        Log.Debug(GetType().FullName, "Nothing seemed to match the user's gesture, so don't do
anything.");
        return;
    }

    Log.Debug(GetType().FullName, "Using the prediction named {0} with a score of {1}.",
prediction.Name, prediction.Score);

    if (prediction.Name.StartsWith("checkmark"))
    {
        _imageView.SetImageResource(Resource.Drawable.checked_me);
    }
    else if (prediction.Name.StartsWith("erase", StringComparison.OrdinalIgnoreCase))
    {
        // Match one of our "erase" gestures
        _imageView.SetImageResource(Resource.Drawable.check_me);
    }
}
```

- Run the application and start up the Custom Gesture Recognizer activity. It should look something like the following screenshot:

Now draw a checkmark on the screen, and the bitmap being displayed should look something like that shown in the next screenshots:



Finally, draw a scribble on the screen. The checkbox should change back to its original image as shown in these screenshots:



You now have an understanding of how to integrate touch and gestures in an Android application using Xamarin.Android.

Related Links

- Android Touch Start (sample)
- Android Touch Final (sample)

Multi-Touch Finger Tracking

7/8/2021 • 5 minutes to read • Edit Online

This topic demonstrates how to track touch events from multiple fingers

There are times when a multi-touch application needs to track individual fingers as they move simultaneously on the screen. One typical application is a finger-paint program. You want the user to be able to draw with a single finger, but also to draw with multiple fingers at once. As your program processes multiple touch events, it needs to distinguish which events correspond to each finger. Android supplies an ID code for this purpose, but obtaining and handling that code can be a little tricky.

For all the events associated with a particular finger, the ID code remains the same. The ID code is assigned when a finger first touches the screen, and becomes invalid after the finger lifts from the screen. These ID codes are generally very small integers, and Android reuses them for later touch events.

Almost always, a program that tracks individual fingers maintains a dictionary for touch tracking. The dictionary key is the ID code that identifies a particular finger. The dictionary value depends on the application. In the FingerPaint program, each finger stroke (from touch to release) is associated with an object that contains all the information necessary to render the line drawn with that finger. The program defines a small `FingerPaintPolyline` class for this purpose:

```
class FingerPaintPolyline
{
    public FingerPaintPolyline()
    {
        Path = new Path();
    }

    public Color Color { set; get; }

    public float StrokeWidth { set; get; }

    public Path Path { private set; get; }
}
```

Each polyline has a color, a stroke width, and an Android graphics `Path` object to accumulate and render multiple points of the line as it's being drawn.

The remainder of the code shown below is contained in a `View` derivative named `FingerPaintCanvasView`. That class maintains a dictionary of objects of type `FingerPaintPolyline` during the time that they are actively being drawn by one or more fingers:

```
Dictionary<int, FingerPaintPolyline> inProgressPolylines = new Dictionary<int, FingerPaintPolyline>();
```

This dictionary allows the view to quickly obtain the `FingerPaintPolyline` information associated with a particular finger.

The `FingerPaintCanvasView` class also maintains a `List` object for the polylines that have been completed:

```
List<FingerPaintPolyline> completedPolylines = new List<FingerPaintPolyline>();
```

The objects in this `List` are in the same order that they were drawn.

`FingerPaintCanvasView` overrides two methods defined by `View`: `OnDraw` and `OnTouchEvent`. In its `OnDraw` override, the view draws the completed polylines and then draws the in-progress polylines.

The override of the `OnTouchEvent` method begins by obtaining a `pointerIndex` value from the `ActionIndex` property. This `ActionIndex` value differentiates between multiple fingers, but it is not consistent across multiple events. For that reason, you use the `pointerIndex` to obtain the pointer `id` value from the `GetPointerId` method. This ID *is* consistent across multiple events:

```
public override bool OnTouchEvent(MotionEvent args)
{
    // Get the pointer index
    int pointerIndex = args.ActionIndex;

    // Get the id to identify a finger over the course of its progress
    int id = args.GetPointerId(pointerIndex);

    // Use ActionMasked here rather than Action to reduce the number of possibilities
    switch (args.ActionMasked)
    {
        // ...
    }

    // Invalidate to update the view
    Invalidate();

    // Request continued touch input
    return true;
}
```

Notice that the override uses the `ActionMasked` property in the `switch` statement rather than the `Action` property. Here's why:

When you're dealing with multi-touch, the `Action` property has a value of `MotionEventsAction.Down` for the first finger to touch the screen, and then values of `Pointer2Down` and `Pointer3Down` as the second and third fingers also touch the screen. As the fourth and fifth fingers make contact, the `Action` property has numeric values that don't even correspond to members of the `MotionEventsAction` enumeration! You'd need to examine the values of bit flags in the values to interpret what they mean.

Similarly, as the fingers leave contact with the screen, the `Action` property has values of `Pointer2Up` and `Pointer3Up` for the second and third fingers, and `Up` for the first finger.

The `ActionMasked` property takes on a fewer number of values because it's intended to be used in conjunction with the `ActionIndex` property to differentiate between multiple fingers. When fingers touch the screen, the property can only equal `MotionEventActions.Down` for the first finger and `PointerDown` for subsequent fingers. As the fingers leave the screen, `ActionMasked` has values of `Pointer1Up` for the subsequent fingers and `Up` for the first finger.

When using `ActionMasked`, the `ActionIndex` distinguishes among the subsequent fingers to touch and leave the screen, but you usually don't need to use that value except as an argument to other methods in the `MotionEvent` object. For multi-touch, one of the most important of these methods is `GetPointerId` called in the code above. That method returns a value that you can use for a dictionary key to associate particular events to fingers.

The `OnTouchEvent` override in the FingerPaint program processes the `MotionEventActions.Down` and `PointerDown` events identically by creating a new `FingerPaintPolyline` object and adding it to the dictionary:

```
public override bool OnTouchEvent(MotionEvent args)
{
    // Get the pointer index
    int pointerIndex = args.ActionIndex;

    // Get the id to identify a finger over the course of its progress
    int id = args.GetPointerId(pointerIndex);

    // Use ActionMasked here rather than Action to reduce the number of possibilities
    switch (args.ActionMasked)
    {
        case MotionEventActions.Down:
        case MotionEventActions.PointerDown:

            // Create a Polyline, set the initial point, and store it
            FingerPaintPolyline polyline = new FingerPaintPolyline
            {
                Color = StrokeColor,
                StrokeWidth = StrokeWidth
            };

            polyline.Path.MoveTo(args.GetX(pointerIndex),
                                 args.GetY(pointerIndex));

            inProgressPolylines.Add(id, polyline);
            break;
        // ...
    }
    // ...
}
```

Notice that the `pointerIndex` is also used to obtain the position of the finger within the view. All the touch information is associated with the `pointerIndex` value. The `id` uniquely identifies fingers across multiple messages, so that's used to create the dictionary entry.

Similarly, the `OnTouchEvent` override also handles the `MotionEventActions.Up` and `Pointer1Up` identically by transferring the completed polyline to the `completedPolylines` collection so they can be drawn during the `OnDraw` override. The code also removes the `id` entry from the dictionary:

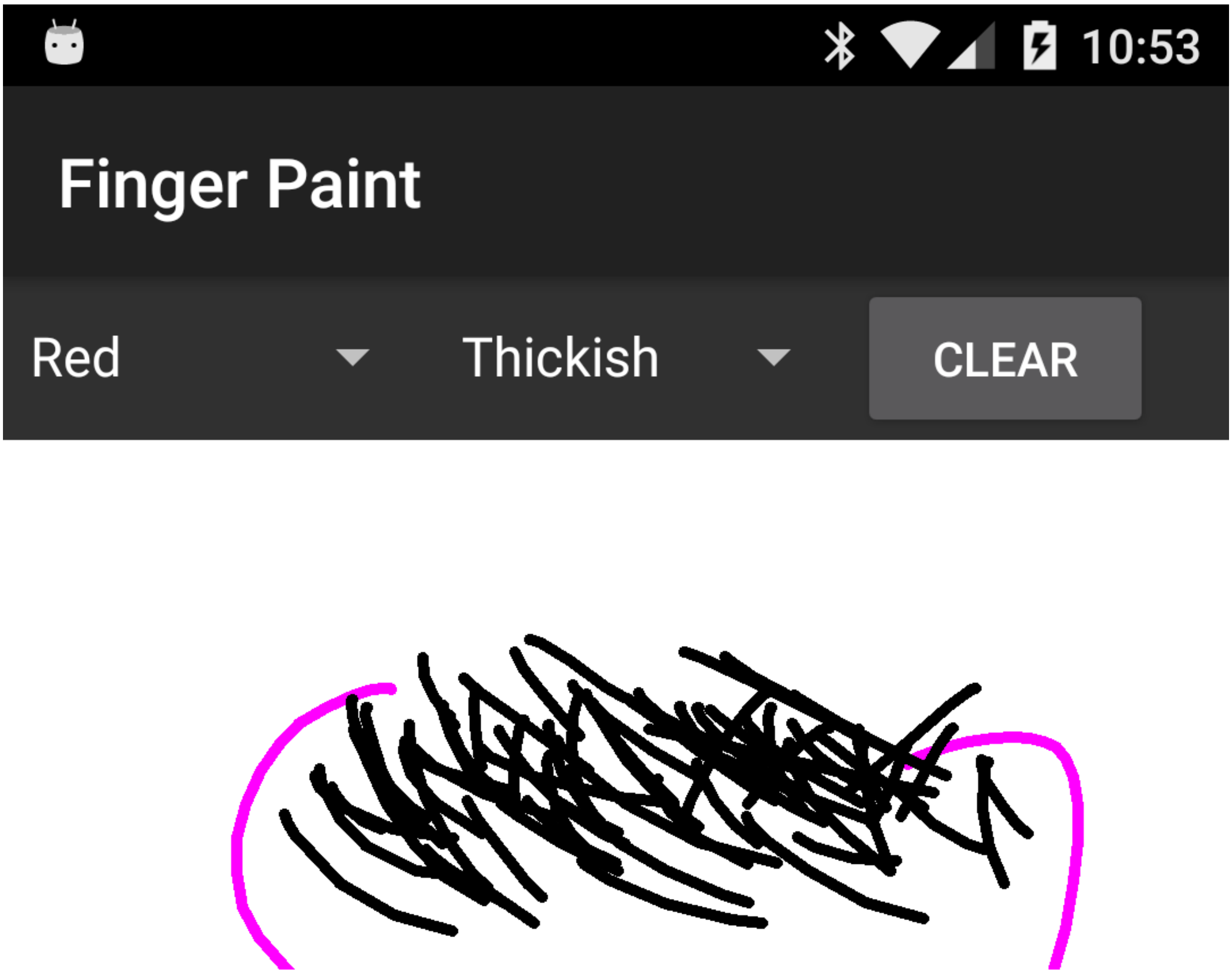
```
public override bool OnTouchEvent(MotionEvent args)
{
    // ...
    switch (args.ActionMasked)
    {
        // ...
        case MotionEventActions.Up:
        case MotionEventActions.Pointer1Up:

            inProgressPolylines[id].Path.LineTo(args.GetX(pointerIndex),
                                                args.GetY(pointerIndex));

            // Transfer the in-progress polyline to a completed polyline
            completedPolylines.Add(inProgressPolylines[id]);
            inProgressPolylines.Remove(id);
            break;

        case MotionEventActions.Cancel:
            inProgressPolylines.Remove(id);
            break;
    }
    // ...
}
```

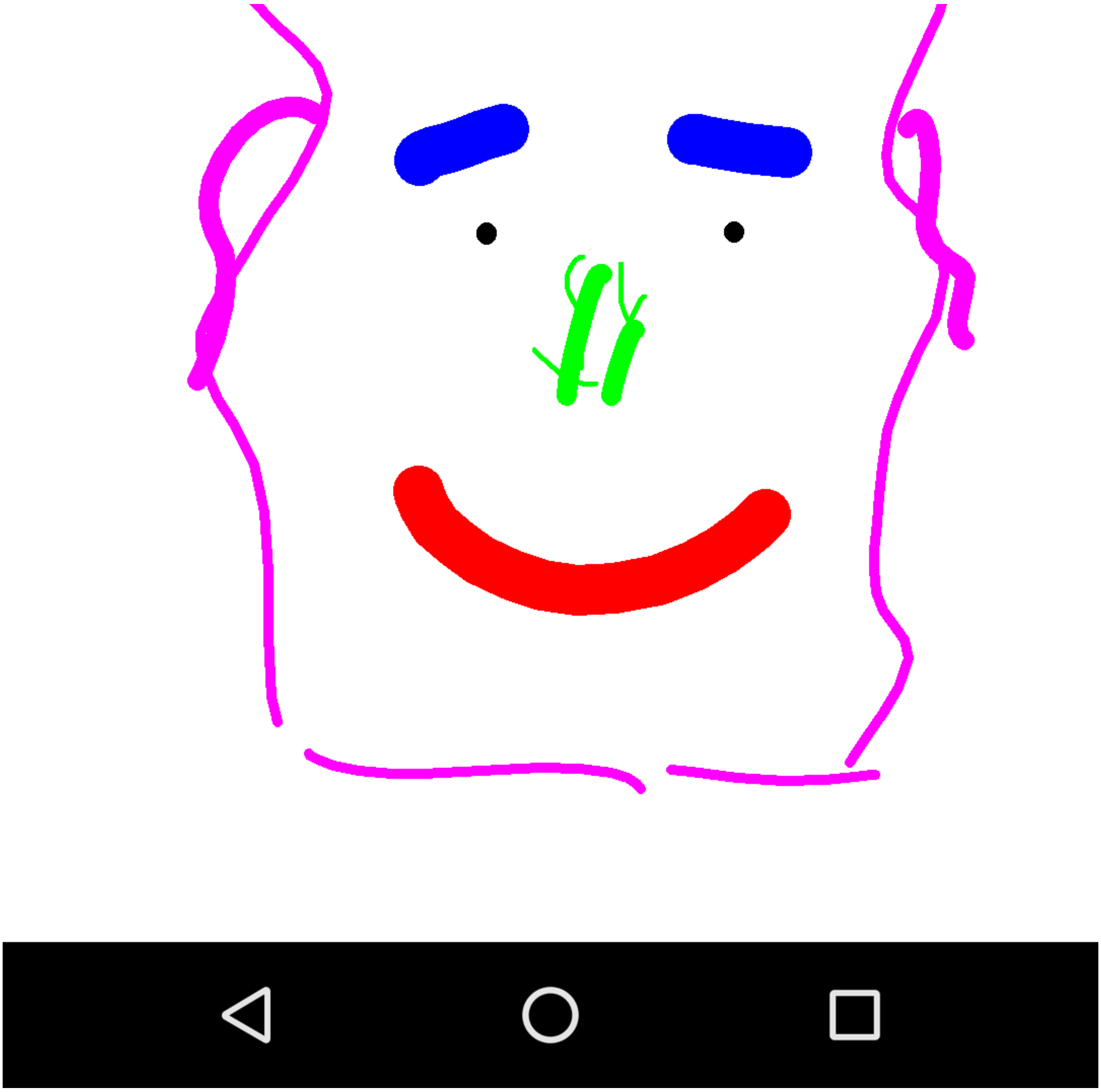
Now for the tricky part.

Between the down and up events, generally there are many `MotionEventActions.Move` events. These are bundled in a single call to `OnTouchEvent`, and they must be handled differently from the `Down` and `Up` events. The `pointerIndex` value obtained earlier from the `ActionIndex` property must be ignored. Instead, the method must obtain multiple `pointerIndex` values by looping between 0 and the `PointerCount` property, and then obtain an `id` for each of those `pointerIndex` values:

```
public override bool OnTouchEvent(MotionEvent args)
{
    // ...
    switch (args.ActionMasked)
    {
        // ...
        case MotionEventActions.Move:

            // Multiple Move events are bundled, so handle them differently
            for (pointerIndex = 0; pointerIndex < args.PointerCount; pointerIndex++)
            {
                id = args.GetPointerId(pointerIndex);

                inProgressPolylines[id].Path.LineTo(args.GetX(pointerIndex),
                                                    args.GetY(pointerIndex));
            }
            break;
        // ...
    }
    // ...
}
```

This type of processing allows the FingerPaint program to track individual fingers and draw the results on the screen:





You've now seen how you can track individual fingers on the screen and distinguish among them.

Related Links

- Equivalent Xamarin iOS guide
- FingerPaint (sample)

HttpClient Stack and SSL/TLS Implementation Selector for Android

7/8/2021 • 4 minutes to read • Edit Online

The HttpClient Stack and SSL/TLS Implementation selectors determine the HttpClient and SSL/TLS implementation that will be used by your Xamarin.Android apps.

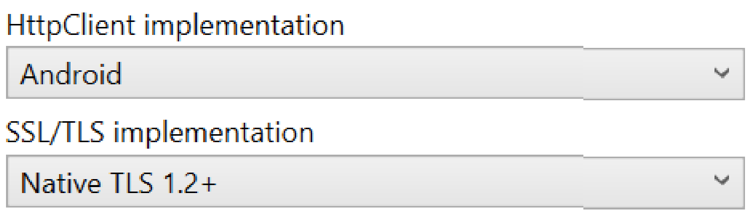
Projects must reference the **System.Net.Http** assembly.

> **WARNING**
>
> **April, 2018** – Due to increased security requirements, including PCI compliance, major cloud providers and web servers are expected to stop supporting TLS versions older than 1.2. Xamarin projects created in previous versions of Visual Studio default to use older versions of TLS.
>
> In order to ensure your apps continue to work with these servers and services, **you should update your Xamarin projects with the `Android HttpClient` and `Native TLS 1.2` settings shown below, then re-build and re-deploy your apps** to your users.

- Visual Studio
- Visual Studio for Mac

The Xamarin.Android HttpClient configuration is in **Project Options > Android Options**, then click the **Advanced Options** button.

These are the recommended settings for TLS 1.2 support:



Alternative configuration options

AndroidClientHandler

AndroidClientHandler is the new handler that delegates to native Java/OS code instead of implementing everything in managed code. **This is the recommended option.**

Pros

- Use native API for better performance and smaller executable size.
- Support for the latest standards, eg. TLS 1.2.

Cons

- Requires Android 4.1 or later.
- Some HttpClient features/options are not available.

Managed (HttpClientHandler)

Managed handler is the fully managed HttpClient handler that has been shipped with previous Xamarin.Android versions.

Pros

- It is the most compatible (features) with MS .NET and older Xamarin versions.

Cons

- It is not fully integrated with the OS (eg. limited to TLS 1.0).
- It is usually much slower (eg. encryption) than native API.
- It requires more managed code, creating larger applications.

Choosing a Handler

The choice between `AndroidClientHandler` and `HttpClientHandler` depends upon the needs of your application. `AndroidClientHandler` is recommended for the most up-to-date security support, eg.

- You require TLS 1.2+ support.
- Your app is targeting Android 4.1 (API 16) or later.
- You need TLS 1.2+ support for `HttpClient`.
- You don't need TLS 1.2+ support for `WebClient`.

`HttpClientHandler` is a good choice if you need TLS 1.2+ support but must support versions of Android earlier than Android 4.1. It is also a good choice if you need TLS 1.2+ support for `WebClient`.

Beginning with Xamarin.Android 8.3, `HttpClientHandler` defaults to Boring SSL (`btls`) as the underlying TLS provider. The Boring SSL TLS provider offers the following advantages:

- It supports TLS 1.2+.
- It supports all Android versions.
- It provides TLS 1.2+ support for both `HttpClient` and `WebClient`.

The disadvantage of using Boring SSL as the underling TLS provider is that it can increase the size of the resulting APK (it adds about 1MB of additional APK size per supported ABI).

Beginning with Xamarin.Android 8.3, the default TLS provider is Boring SSL (`btls`). If you do not want to use Boring SSL, you can revert to the historical managed SSL implementation by setting the `$(AndroidTlsProvider)` property to `legacy` (for more information about setting build properties, see Build Process).

Programatically Using `AndroidClientHandler`

The `Xamarin.Android.Net.AndroidClientHandler` is an `HttpMessageHandler` implementation specifically for Xamarin.Android. Instances of this class will use the native `java.net.URLConnection` implementation for all HTTP connections. This will theoretically provide an increase in HTTP performance and smaller APK sizes.

This code snippet is an example of how to explicitly for a single instance of the `HttpClient` class:

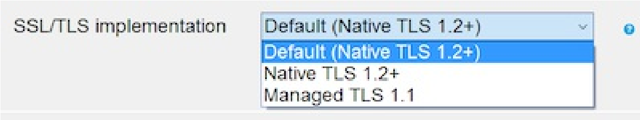
```
// Android 4.1 or higher, Xamarin.Android 6.1 or higher
HttpClient client = new HttpClient(new Xamarin.Android.Net.AndroidClientHandler ());
```

> **NOTE**
>
> The underlying Android device must support TLS 1.2 (ie. Android 4.1 and later). Please note that the official support for TLS 1.2 is in Android 5.0+. However some devices support TLS 1.2 in Android 4.1+.

SSL/TLS implementation build option

This project option controls what underlying TLS library will be used by all web request, both `HttpClient` and `WebRequest`. By default, TLS 1.2 is selected:

- Visual Studio
- Visual Studio for Mac



For example:

```
var client = new HttpClient();
```

If the HttpClient implementation was set to **Managed** and the TLS implementation was set to **Native TLS 1.2+**, then the `client` object would automatically use the managed `HttpClientHandler` and TLS 1.2 (provided by the BoringSSL library) for its HTTP requests.

However, if the **HttpClient implementation** is set to `AndroidHttpClient`, then all `HttpClient` objects will use the underlying Java class `java.net.URLConnection` and will be unaffected by the **TLS/SSL implementation** value. `WebRequest` objects would use the BoringSSL library.

Other ways to control SSL/TLS configuration

There are three ways that a Xamarin.Android application can control the TLS settings:

1. Select the HttpClient implementation and default TLS library in the Project Options.
2. Programatically using `Xamarin.Android.Net.AndroidClientHandler`.
3. Declare environment variables (optional).

Of the three choices, the recommended approach is to use the Xamarin.Android project options to declare the default `HttpMessageHandler` and TLS for the entire app. Then, if necessary, programmatically instantiate `Xamarin.Android.Net.AndroidClientHandler` objects. These options are described above.

The third option – using environment variables – is explained below.

Declare Environment Variables

There are two environment variables that are related to the use of TLS in Xamarin.Android:

- `XA_HTTP_CLIENT_HANDLER_TYPE` – This environment variable declares the default `HttpMessageHandler` that the application will use. For example:

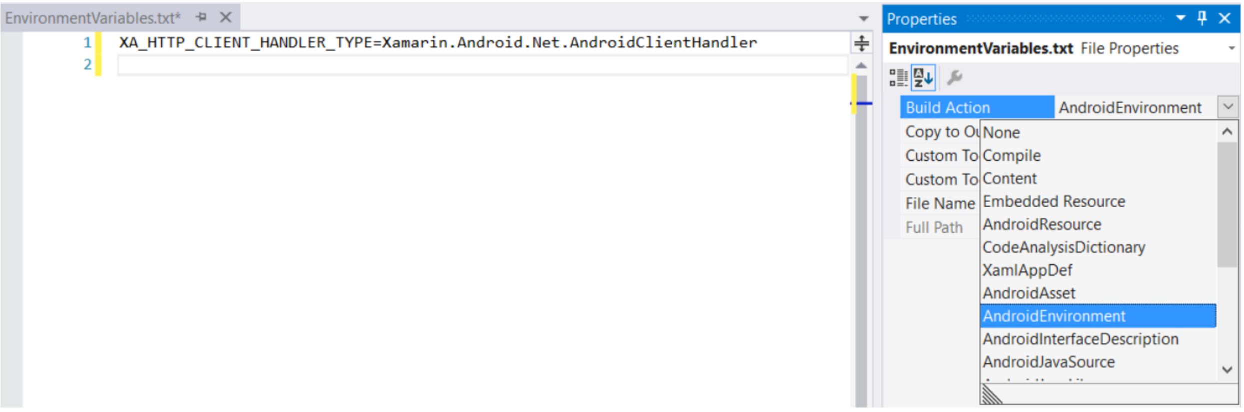
  ```
  XA_HTTP_CLIENT_HANDLER_TYPE=Xamarin.Android.Net.AndroidClientHandler
  ```

- `XA_TLS_PROVIDER` – This environment variable will declare which TLS library will be used, either `btls`, `legacy`, or `default` (which is the same as omitting this variable):

  ```
  XA_TLS_PROVIDER=btls
  ```

This environment variable is set by adding an *environment file* to the project. An environment file is a Unix-formatted plain-text file with a build action of **AndroidEnvironment**:

- Visual Studio
- Visual Studio for Mac



Please see the Xamarin.Android Environment guide for more details about environment variables and Xamarin.Android.

Related Links

- Transport Layer Security (TLS)

Writing Responsive Applications

10/28/2019 • 2 minutes to read • Edit Online

One of the keys to maintaining a responsive GUI is to do long-running tasks on a background thread so the GUI doesn't get blocked. Let's say we want to calculate a value to display to the user, but that value takes 5 seconds to calculate:

```
public class ThreadDemo : Activity
{
    TextView textview;

    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);

        // Create a new TextView and set it as our view
        textview = new TextView (this);
        textview.Text = "Working..";

        SetContentView (textview);

        SlowMethod ();
    }

    private void SlowMethod ()
    {
        Thread.Sleep (5000);
        textview.Text = "Method Complete";
    }
}
```

This will work, but the application will "hang" for 5 seconds while the value is calculated. During this time, the app will not respond to any user interaction. To get around this, we want to do our calculations on a background thread:

```
public class ThreadDemo : Activity
{
    TextView textview;

    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);

        // Create a new TextView and set it as our view
        textview = new TextView (this);
        textview.Text = "Working..";

        SetContentView (textview);

        ThreadPool.QueueUserWorkItem (o => SlowMethod ());
    }

    private void SlowMethod ()
    {
        Thread.Sleep (5000);
        textview.Text = "Method Complete";
    }
}
```

Now we calculate the value on a background thread so our GUI stays responsive during the calculation. However, when the calculation is done, our app crashes, leaving this in the log:

```
E/mono    (11207): EXCEPTION handling: Android.Util.AndroidRuntimeException: Exception of type
'Android.Util.AndroidRuntimeException' was thrown.
E/mono    (11207):
E/mono    (11207): Unhandled Exception: Android.Util.AndroidRuntimeException: Exception of type
'Android.Util.AndroidRuntimeException' was thrown.
E/mono    (11207):   at Android.Runtime.JNIEnv.CallVoidMethod (IntPtr jobject, IntPtr jmethod,
Android.Runtime.JValue[] parms)
E/mono    (11207):   at Android.Widget.TextView.set_Text (IEnumerable`1 value)
E/mono    (11207):   at MonoDroidDebugging.Activity1.SlowMethod ()
```

This is because you must update the GUI from the GUI thread. Our code updates the GUI from the ThreadPool thread, causing the app to crash. We need to calculate our value on the background thread, but then do our update on the GUI thread, which is handled with Activity.RunOnUIThread:

```
public class ThreadDemo : Activity
{
    TextView textview;

    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);

        // Create a new TextView and set it as our view
        textview = new TextView (this);
        textview.Text = "Working..";

        SetContentView (textview);

        ThreadPool.QueueUserWorkItem (o => SlowMethod ());
    }

    private void SlowMethod ()
    {
        Thread.Sleep (5000);
        RunOnUiThread (() => textview.Text = "Method Complete");
    }
}
```

This code works as expected. This GUI stays responsive and gets properly updated once the calculation is comple.

Note this technique isn't just used for calculating an expensive value. It can be used for any long-running task that can be done in the background, like a web service call or downloading internet data.

User Interfaces with Xamarin.Android

11/2/2020 • 2 minutes to read • Edit Online

The following sections explain the various tools and building blocks that are used to compose user interfaces in Xamarin.Android apps.

Android Designer

This section explains how to use the Android Designer to lay out controls visually and edit properties. It also explains how to use the Designer to work with user interfaces and resources across various configurations, such as themes, languages, and device configurations, as well as how to design for alternative views like landscape and portrait.

Material Theme

Material Theme is the user interface style that determines the look and feel of views and activities in Android. Material Theme is built into Android, so it is used by the system UI as well as by applications. This guide introduces Material Design principles and explains how to theme an app using either built-in Material Themes or a custom theme.

User Profile

This guide explains how to access the personal profile for the owner of a device, including contact data such as the device owner's name and phone number.

Splash Screen

An Android app takes some time to start up, especially when the app is first launched on a device. A splash screen may display start up progress to the user. This guide explains how to create a splash screen for your app.

Layouts

Layouts are used to define the visual structure for a user interface. Layouts such as `ListView` and `RecyclerView` are the most fundamental building blocks of Android applications. Typically, a layout will use an `Adapter` to act as a bridge from the layout to the underlying data that is used to populate data items in the layout. This section explains how to use layouts such as `LinearLayout`, `RelativeLayout`, `TableLayout`, `RecyclerView`, and `GridView`.

Controls

Android controls (also called *widgets*) are the UI elements that you use to build a user interface. This section explains how to use controls such as buttons, toolbars, date/time pickers, calendars, spinners, switches, pop-up menus, view pagers, and web views.

Xamarin.Android Designer

4/8/2020 • 2 minutes to read • Edit Online

This article describes the features of the Xamarin.Android Designer. It explains designer basics, demonstrating how to use the Designer to lay out widgets visually and edit properties. It also illustrates how to use the Designer to work with user interfaces and resources across various configurations, such as themes, languages, and device configurations, as well as how to design for alternative views such as landscape and portrait.

Overview

Xamarin.Android supports both a declarative style of user interface design based in XML files, as well as programmatic user interface creation in code. When using the declarative approach, XML files can be either hand-edited or modified visually by using the Xamarin.Android Designer. Use of a designer allows immediate feedback during UI creation, speeds up development, and makes the process of UI creation less laborious.

This article surveys the many features of the Xamarin.Android Designer. It explains the following:

1. The basics of using the Designer.
2. The various parts that make up the Designer.
3. How to load an Android layout into the Designer.
4. How to add widgets.
5. How to edit properties.
6. How to work with various resources and device configurations.
7. How to modify a user interface for alternative views such as landscape and portrait.
8. How to handle conflicts that may arise when working with alternative views.
9. How to use Material Design tools to build Material Design-compliant apps.

Sections

Using the Android Designer

Designer Basics

Resource Qualifiers and Visualization Options

Alternative Layout Views

Material Design Features

Android Layout Diagnostics

Android Designer Diagnostic Analyzers

Summary

This article discussed the feature set of the Xamarin.Android Designer. It showed how to get started with the Designer, and explained its various parts. It described how to load a layout, as well as how to add and modify widgets, by using both the **Designer Surface** as well as the **Source** view. It also explained how to work with various resources and device configurations. Finally, it examined how to use the Designer to develop user interfaces that are built specifically for alternative views, such as landscape and portrait, as well as how to resolve conflicts that may arise between such views.

Related links

- Designer walkthrough
- Android resources

Using the Xamarin.Android Designer

7/8/2021 • 14 minutes to read • Edit Online

This article is a walkthrough of the Xamarin.Android Designer. It demonstrates how to create a user interface for a small color browser app; this user interface is created entirely in the Designer.

Overview

Android user interfaces can be created declaratively by using XML files or programmatically by writing code. The Xamarin.Android Designer allows developers to create and modify declarative layouts visually, without requiring hand-editing of XML files. The Designer also provides real-time feedback that lets the developer evaluate UI changes without having to redeploy the application to a device or to an emulator. These Designer features can speed up Android UI development tremendously. This article demonstrates how to use the Xamarin.Android Designer to visually create a user interface.

> **TIP**
>
> Newer releases of Visual Studio support opening .xml files inside the Android Designer.
>
> Both .axml and .xml files are supported in the Android Designer.

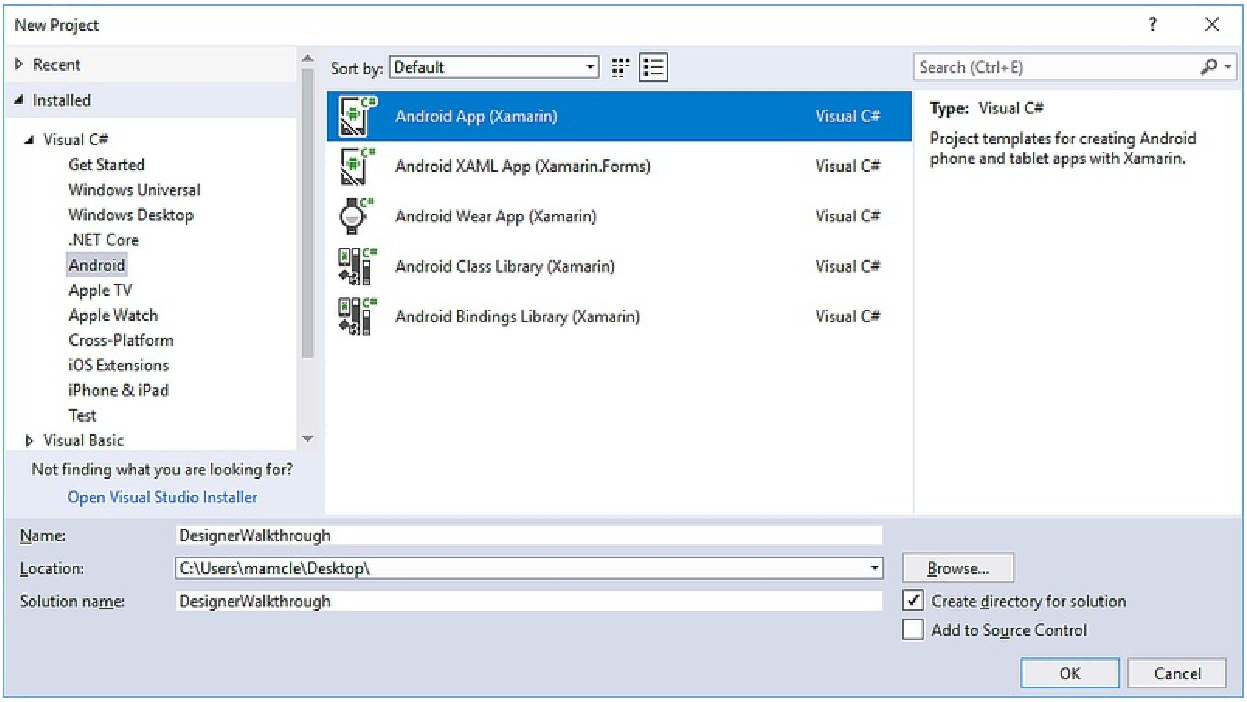
Walkthrough

The objective of this walkthrough is to use the Android Designer to create a user interface for an example color browser app. The color browser app presents a list of colors, their names, and their RGB values. You'll learn how to add widgets to the **Design Surface** as well as how to lay out these widgets visually. After that, you'll learn how to modify widgets interactively on the **Design Surface** or by using the Designer's **Properties** pane. Finally, you'll see how the design looks when the app runs on a device or emulator.

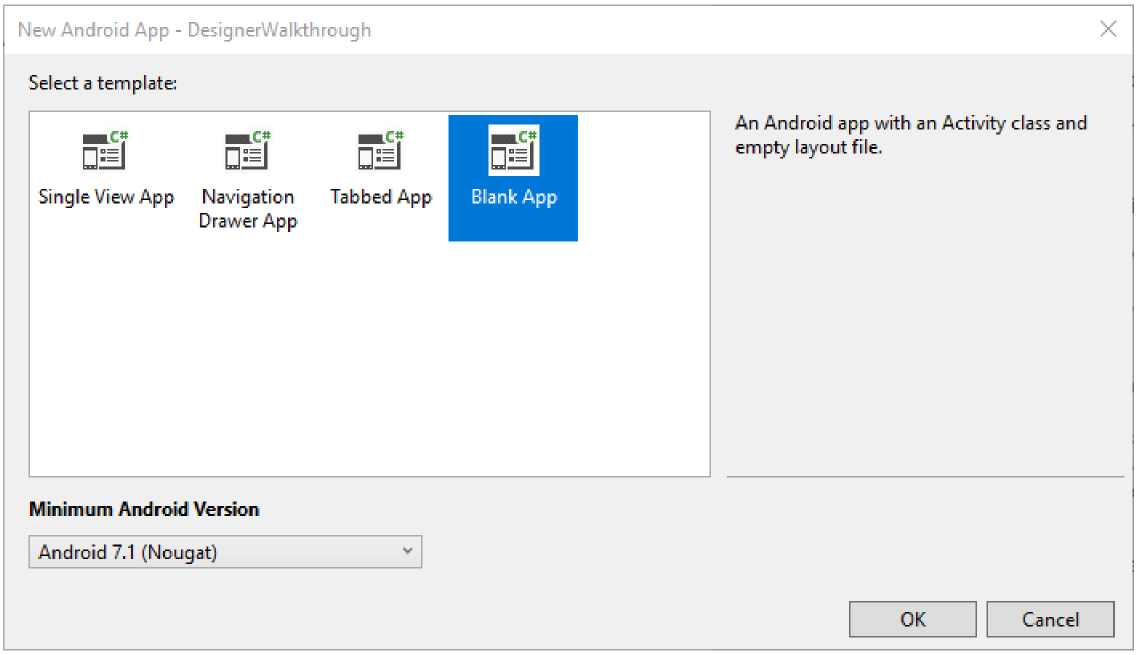
- Visual Studio
- Visual Studio for Mac

Creating a new project

The first step is to create a new Xamarin.Android project. Launch Visual Studio, click **New Project...**, and choose the **Visual C# > Android > Android App (Xamarin)** template. Name the new app **DesignerWalkthrough** and click **OK**.

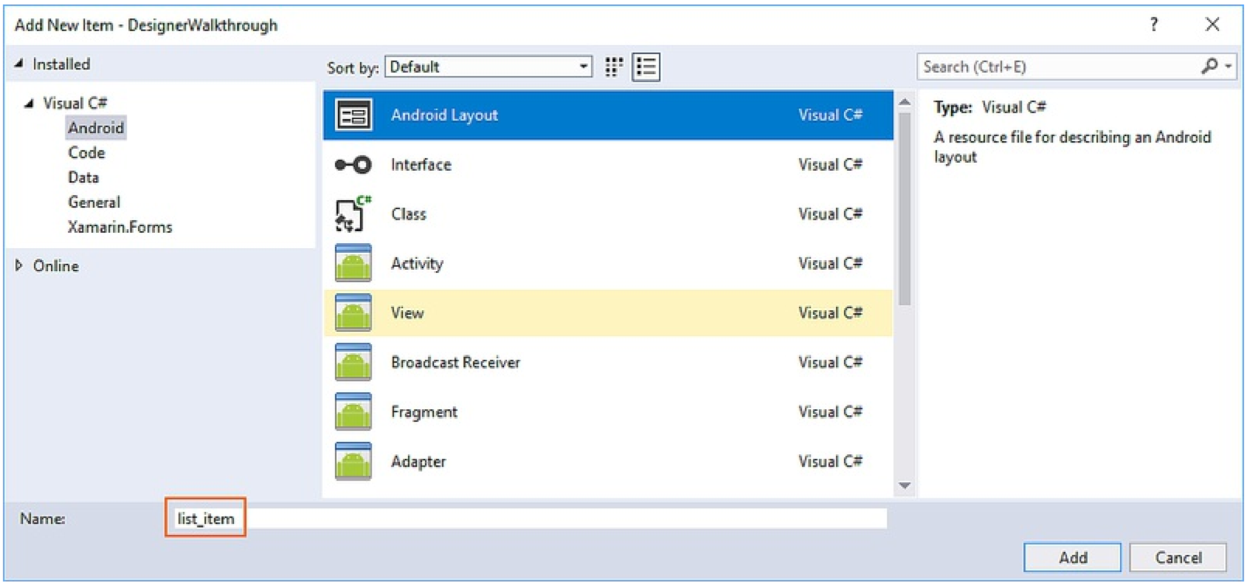


In the **New Android App** dialog, choose **Blank App** and click **OK**:

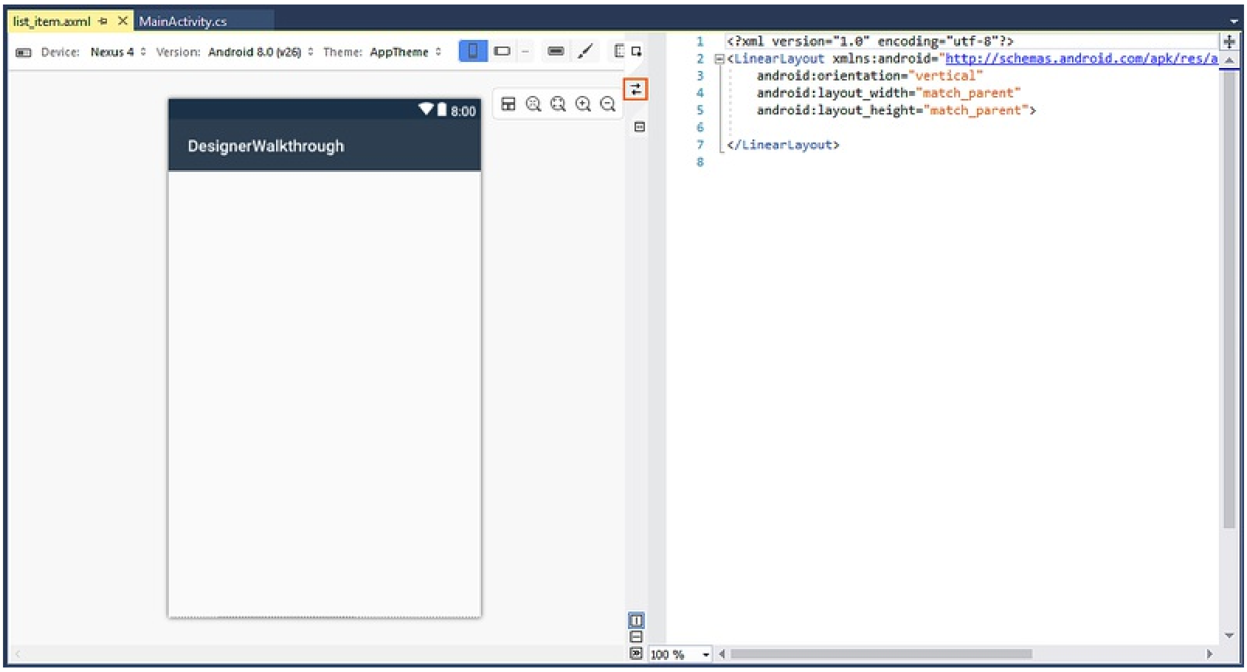


Adding a layout

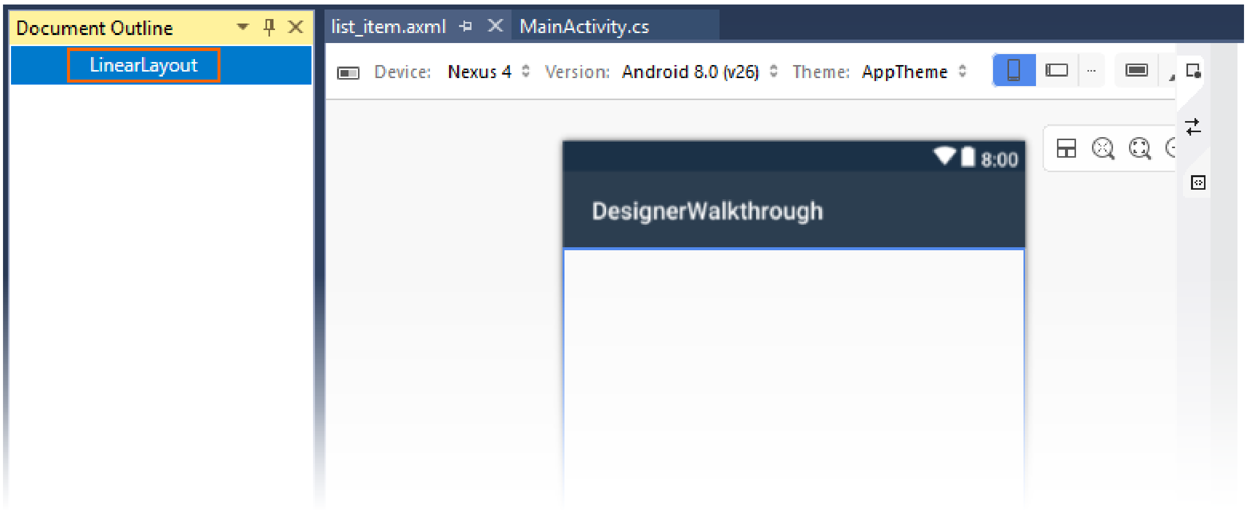
The next step is to create a **LinearLayout** that will hold the user interface elements. Right-click **Resources/layout** in the **Solution Explorer** and select **Add > New Item...**. In the **Add New Item** dialog, select **Android Layout**. Name the file **list_item** and click **Add**:



The new **list_item** layout is displayed in the Designer. Notice that two panes are displayed – the *Design Surface* for the **list_item** is visible in the left pane while its XML source is shown on the right pane. You can swap the positions of the **Design Surface** and **Source** panes by clicking the **Swap Panes** icon located between the two panes:



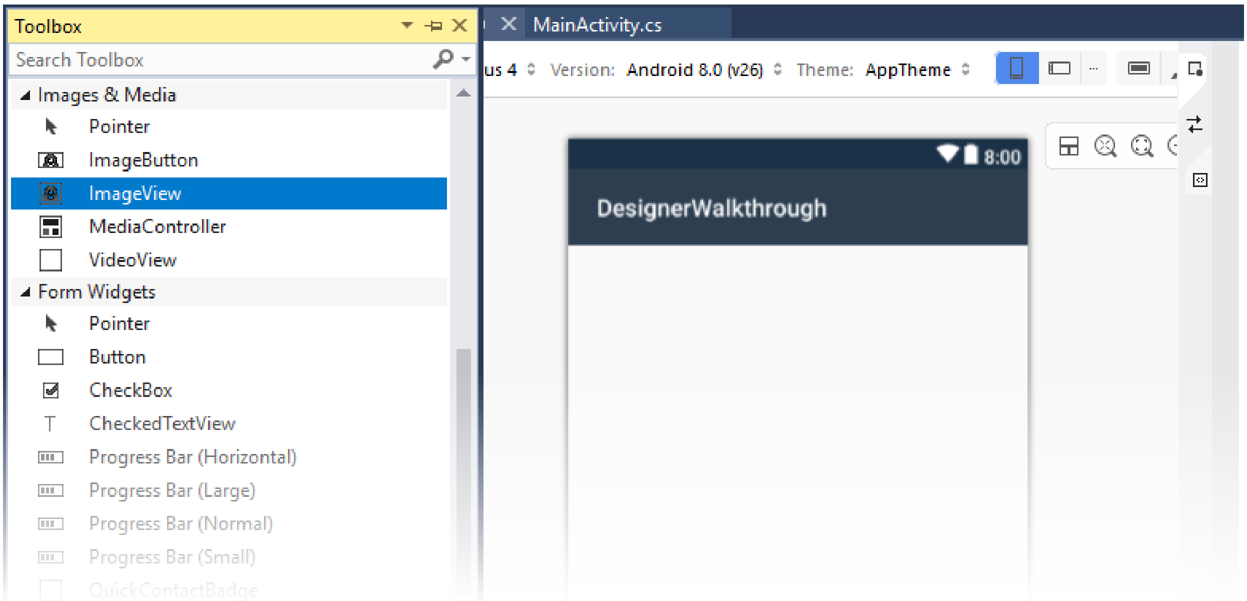
From the **View** menu, click **Other Windows > Document Outline** to open the **Document Outline**. The **Document Outline** shows that the layout currently contains a single **LinearLayout** widget:



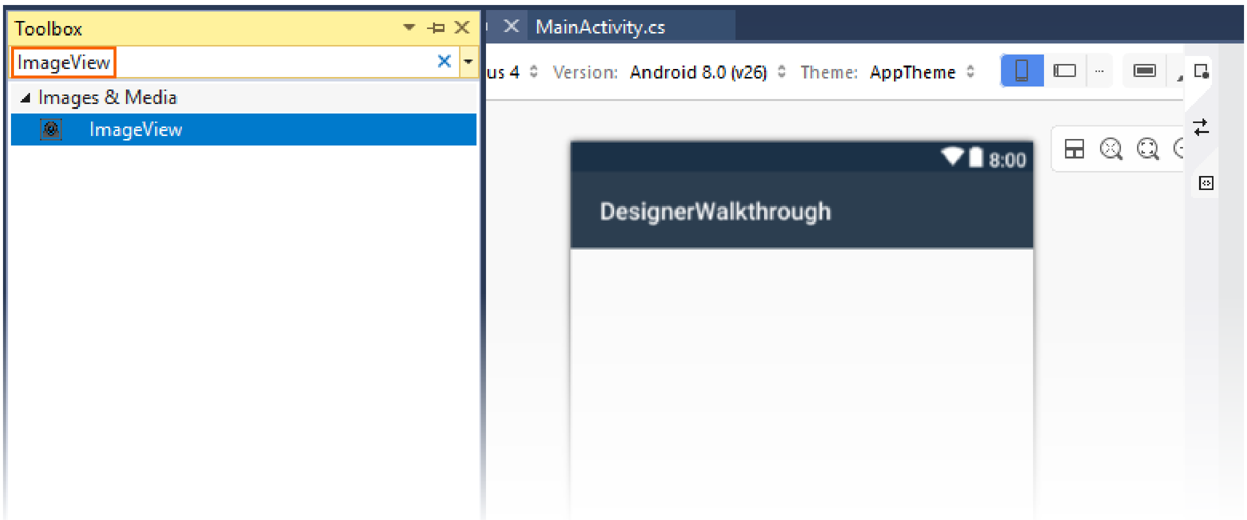
The next step is to create the user interface for the color browser app within this `LinearLayout`.

Creating the List Item user interface

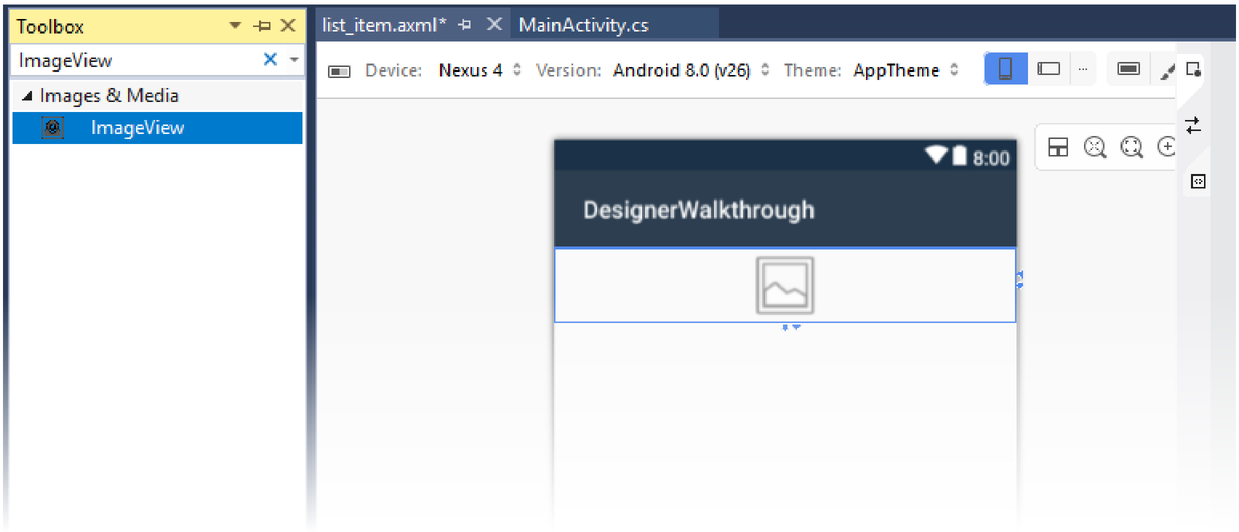
If the **Toolbox** pane is not showing, click the **Toolbox** tab on the left. In the **Toolbox**, scroll down to the **Images & Media** section and scroll down further until you locate an `ImageView`:



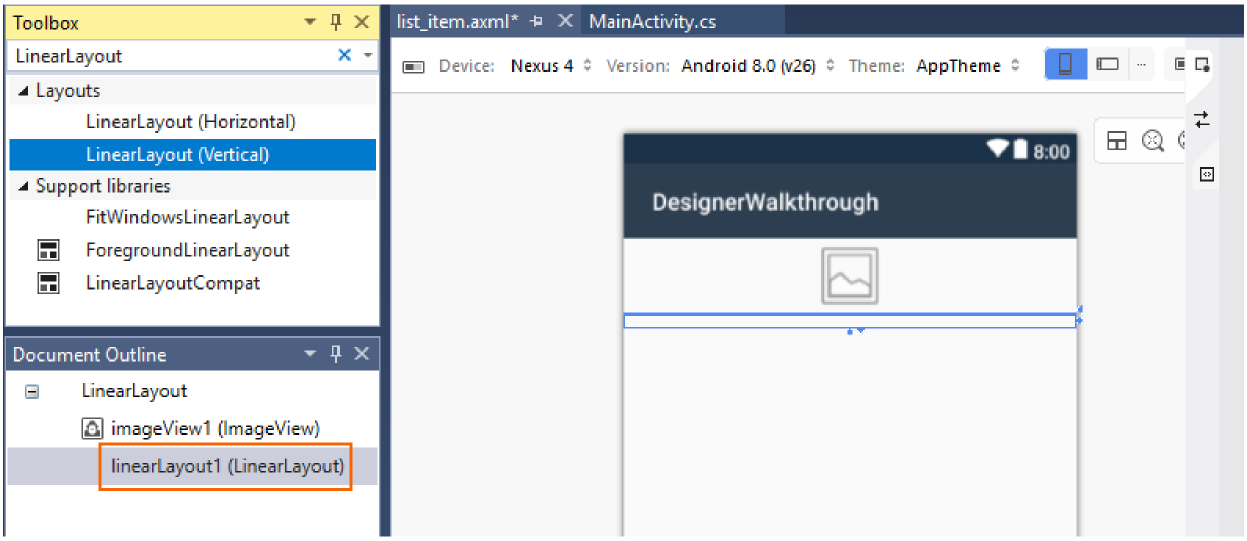
Alternately, you can enter *ImageView* into the search bar to locate the `ImageView`:



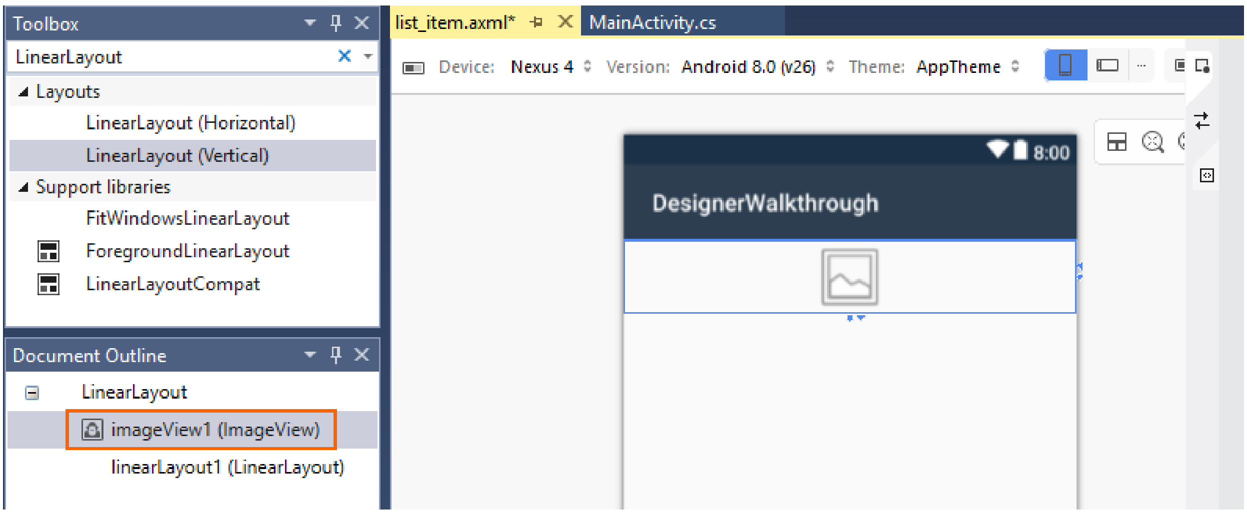
Drag this `ImageView` onto the Design Surface (this `ImageView` will be used to display a color swatch in the color browser app):



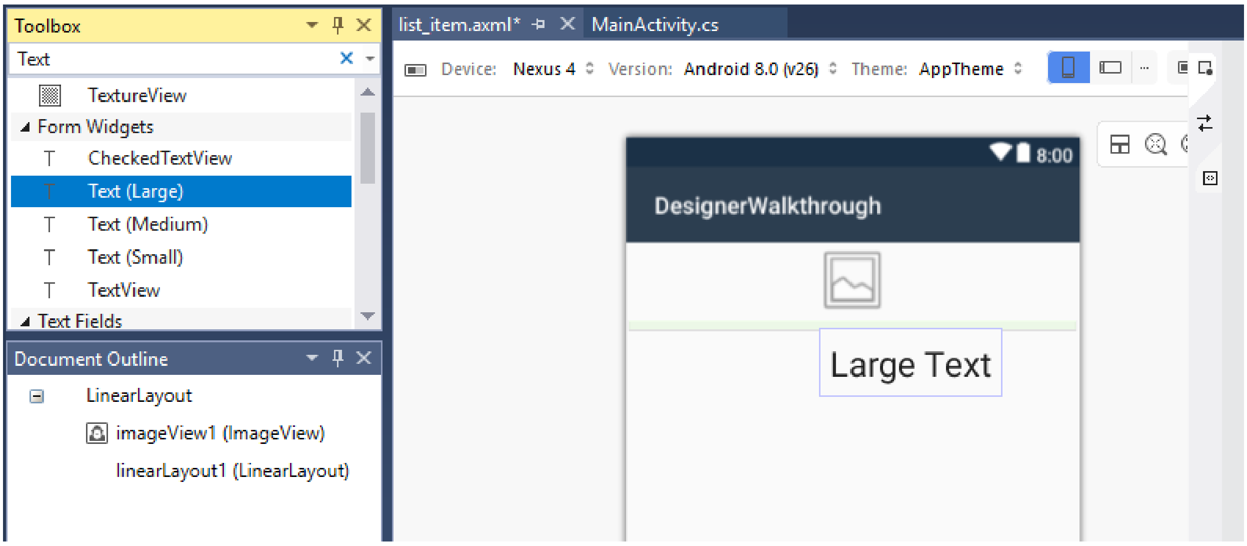
Next, drag a `LinearLayout (Vertical)` widget from the **Toolbox** into the Designer. Notice that a blue outline indicates the boundaries of the added `LinearLayout`. The **Document Outline** shows that it is a child of `LinearLayout`, located under `imageView1 (ImageView)`:



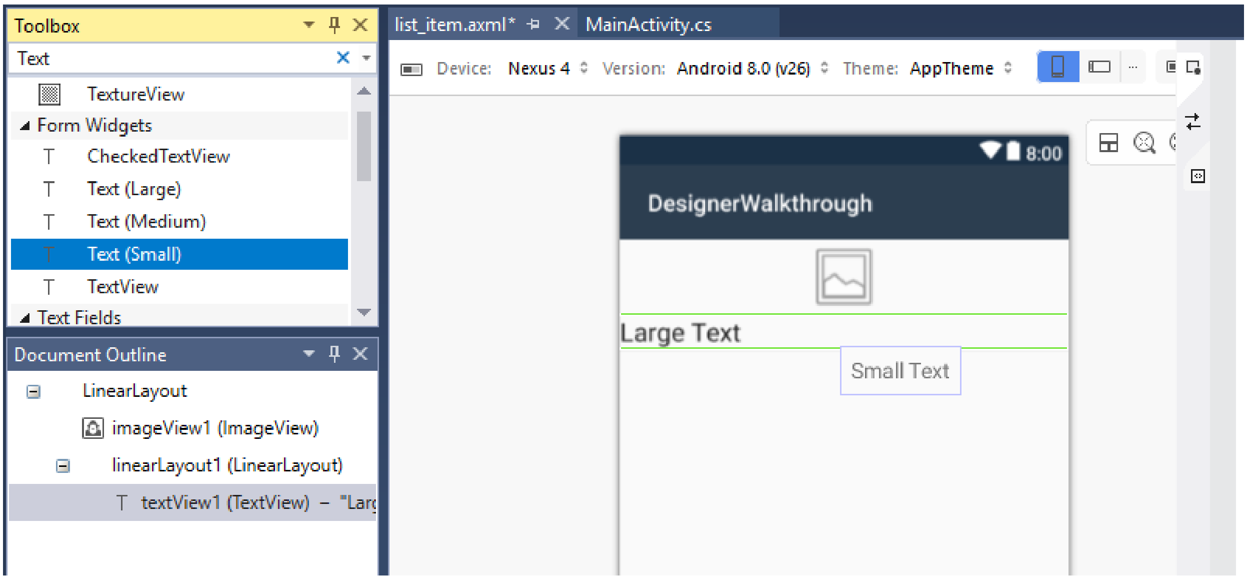
When you select the `ImageView` in the Designer, the blue outline moves to surround the `ImageView`. In addition, the selection moves to `imageView1 (ImageView)` in the **Document Outline**:



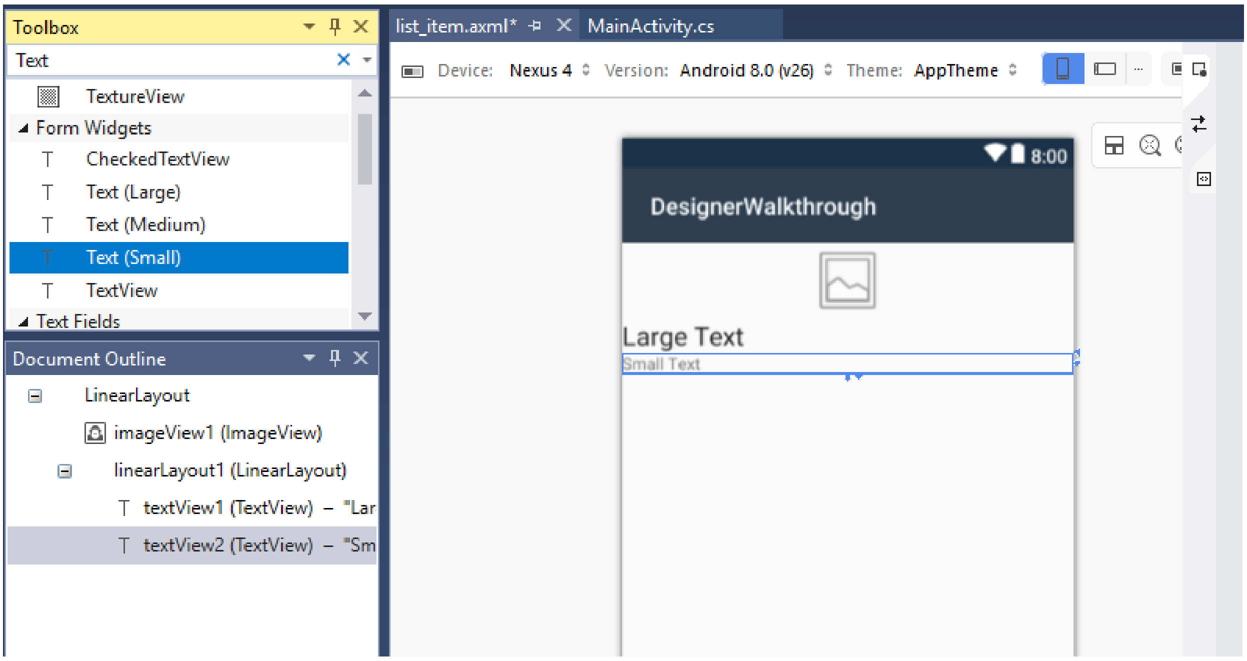
Next, drag a `Text (Large)` widget from the **Toolbox** into the newly-added `LinearLayout`. Notice that the Designer uses green highlights to indicate where the new widget will be inserted:



Next, add a `Text (Small)` widget below the `Text (Large)` widget:



At this point, the Designer surface should resemble the following screenshot:



If the two `textView` widgets are not inside `linearLayout1`, you can drag them to `linearLayout1` in the **Document Outline** and position them so they appear as shown in the previous screenshot (indented under `linearLayout1`).

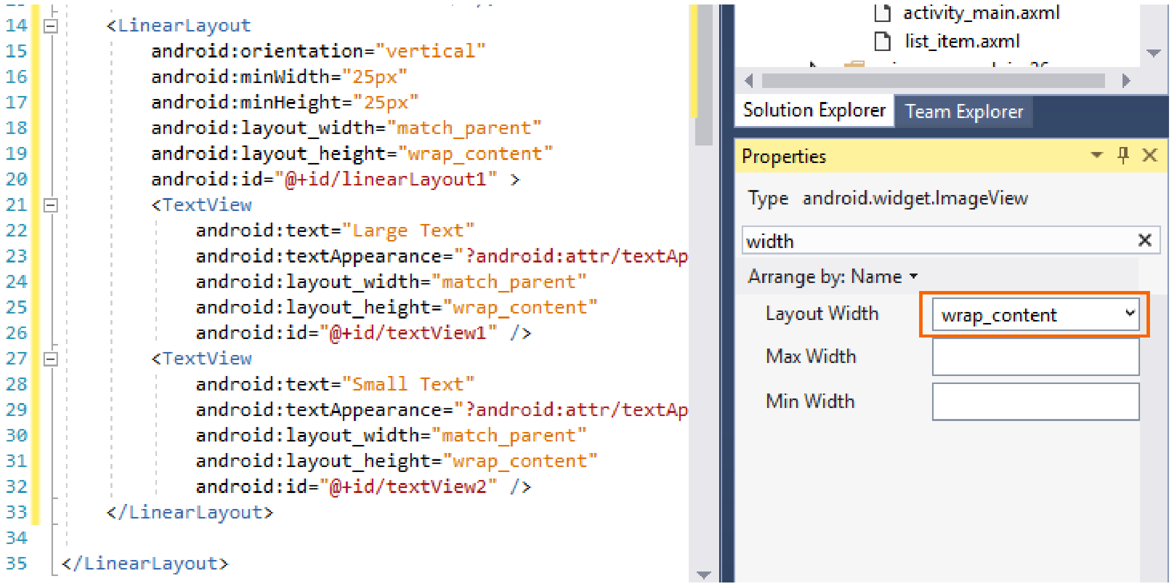
Arranging the user interface

The next step is to modify the UI to display the `ImageView` on the left, with the two `TextView` widgets stacked to the right of the `ImageView` .

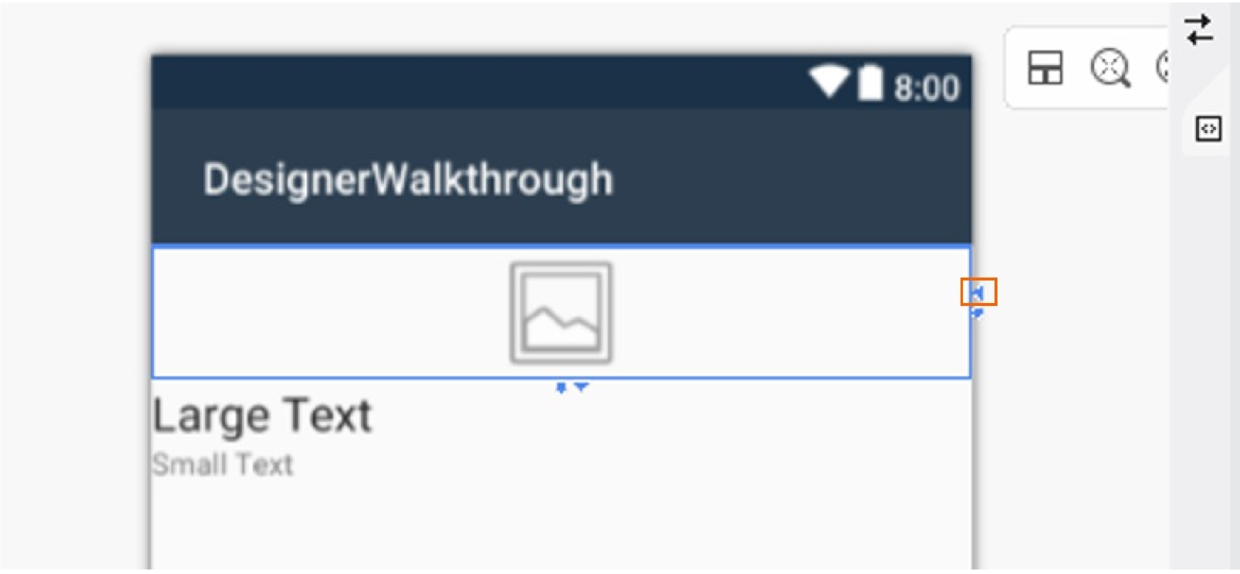
1. Select the `ImageView` .

2. In the **Properties window**, enter *width* in the search box and locate **Layout Width**.

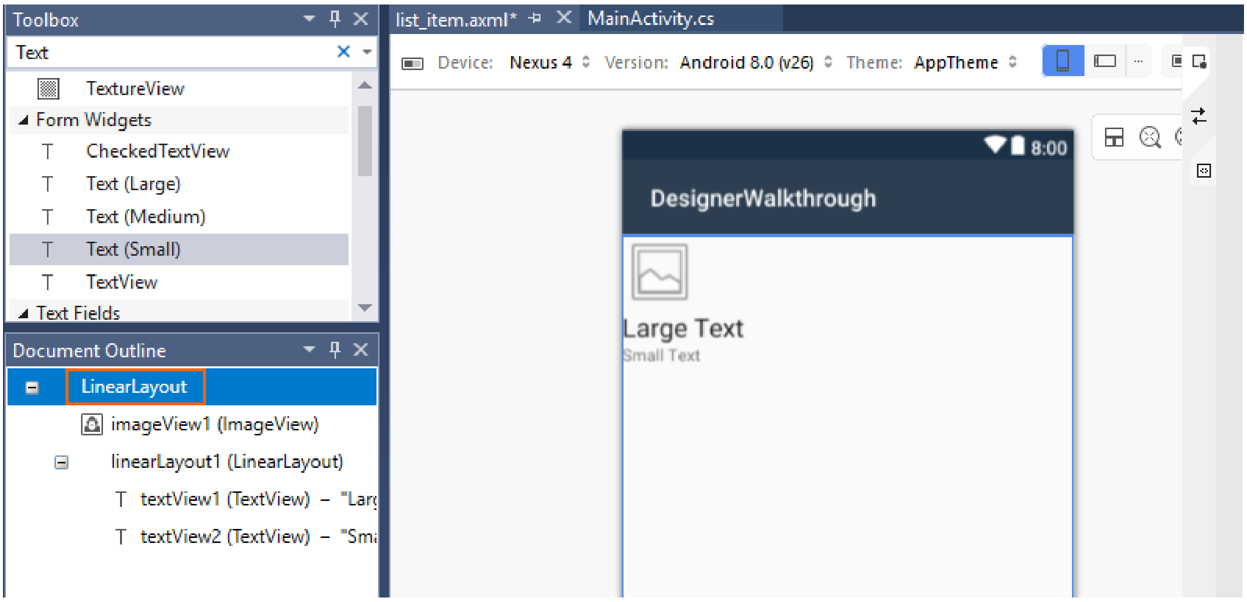
3. Change the **Layout Width** setting to `wrap_content` :



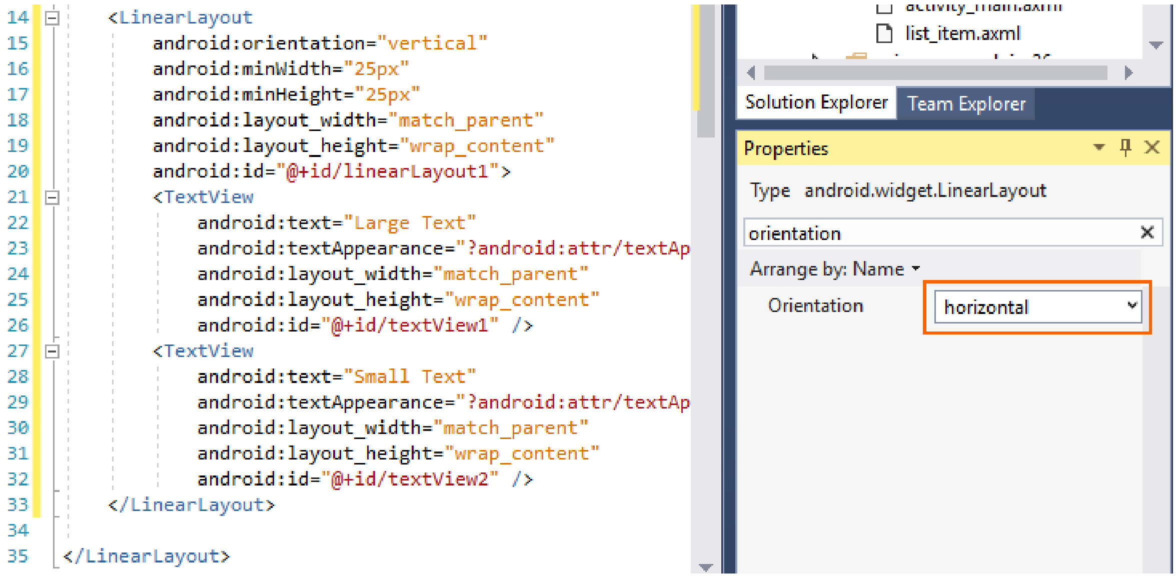
Another way to change the `Width` setting is to click the triangle on the right-hand side of the widget to toggle its width setting to `wrap_content` :



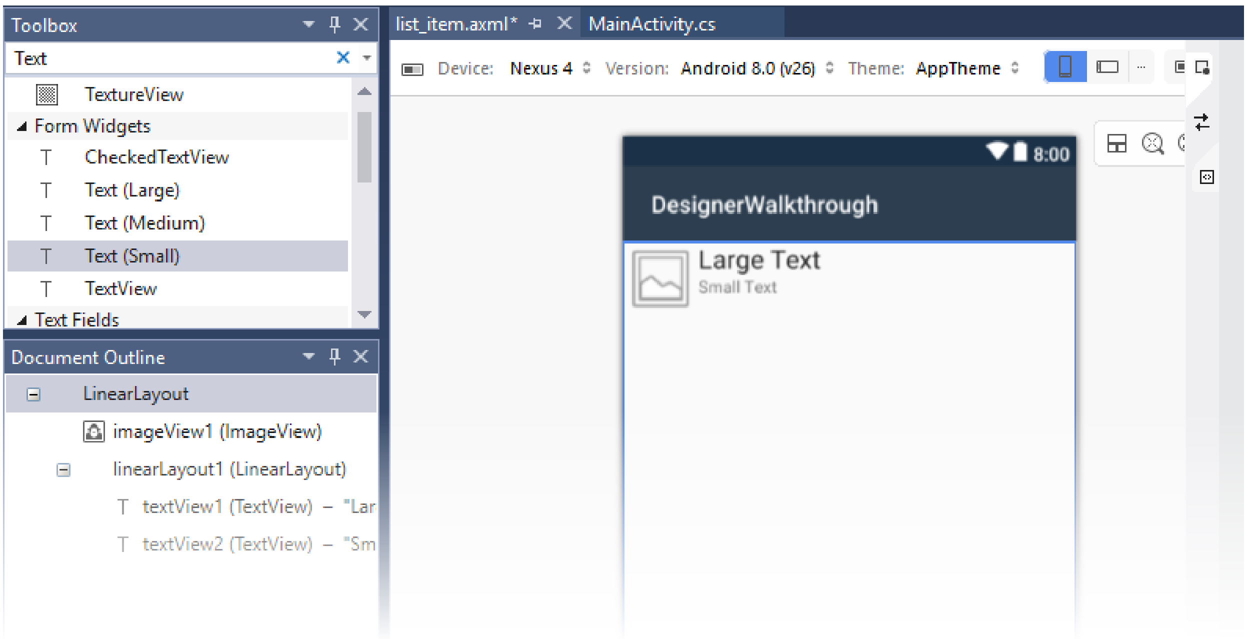
Clicking the triangle again returns the `Width` setting to `match_parent` . Next, go to the **Document Outline** pane and select the root `LinearLayout` :



With the root `LinearLayout` selected, return to the **Properties** pane, enter *orientation* into the search box and locate the **Orientation** setting. Change **Orientation** to `horizontal`:

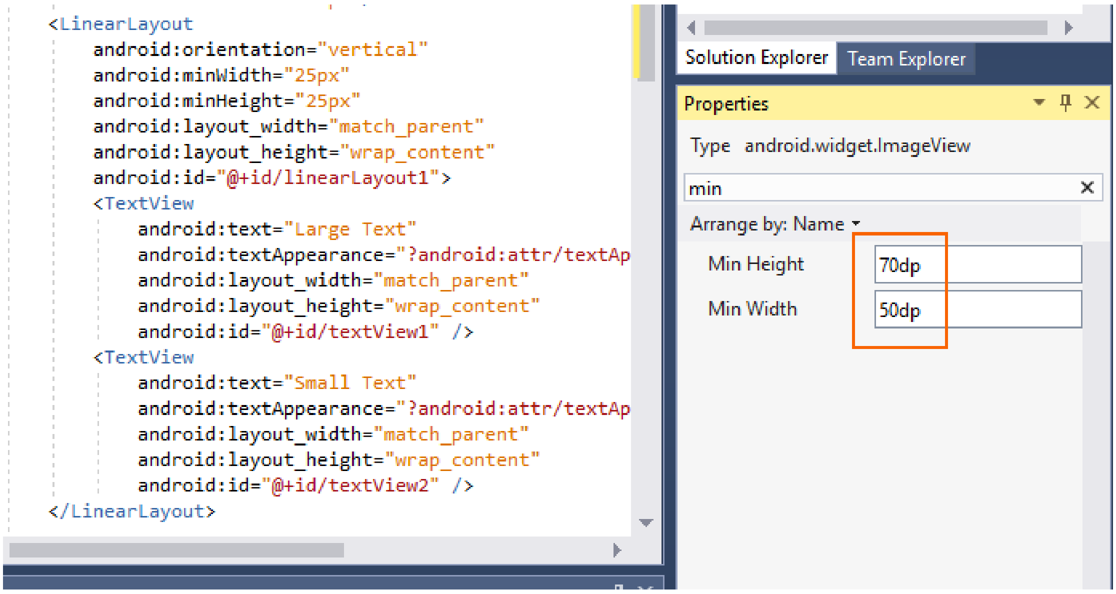


At this point, the Designer surface should resemble the following screenshot. Notice that the `TextView` widgets have been moved to the right of the `ImageView`:

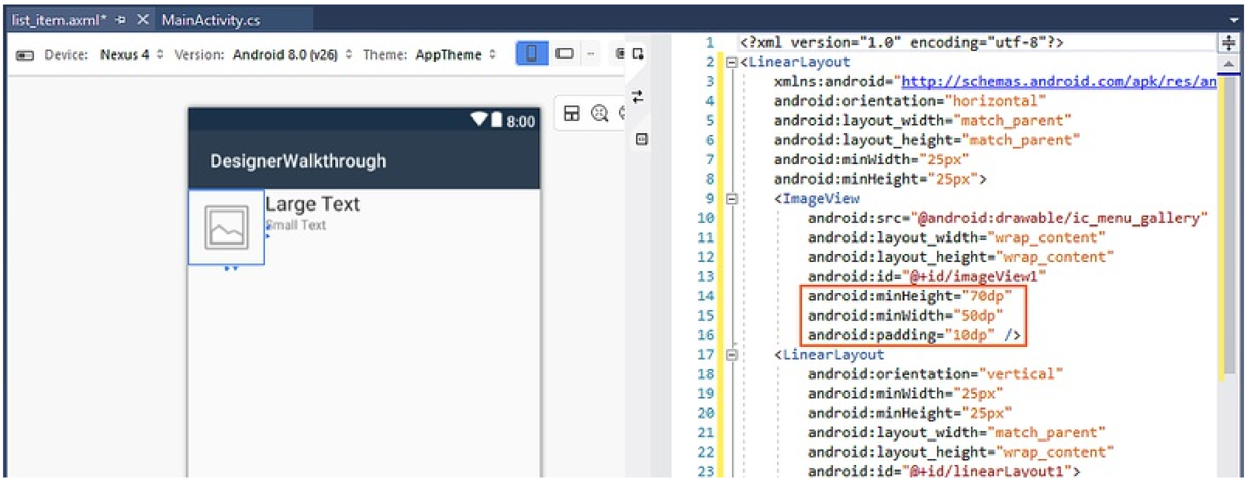


Modifying the spacing

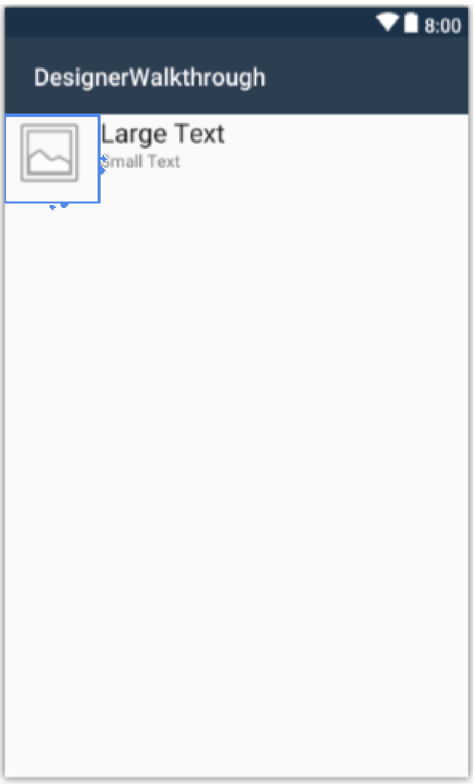
The next step is to modify padding and margin settings in the UI to provide more space between the widgets. Select the `ImageView` on the Design surface. In the **Properties** pane, enter `min` in the search box. Enter `70dp` for **Min Height** and `50dp` for **Min Width**:

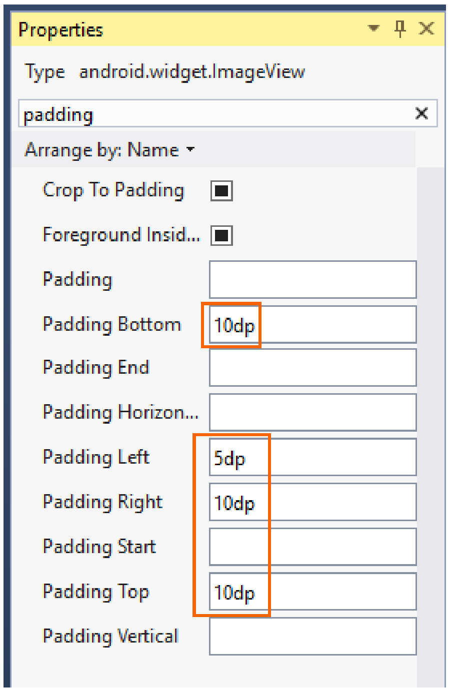


In the **Properties** pane, enter `padding` in the search box and enter `10dp` for **Padding**. These `minHeight`, `minWidth` and `padding` settings add padding around all sides of the `ImageView` and elongate it vertically. Notice that the layout XML changes as you enter these values:

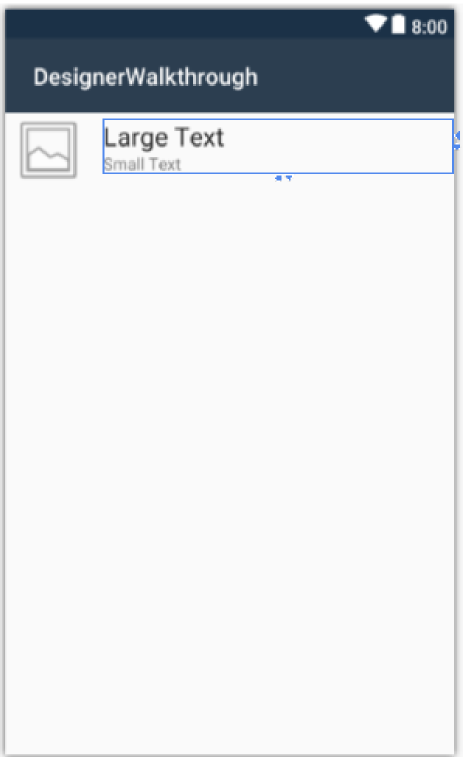


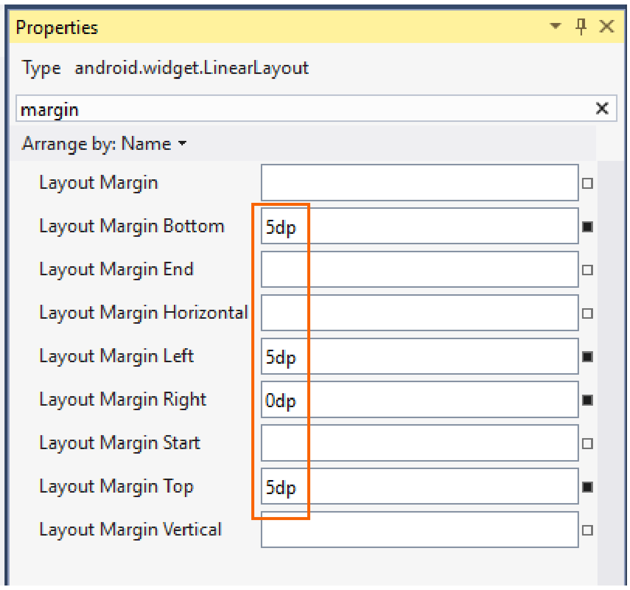
The bottom, left, right, and top padding settings can be set independently by entering values into the **Padding Bottom**, **Padding Left**, **Padding Right**, and **Padding Top** fields, respectively. For example, set the **Padding Left** field to `5dp` and the **Padding Bottom**, **Padding Right**, and **Padding Top** fields to `10dp`:





Next, adjust the position of the `LinearLayout` widget that contains the two `TextView` widgets. In the **Document Outline**, select `linearLayout1`. In the **Properties** window, enter `margin` in the search box. Set **Layout Margin Bottom**, **Layout Margin Left**, and **Layout Margin Top** to `5dp`. Set **Layout Margin Right** to `0dp`:





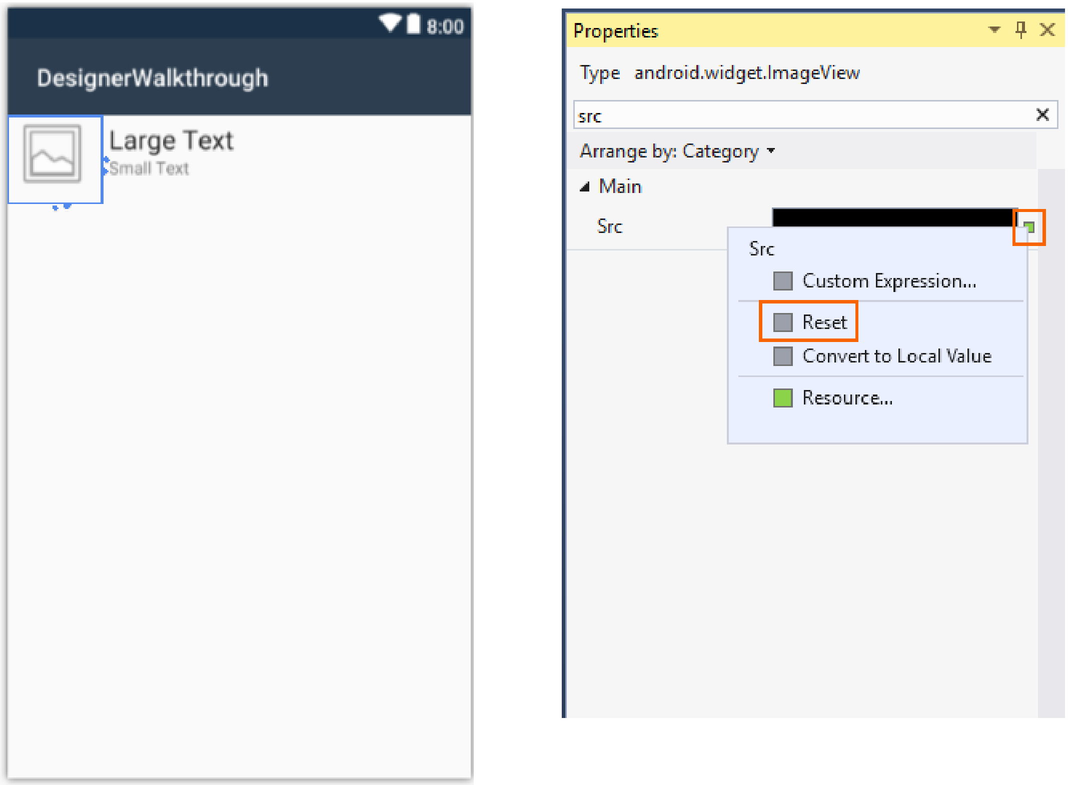
Removing the default image

Because the `ImageView` is being used to display colors (rather than images), the next step is to remove the default image source added by the template.

1. Select the `ImageView` on the **Designer Surface**.

2. In **Properties**, enter *src* in the search box.

3. Click the small square to the right of the **Src** property setting and select **Reset**:

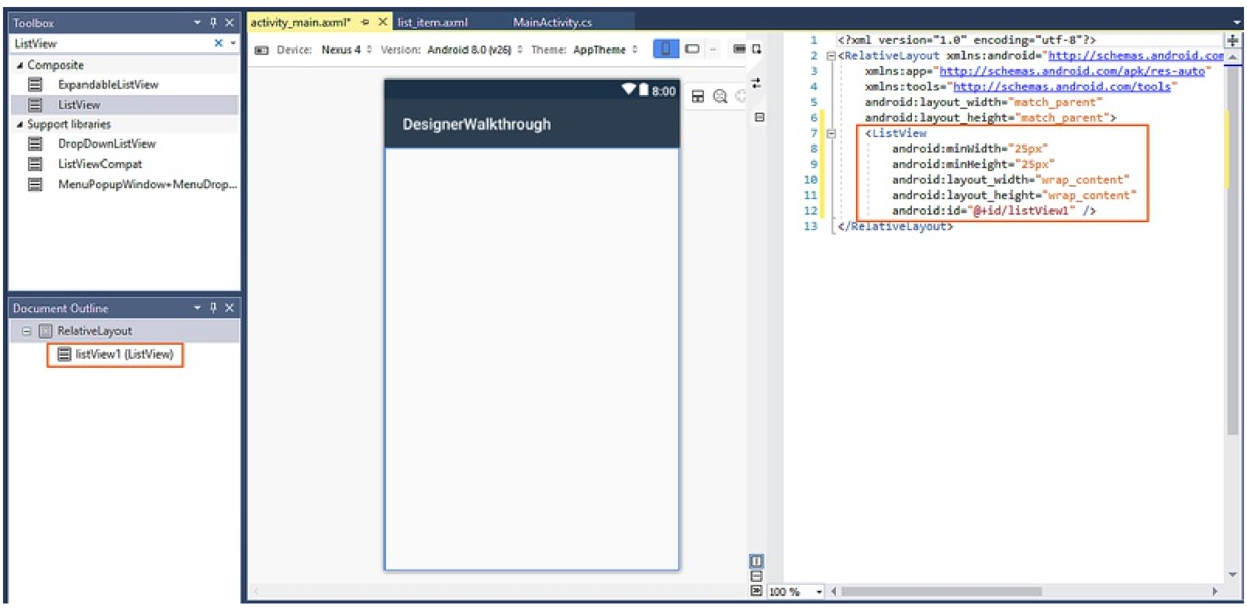


This removes `android:src="@android:drawable/ic_menu_gallery"` from the source XML for that `ImageView`.

Adding a ListView container

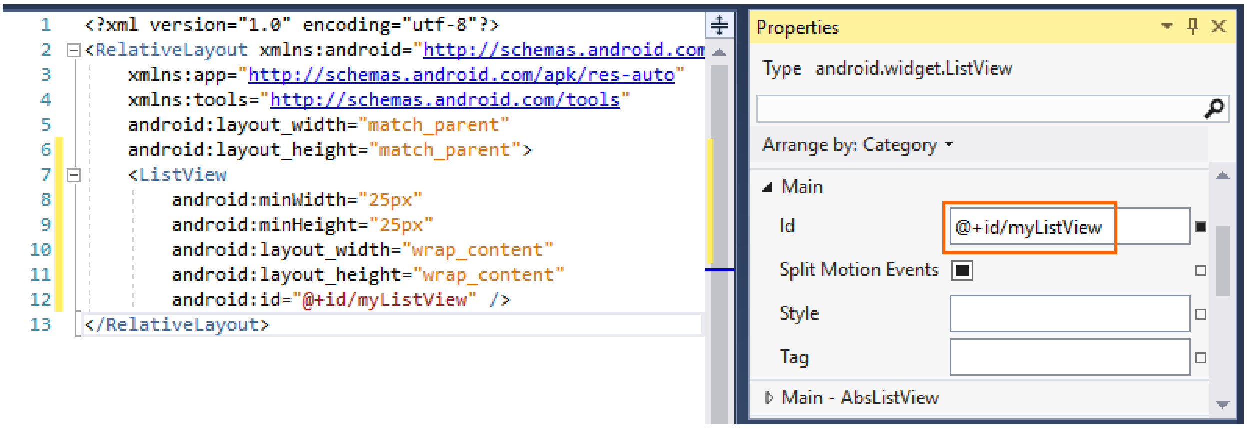
Now that the **list_item** layout is defined, the next step is to add a `ListView` to the Main layout. This `ListView` will contain a list of **list_item**.

In the **Solution Explorer**, open **Resources/layout/activity_main.axml**. In the **ToolBox**, locate the `ListView` widget and drag it onto the **Design Surface**. The `ListView` in the Designer will be blank except for blue lines that outline its border when it is selected. You can view the **Document Outline** to verify that the **ListView** was added correctly:



By default, the `ListView` is given an `Id` value of `@+id/listView1`. While `listView1` is still selected in the

Document Outline, open the **Properties** pane, click **Arrange by**, and select **Category**. Open **Main**, locate the **Id** property, and change its value to `@+id/myListView`:



At this point, the user interface is ready to use.

Running the application

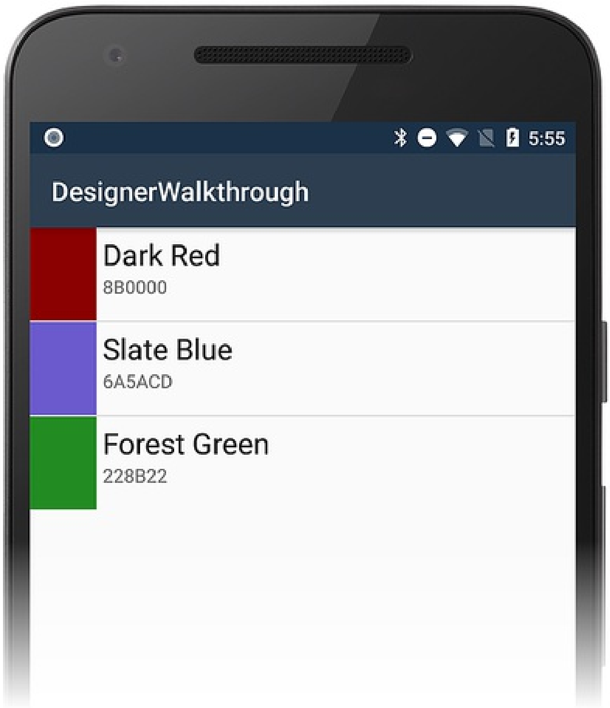
Open **MainActivity.cs** and replace its code with the following:

```
using Android.App;
using Android.Widget;
using Android.Views;
using Android.OS;
using Android.Support.V7.App;
using System.Collections.Generic;

namespace DesignerWalkthrough
{
    [Activity(Label = "@string/app_name", Theme = "@style/AppTheme", MainLauncher = true)]
    public class MainActivity : AppCompatActivity
    {
        List<ColorItem> colorItems = new List<ColorItem>();
        ListView listView;

        protected override void OnCreate(Bundle savedInstanceState)
        {
            base.OnCreate(savedInstanceState);

            // Set our view from the "main" layout resource
            SetContentView(Resource.Layout.activity_main);
            listView = FindViewById<ListView>(Resource.Id.myListView);

            colorItems.Add(new ColorItem()
            {
                Color = Android.Graphics.Color.DarkRed,
                ColorName = "Dark Red",
                Code = "8B0000"
            });
            colorItems.Add(new ColorItem()
            {
                Color = Android.Graphics.Color.SlateBlue,
                ColorName = "Slate Blue",
                Code = "6A5ACD"
            });
            colorItems.Add(new ColorItem()
            {
                Color = Android.Graphics.Color.ForestGreen,
                ColorName = "Forest Green",
                Code = "228B22"
            });

            listView.Adapter = new ColorAdapter(this, colorItems);
        }
```

```
    }

    public class ColorAdapter : BaseAdapter<ColorItem>
    {
        List<ColorItem> items;
        Activity context;
        public ColorAdapter(Activity context, List<ColorItem> items)
            : base()
        {
            this.context = context;
            this.items = items;
        }
        public override long GetItemId(int position)
        {
            return position;
        }
        public override ColorItem this[int position]
        {
            get { return items[position]; }
        }
        public override int Count
        {
            get { return items.Count; }
        }
        public override View GetView(int position, View convertView, ViewGroup parent)
        {
            var item = items[position];

            View view = convertView;
            if (view == null) // no view to re-use, create new
                view = context.LayoutInflater.Inflate(Resource.Layout.list_item, null);
            view.FindViewById<TextView>(Resource.Id.textView1).Text = item.ColorName;
            view.FindViewById<TextView>(Resource.Id.textView2).Text = item.Code;
            view.FindViewById<ImageView>(Resource.Id.imageView1).SetBackgroundColor(item.Color);

            return view;
        }
    }

    public class ColorItem
    {
        public string ColorName { get; set; }
        public string Code { get; set; }
        public Android.Graphics.Color Color { get; set; }
    }
}
```

This code uses a custom `ListView` adapter to load color information and to display this data in the UI that was just created. To keep this example short, the color information is hard-coded in a list, but the adapter could be modified to extract color information from a data source or to calculate it on the fly. For more information about `ListView` adapters, see ListView.

Build and run the application. The following screenshot is an example of how the app appears when running on a device:



Summary

This article walked through the process of using the Xamarin.Android Designer in Visual Studio to create a user interface for a basic app. It demonstrated how to create the interface for a single item in a list, and it illustrated how to add widgets and lay them out visually. It also explained how to assign resources and then set various properties on those widgets.

Xamarin.Android Designer basics

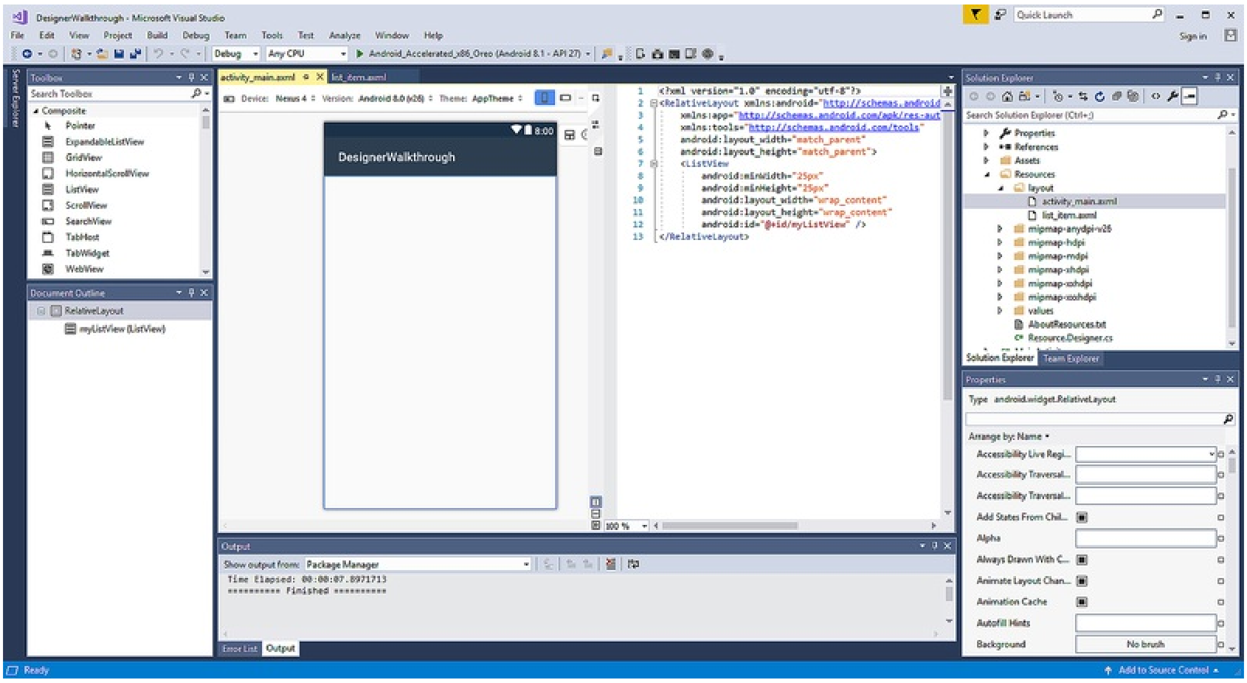
7/8/2021 • 22 minutes to read • Edit Online

This topic introduces Xamarin.Android Designer features, explains how to launch the Designer, describes the Design Surface, and details how to use the Properties pane to edit widget properties.

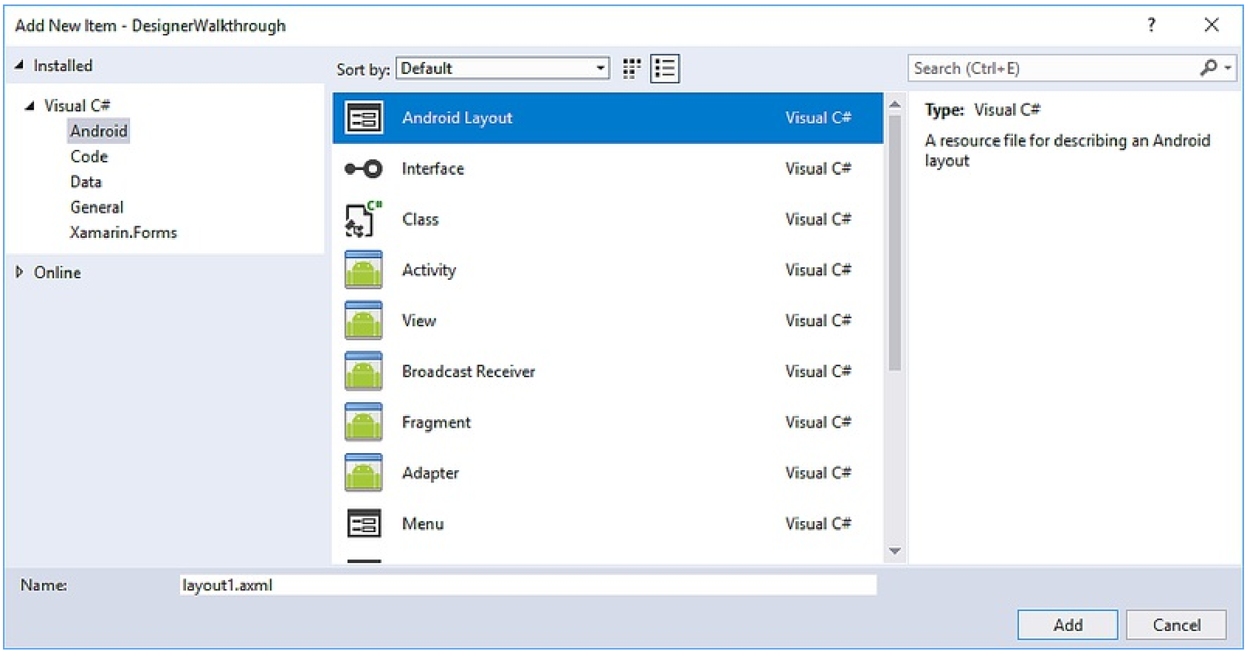
- Visual Studio
- Visual Studio for Mac

Launching the Designer

The Designer is launched automatically when a layout is created, or it can be launched by double-clicking an existing layout file. For example, double-clicking **activity_main.axml** in the **Resources > Layout** folder will load the Designer as seen in this screenshot:



Likewise, you can add a new layout by right-clicking the **layout** folder in the **Solution Explorer** and selecting **Add > New Item... > Android Layout**:

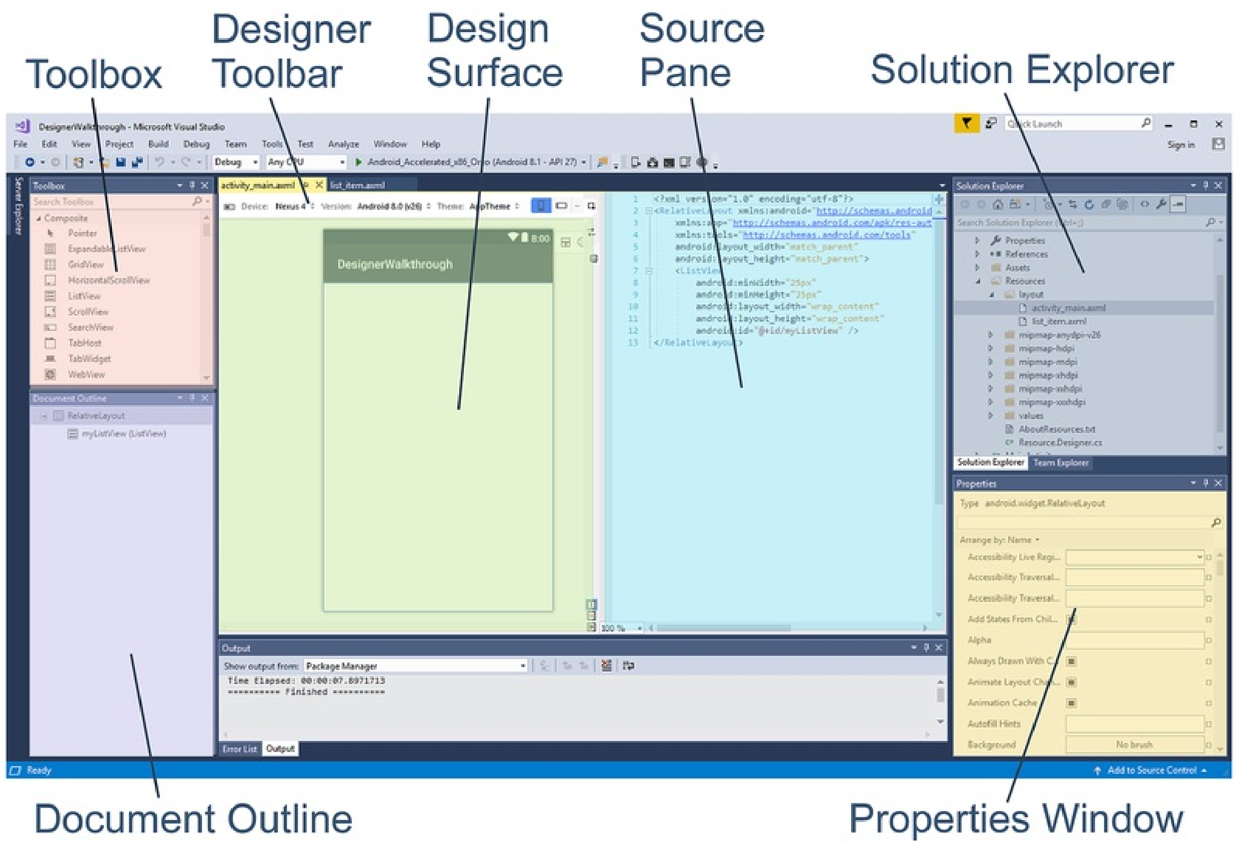


This creates a new **.axml** layout file and loads it into the Designer.

> **TIP**
>
> Newer releases of Visual Studio support opening .xml files inside the Android Designer.
>
> Both .axml and .xml files are supported in the Android Designer.

Designer features

The Designer is composed of several sections that support its various features, as shown in the following screenshot:

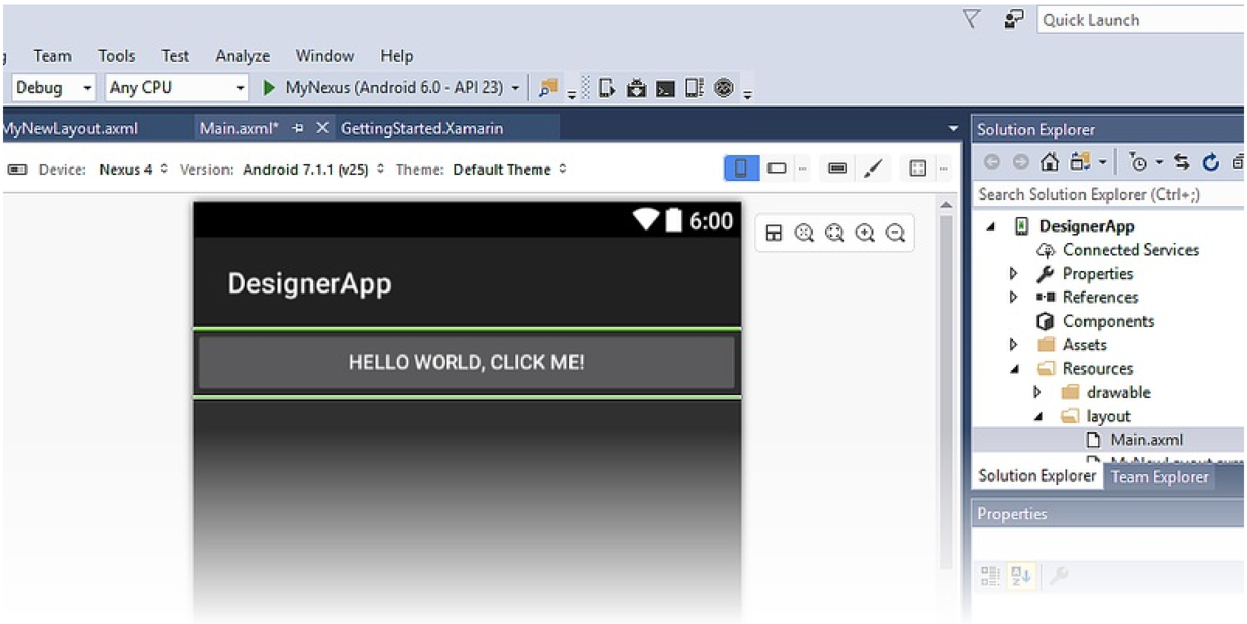


When you edit a layout in the Designer, you use the following features to create and shape your design:

- **Design Surface** – Facilitates the visual construction of the user interface by giving you an editable representation of how the layout will appear on the device. The **Design Surface** is displayed inside the **Design Pane** (with the **Designer Toolbar** positioned above it).
- **Source Pane** – Provides a view of the underlying XML source that corresponds to the design presented on the **Design Surface**.
- **Designer Toolbar** – Displays a list of selectors: **Device**, **Version**, **Theme**, layout configuration, and Action Bar settings. The **Designer Toolbar** also includes icons for launching the Theme Editor and for enabling the Material Design Grid.
- **Toolbox** – Provides a list of widgets and layouts that you can drag and drop onto the **Design Surface**.
- **Properties Window** – Lists the properties of the selected widget for viewing and editing.
- **Document Outline** – Displays the tree of widgets that compose the layout. You can click an item in the tree to cause it to be selected on the **Design Surface**. Also, clicking an item in the tree loads the item's properties into the **Properties** window.

Design Surface

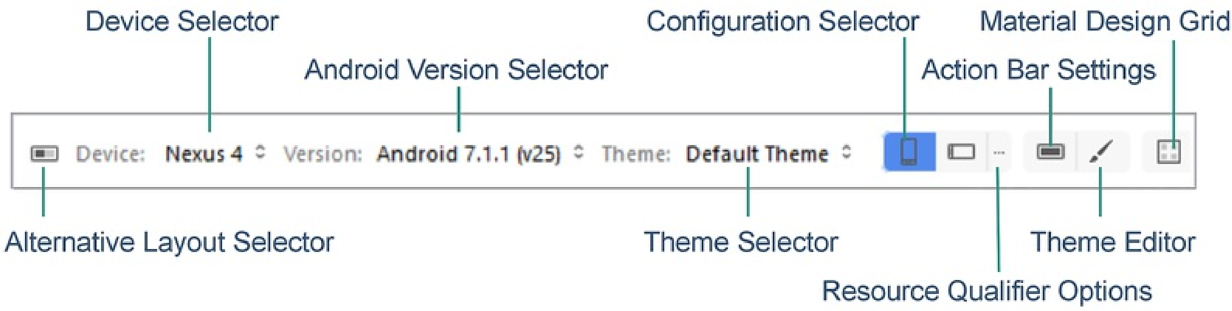
The Designer makes it possible for you to drag and drop widgets from the toolbox onto the **Design Surface**. When you interact with widgets in the Designer (by either adding new widgets or repositioning existing ones), vertical and horizontal lines are displayed to mark the available insertion points. In the following example, a new `Button` widget is being dragged to the **Design Surface**:



Additionally, widgets can be copied: you can use copy and paste to copy a widget, or you can drag and drop an existing widget while pressing the CTRL key.

Designer Toolbar

The **Designer Toolbar** (positioned above the **Design Surface**) presents configuration selectors and tool menus:



The **Designer Toolbar** provides access to the following features:

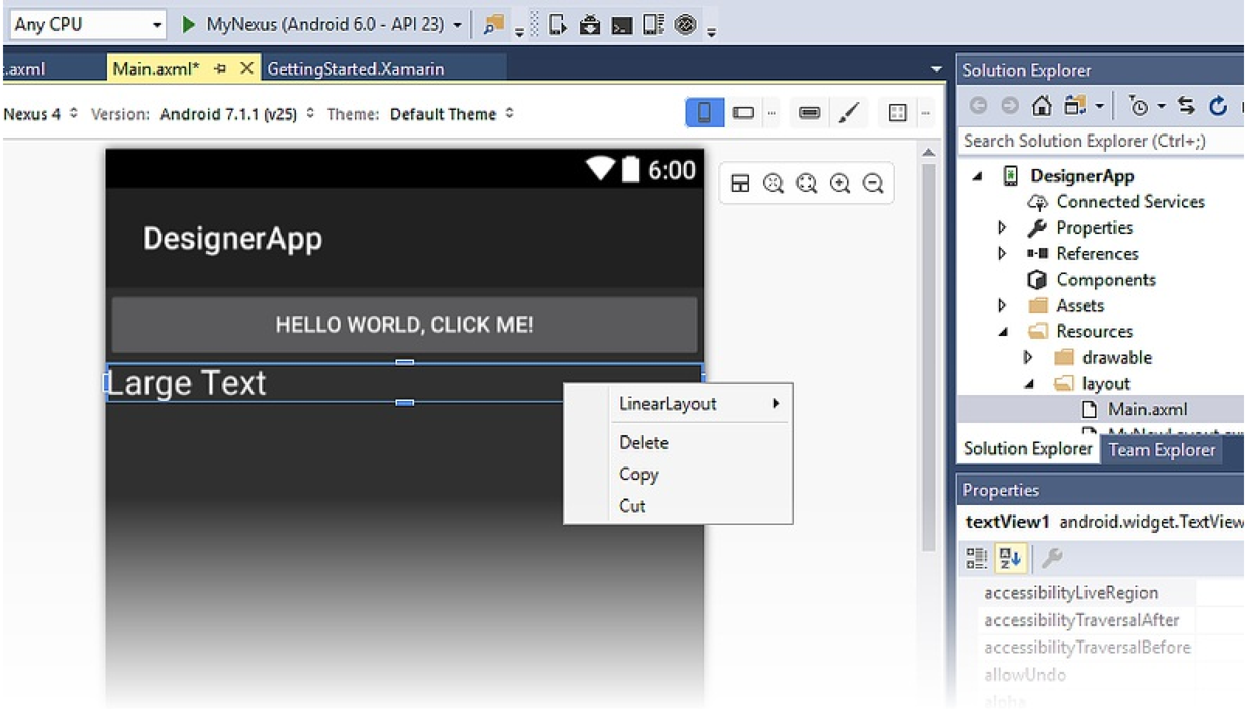
- **Alternative Layout Selector** – Allows you to select from different layout versions.
- **Device Selector** – Defines a set of qualifiers (such as screen size, resolution, and keyboard availability) associated with a particular device. You can also add and delete new devices.
- **Android Version Selector** – The Android version that the layout is targeting. The Designer will render the layout according to the selected Android version.
- **Theme Selector** – Selects the UI theme for the layout.
- **Configuration Selector** – Selects the device configuration, such as *portrait* or *landscape*.
- **Resource Qualifier Options** – Opens a dialog that presents drop-down menus for selecting *Language*, *UI Mode*, *Night Mode*, and *Round Screen* options.
- **Action Bar Settings** – Configures the Action Bar settings for the layout.
- **Theme Editor** – Opens the *Theme Editor*, which makes it possible for you to customize elements of the selected theme.
- **Material Design Grid** – Enables or disables the *Material Design Grid*. The drop-down menu item adjacent to the Material Design Grid opens a dialog that enables you to customize the grid.

Each of these features is explained in more detail in these topics:

- Resource Qualifiers and Visualization Options provides detailed information about the **Device Selector**, **Android Version Selector**, **Theme Selector**, **Configuration Selector**, **Resource Qualifications Options**, and **Action Bar Settings**.
- Alternative Layout Views explains how to use the **Alternative Layout Selector**.
- Xamarin.Android Designer Material Design Features provides a comprehensive overview of the **Theme Editor** and the **Material Design Grid**.

Context menu commands

A context menu is available both in the **Design Surface** and in the **Document Outline**. This menu displays commands that are available for the selected widget and its container, making it easier for you to perform operations on containers (which are not always easy to select on the **Design Surface**). Here is an example of a context menu:

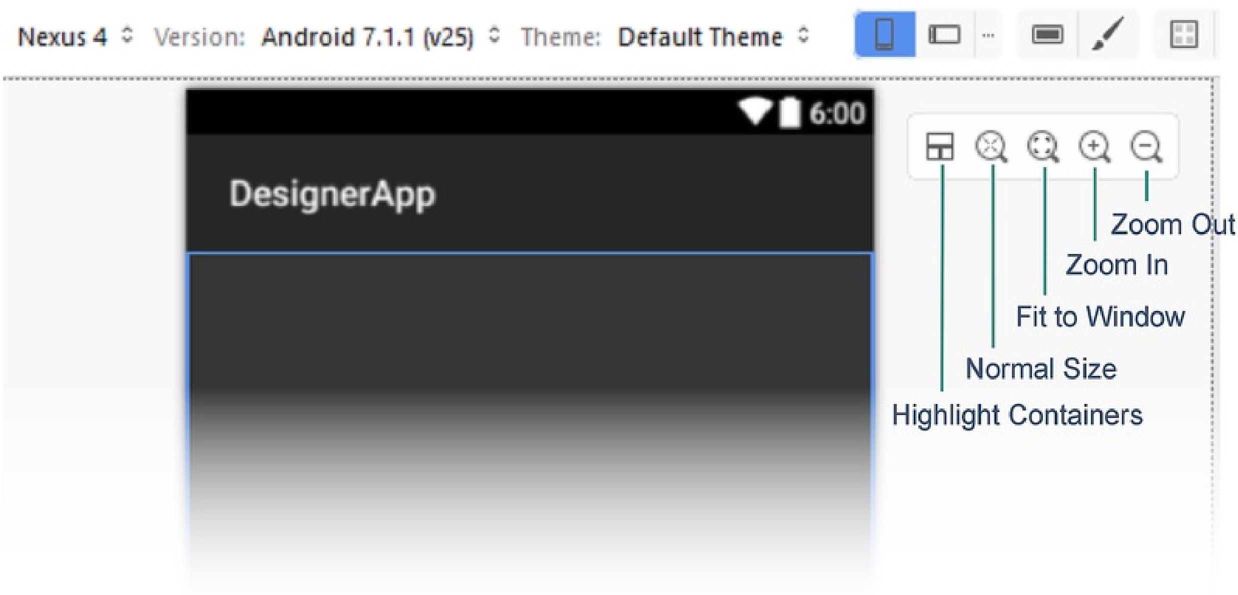


In this example, right-clicking a `TextView` opens a context menu that provides several options:

- **LinearLayout** – opens a submenu for editing the `LinearLayout` parent of the `TextView`.
- **Delete**, **Copy**, and **Cut** – operations that apply to the right-clicked `TextView`.

Zoom controls

The **Design Surface** supports zooming via several controls as shown:



These controls make it easier to see certain areas of the user interface in the Designer:

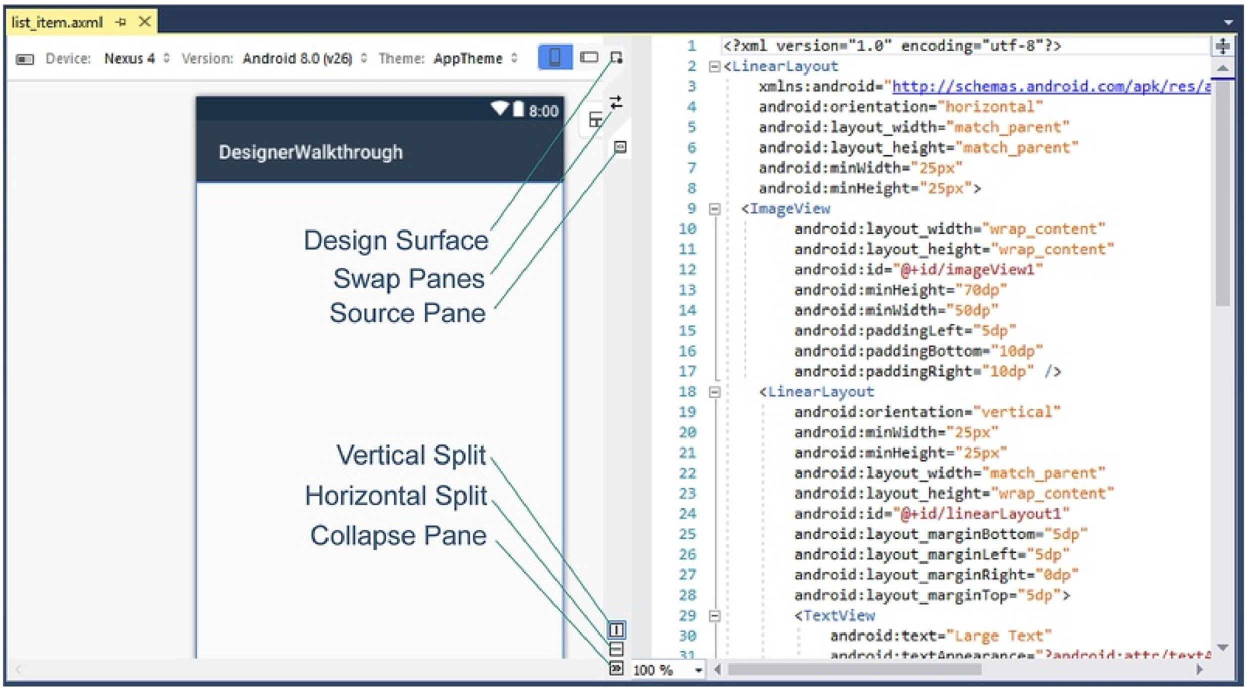
- **Highlight Containers** – Highlights containers on the **Design Surface** so that they are easier to locate while zooming in and out.
- **Normal Size** – Renders the layout pixel-for-pixel so that you can see how the layout will look at the resolution of the selected device.
- **Fit to Window** – Sets the zoom level so that the entire layout is visible on the Design Surface.
- **Zoom In** – Zooms in incrementally with each click, magnifying the layout.

- **Zoom Out** – Zooms out incrementally with each click, making the layout appear smaller on the Design Surface.

Note that the chosen zoom setting does not affect the user interface of the application at runtime.

Switching between Design and Source panes

In the center strip between the **Design** and **Source** panes, there are several buttons that are used to modify how the **Design** and **Source** panes are displayed:



These buttons do the following:

- **Design** – This topmost button, **Design**, selects the **Design** pane. When this button is clicked, the **Toolbox** and **Properties** panes are enabled and the **Text Editor Toolbar** is not displayed. When the **Collapse** button is clicked (see below), the **Design** pane is presented alone without the **Source** pane.

- **Swap Panes** – This button (which resembles two opposing arrows) swaps the **Design** and **Source** panes so that the **Source** pane is on the left and the **Design** pane is on the right. Clicking it again swaps these panes back to their original locations.

- **Source** – This button (which resembles two opposing angle brackets) selects the **Source** pane. When this button is clicked, the **Toolbox** and **Properties** panes are disabled and the **Text Editor Toolbar** is made visible at the top of Visual Studio. When the **Collapse** button is clicked (see below), clicking the **Source** button displays the **Source** pane instead of the **Design** pane.

- **Vertical Split** – This button (which resembles a vertical bar), displays the **Design** and **Source** panes side-by-side. This is the default arrangement.

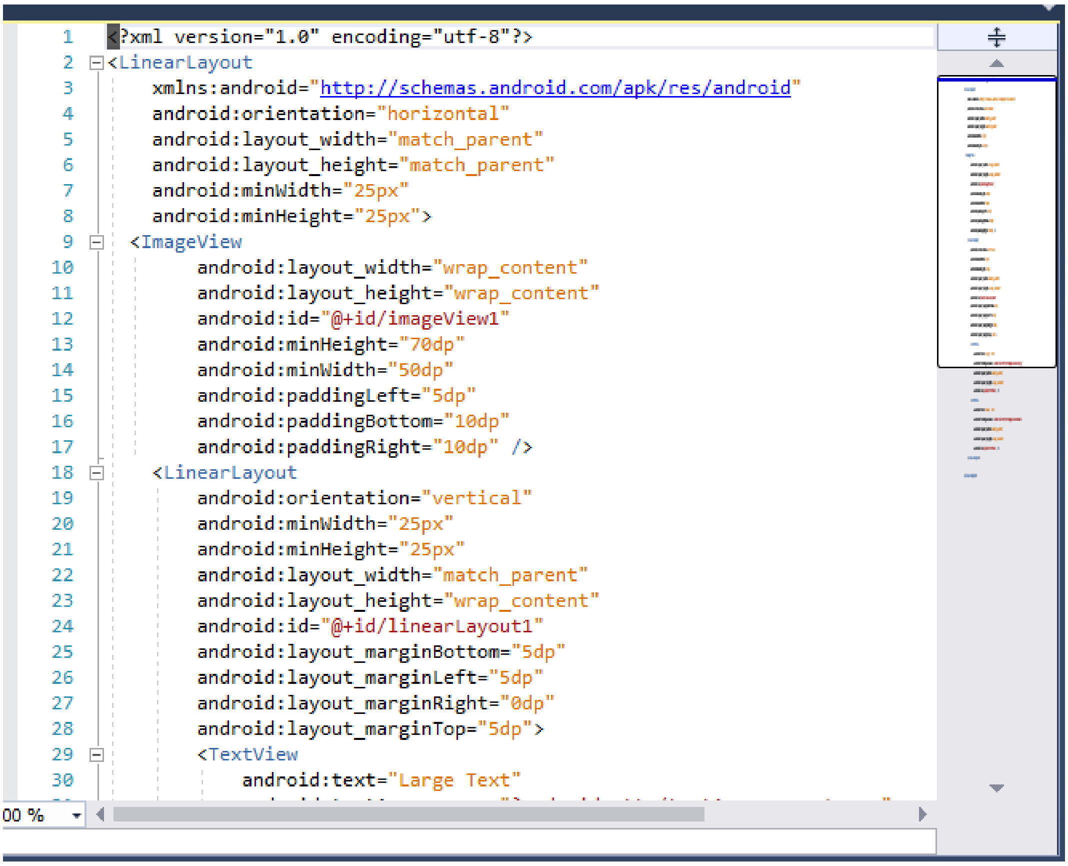
- **Horizontal Split** – This button (which resembles a horizontal bar), displays the **Design** pane above the **Source** pane. **Swap Panes** can be clicked to place the **Source** pane above the **Design** pane.

- **Collapse Pane** – This button (which resembles two right-pointing angle brackets) "collapses" the dual-pane display of **Design** and **Source** into a single view of one of these panes. This button becomes the **Expand Pane** button (resembling two left-pointing angle brackets), which can be clicked to return the view back to dual-pane (**Design** and **Source**) display mode.

When **Collapse Pane** is clicked, only the **Design** pane is displayed. However, you can click the **Source** button to instead view only the **Source** pane. Click the **Design** button again to return to the **Design** pane.

Source pane

The **Source** pane displays the XML source underlying the design shown on the **Design Surface**. Because both views are available at the same time, it is possible to create a UI design by going back and forth between a visual representation of the design and the underlying XML source for the design:

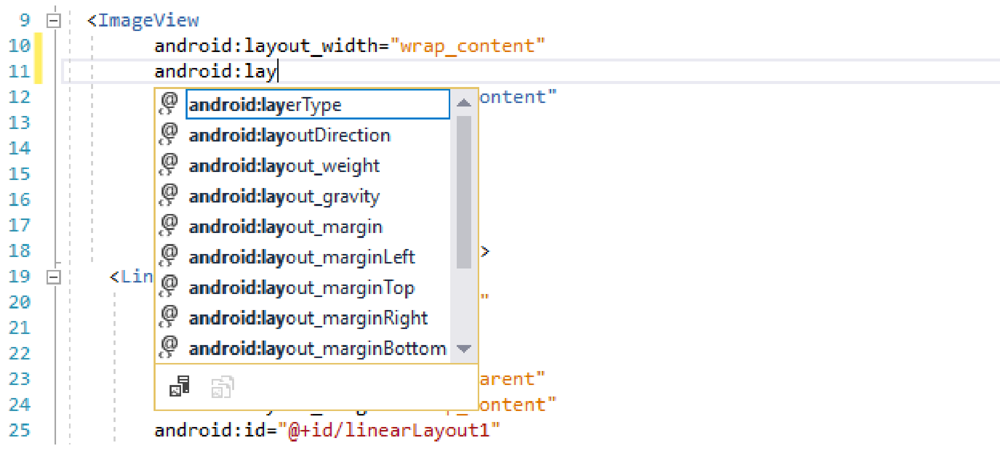


Changes made to the XML source are immediately rendered on the **Design Surface**; changes made on the **Design Surface** cause the XML source displayed in the **Source** pane to be updated accordingly. When you make changes to XML in the **Source** pane, autocompletion and IntelliSense features are available to speed XML-based UI development as explained next.

For greater navigational ease when working with long XML files, the **Source** pane supports the Visual Studio scrollbar (as seen on the right in the previous screenshot). For more information about the scrollbar, see How to Track Your Code by Customizing the Scrollbar.

Autocompletion

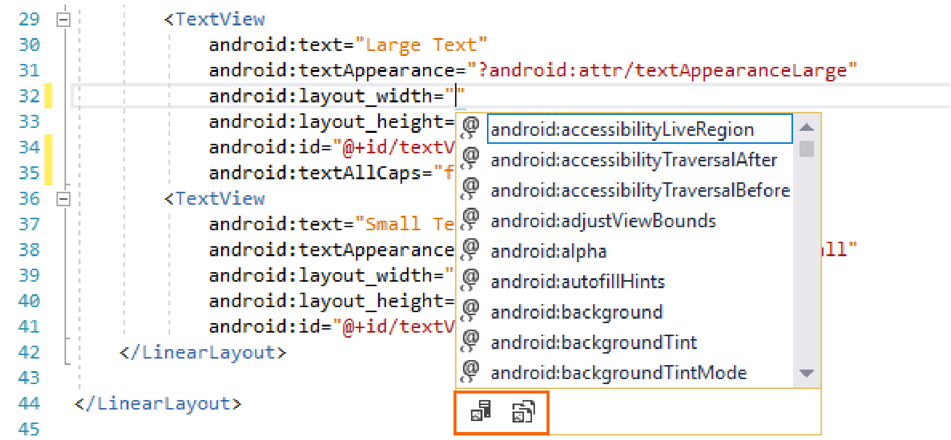
When you begin to type the name of an attribute for a widget, you can press CTRL+SPACE to see a list of possible completions. For example, after entering `android:lay` in the following example (followed by typing CTRL+SPACE), the following list is presented:



Press ENTER to accept the first listed completion, or use the arrow keys to scroll to the desired completion and press ENTER. Alternatively, you can use the mouse to scroll to and click the desired completion.

IntelliSense

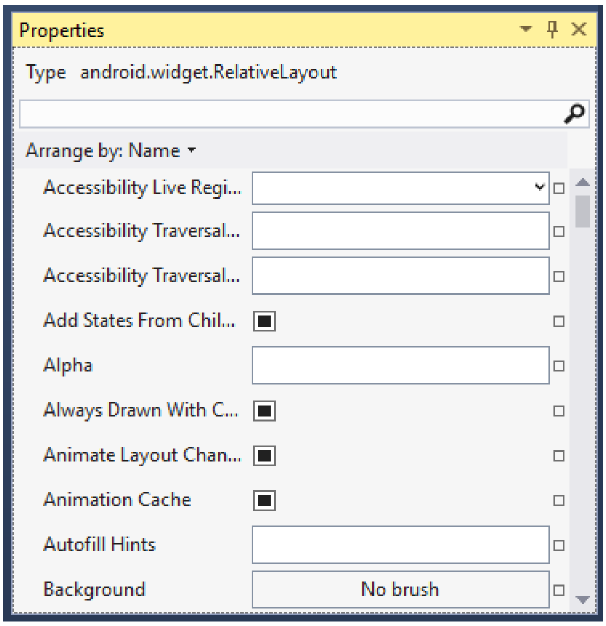
After you enter a new attribute for a widget and begin to assign it a value, IntelliSense pops up after a trigger character is typed and provides a list of valid values to use for that attribute. For example, after the first double-quote is entered for `android:layout_width` in the following example, an autocompletion selector pops up to provide the list of valid choices for this width:



At the bottom of this popup are two buttons (as outlined in red in the above screenshot). Clicking the **Project Resources** button on the left restricts the list to resources that are part of the app project, while clicking the **Framework Resources** button on the right restricts the list to display resources available from the framework. These buttons toggle on or off: you can click them again to disable the filtering action that each provides.

Properties pane

The Designer supports the editing of widget properties through the **Properties** pane:



The properties listed in the **Properties** pane change depending on which widget is selected on the **Design Surface**.

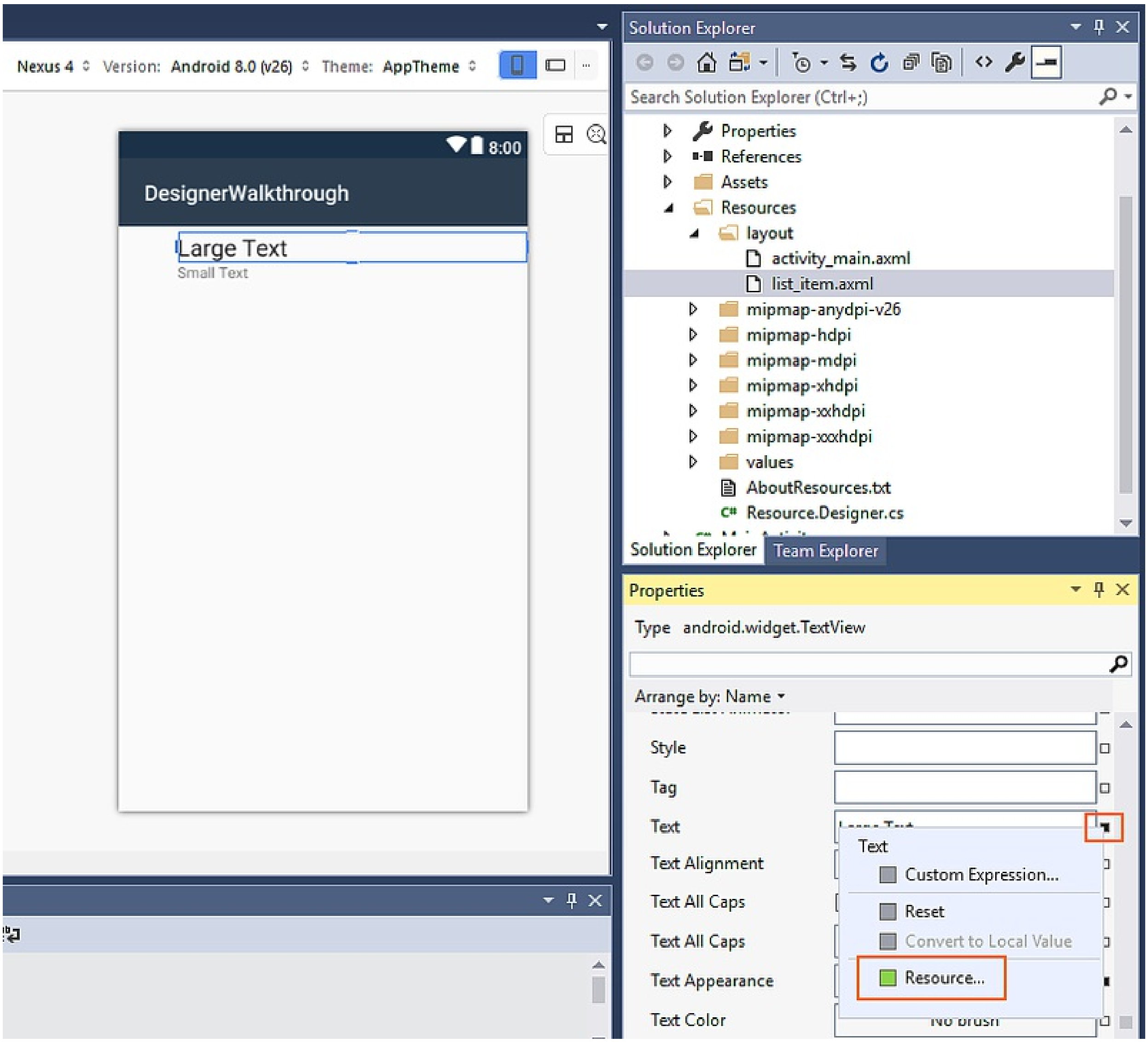
Default values

The properties of most widgets will be blank in the **Properties** window because their values inherit from the selected Android theme. The **Properties** window will only show values that are explicitly set for the selected widget; it will not show values that are inherited from the theme.

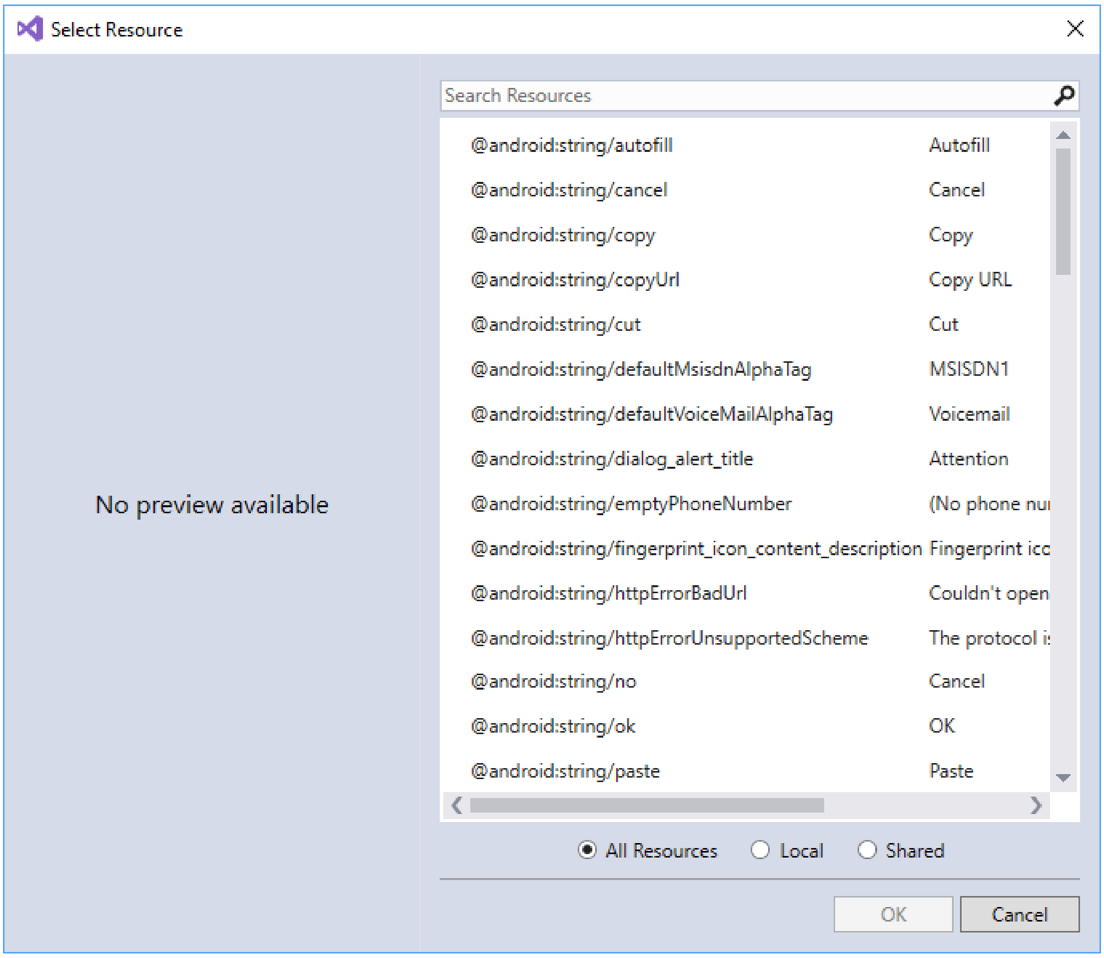
Referencing resources

Some properties can reference resources that are defined in files other than the layout **.axml** file. The most common cases of this type are `string` and `drawable` resources. However, references can also be used for other resources, such as `Boolean` values and dimensions. When a property supports resource references, a browse icon (a square) is shown next to the text entry for the property. This button opens a resource selector when clicked.

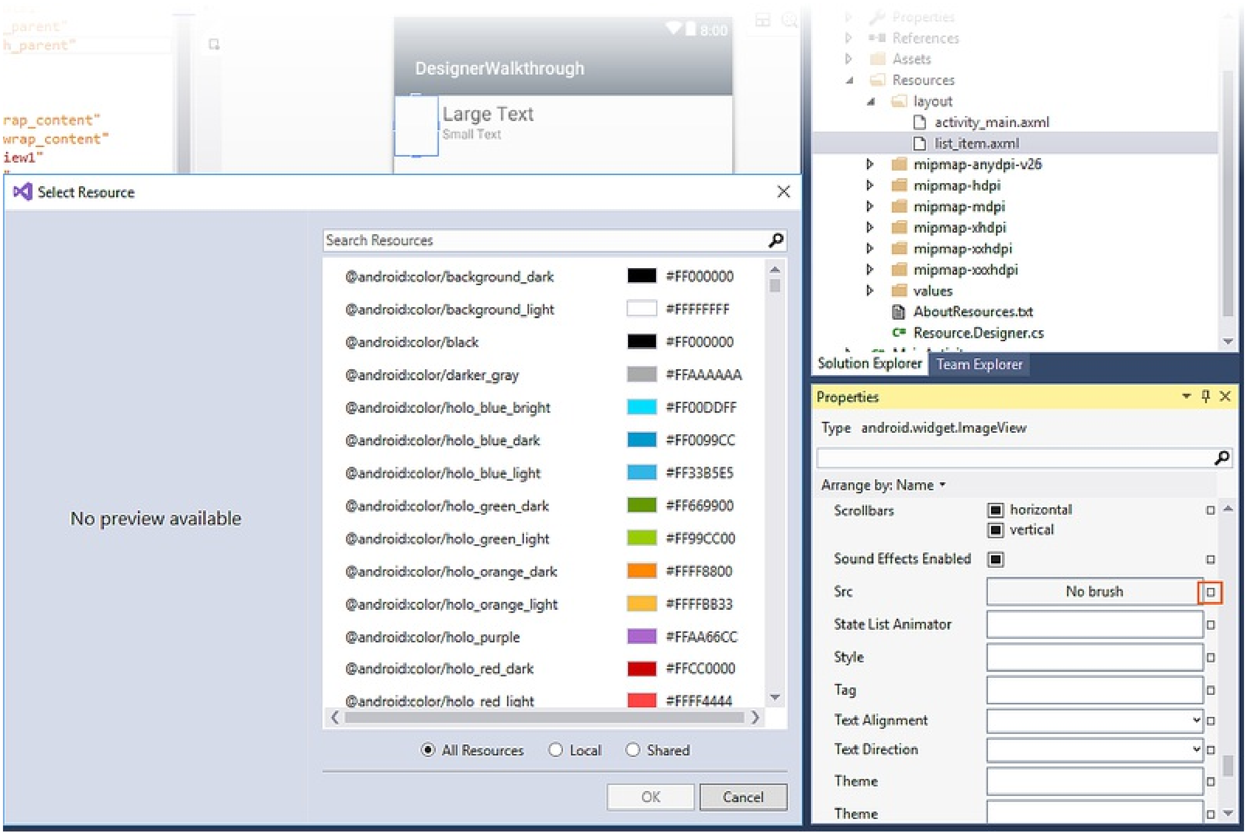
For example, the following screenshot shows the options available when clicking the darkened square to the right of the text field for a `Text` widget in the **Properties** window:



When **Resource...** is clicked, the **Select Resource** dialog is presented:



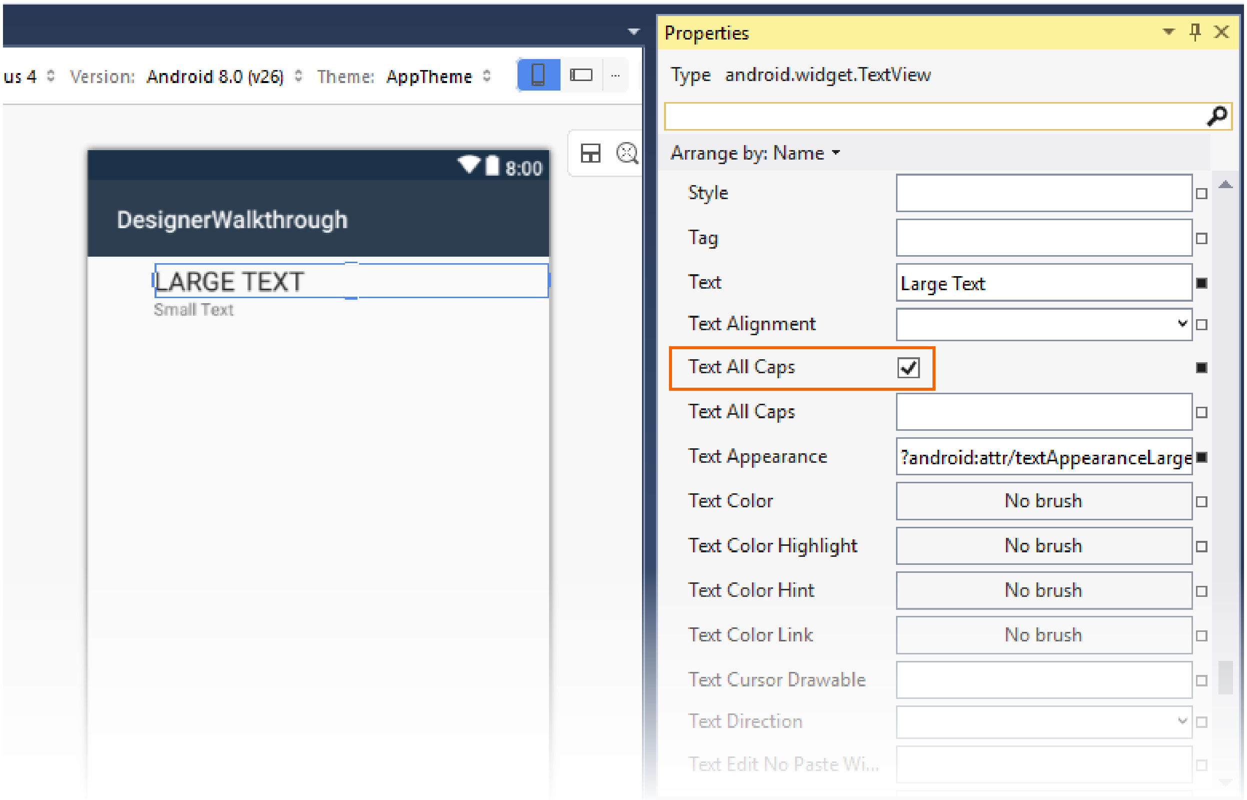
From this list, you can select a text resource to use for that widget instead of hard-coding the text in the **Properties** pane. The next example illustrates the resource selector for the `Src` property of an `ImageView`:



Clicking the blank square to the right of the `Src` property opens the **Select Resource** dialog with a list of resources ranging from colors (as shown above) to drawables.

Boolean property references

Boolean properties are normally selected as check marks next to a property in the Properties window. You can designate a `true` or `false` value by checking or unchecking this check box, or you can select a property reference by clicking the dark-filled square to the right of the property. In the following example, text is changed to all caps by clicking the **Text All Caps** boolean property reference associated with the selected `TextView`:

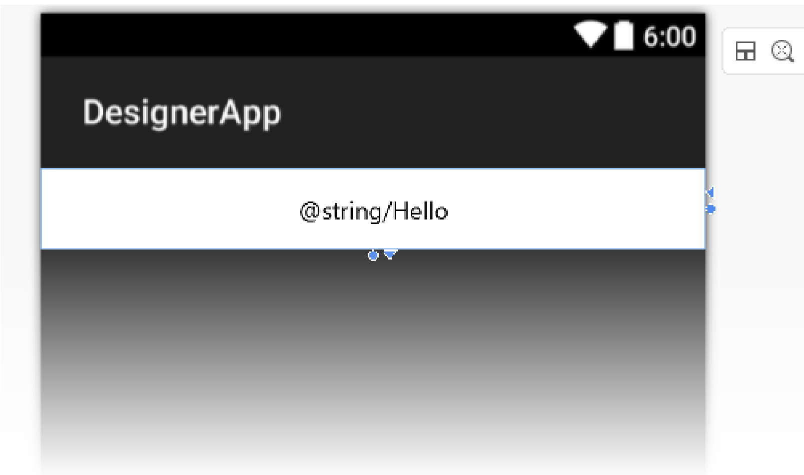


Editing properties inline

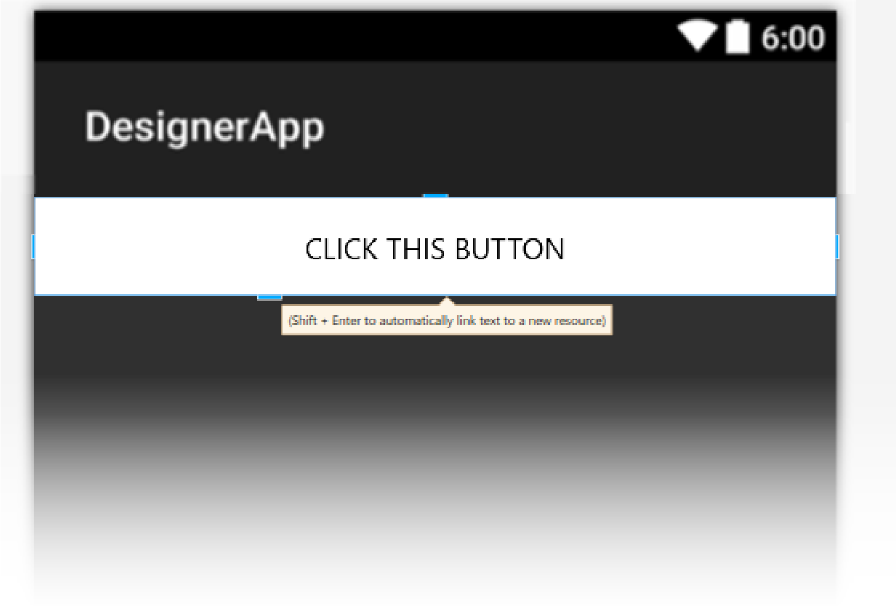
The Android Designer supports direct editing of certain properties on the **Design Surface** (so you don't have to search for these properties in the property list). Properties that can be directly edited include text, margin, and size.

Text

The text properties of some widgets (such as `Button` and `TextView`), can be edited directly on the **Design Surface**. Double-clicking a widget will put it into edit mode, as shown below:



You can enter a new text value or you can enter a new resource string. In the following example, the `@string/hello` resource is being replaced with the text, `CLICK THIS BUTTON`:

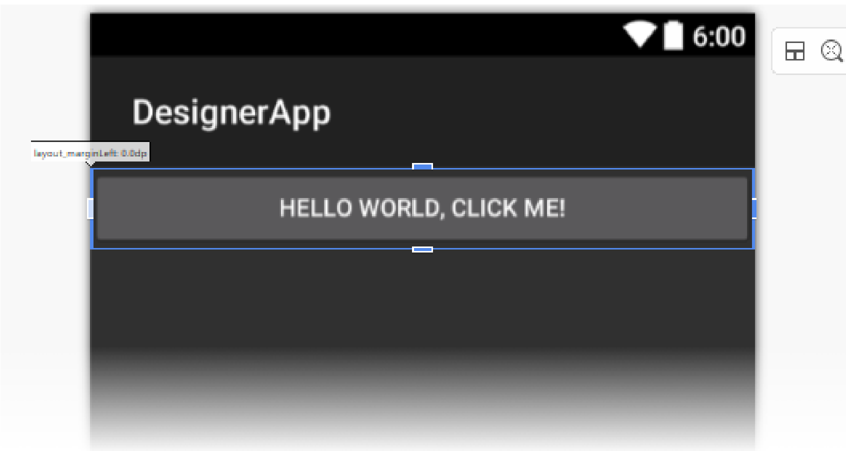


This change is stored in the widget's `text` property; it does not modify the value assigned to the `@string/hello` resource. When you key in a new text string, you can press `Shift + Enter` to automatically link the entered text to a new resource.

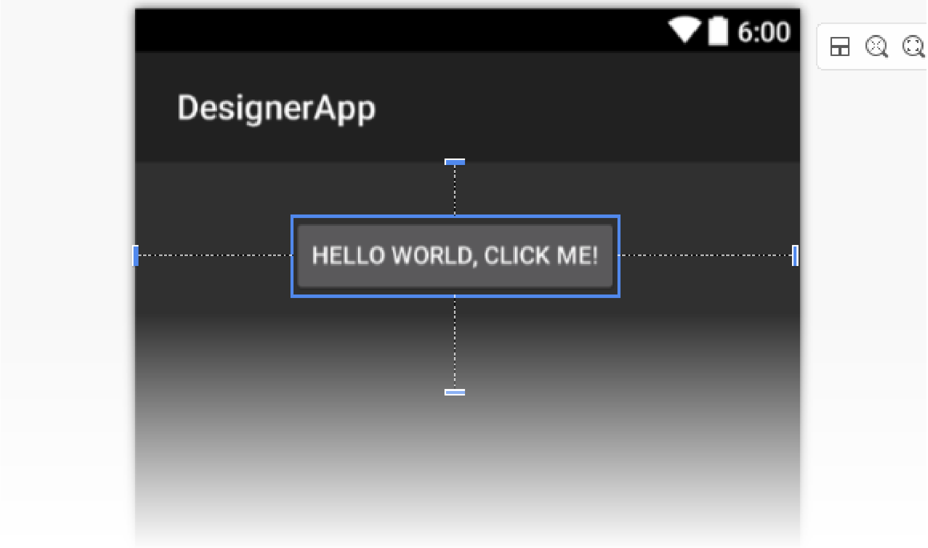
Margin

When you select a widget, the Designer displays handles that allow you to change the size or margin of the widget interactively. Clicking the widget while it is selected toggles between margin-editing mode and size-editing mode.

When you click a widget for the first time, margin handles are displayed. If you move the mouse to one of the handles, the Designer displays the property that the handle will change (as shown below for the `layout_marginLeft` property):



If a margin has already been set, dotted lines are displayed, indicating the space that the margin occupies:



Size

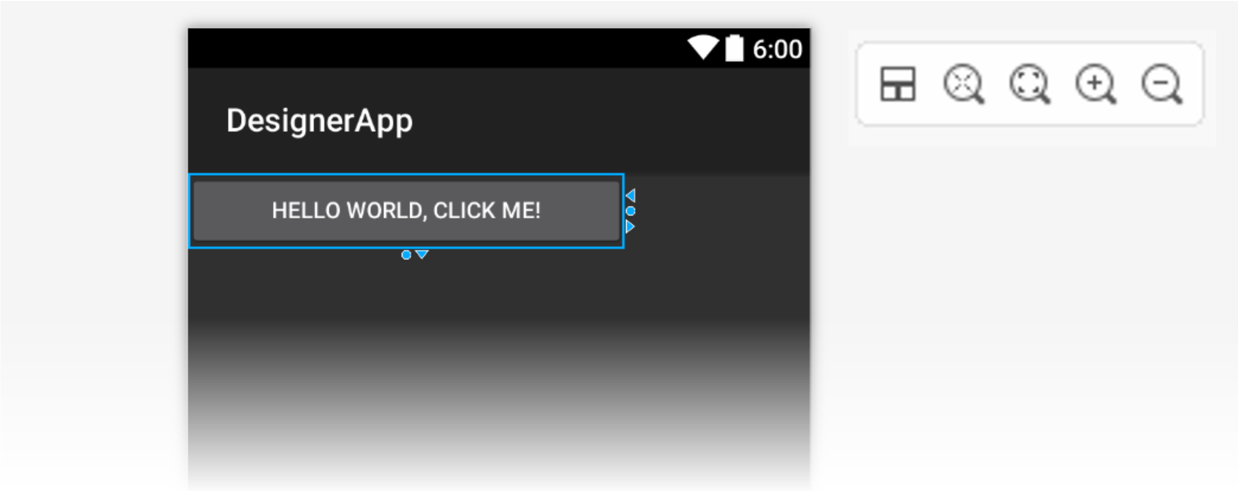
As mentioned earlier, you can switch to size-editing mode by clicking a widget while it is already selected. Click the triangular handle to set the size for the indicated dimension to `wrap_content`:

Clicking the **Wrap Content** handle shrinks the widget in that dimension so that it is no larger than necessary to wrap the enclosed content. In this example, the button text shrinks horizontally as shown in the next screenshot.

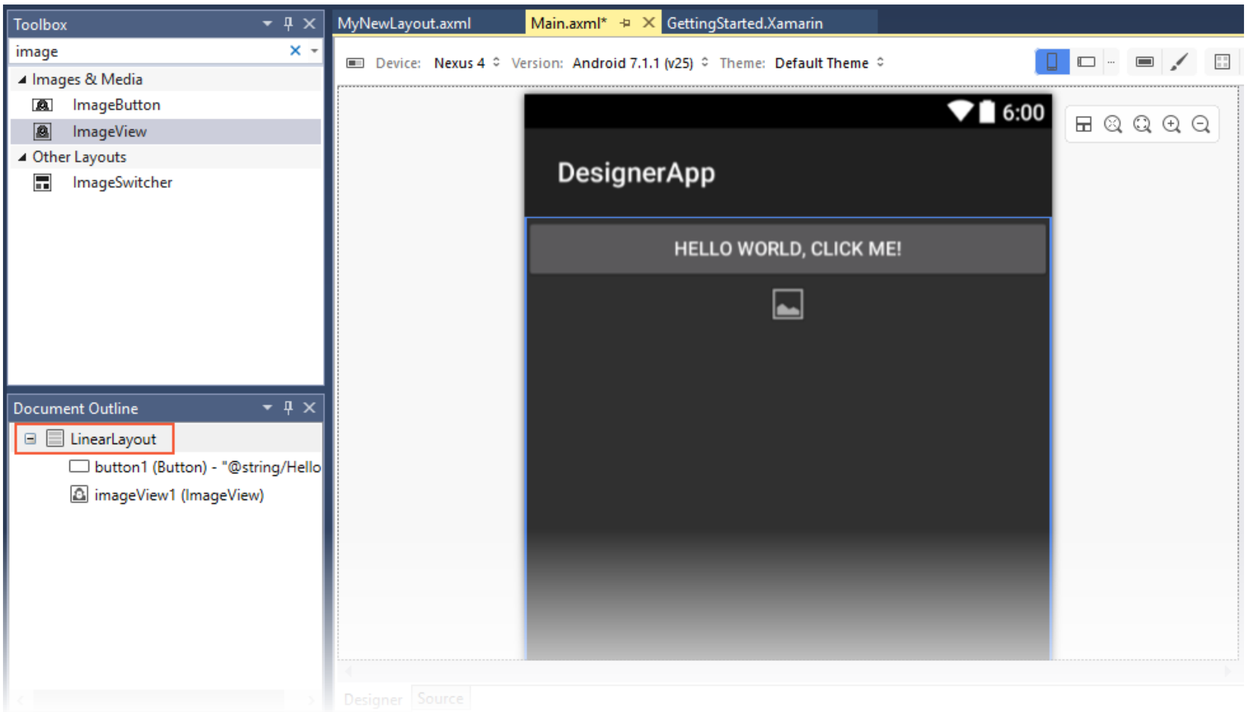
When the size value is set to **Wrap Content**, the Designer displays a triangular handle pointing in the opposite direction for changing the size to `match_parent`:

Clicking the **Match Parent** handle restores the size in that dimension so that it is the same as the parent widget.

Also, you can drag the circular resize handle (as shown in the above screenshots) to resize the widget to an arbitrary `dp` value. When you do so, both **Wrap Content** and **Match Parent** handles are presented for that dimension:

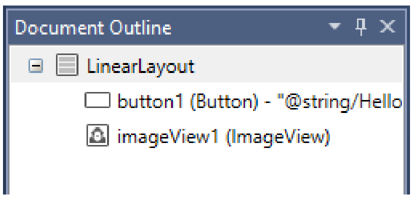


Not all containers allow editing the `Size` of a widget. For example, notice that in the screenshot below with the `LinearLayout` selected, the resize handles do not appear:



Document Outline

The **Document Outline** displays the widget hierarchy of the layout. In the following example, the containing `LinearLayout` widget is selected:



The outline of the selected widget (in this case, a `LinearLayout`) is also highlighted on the **Design Surface**. The selected widget in the Document Outline stays in sync with its counterpart on the **Design Surface**. This is useful for selecting view groups, which are not always easy to select on the **Design Surface**.

The **Document Outline** supports copy and paste, or you can use drag and drop. Drag and drop is supported from the **Document Outline** to the **Design Surface** as well as from the **Design Surface** to the **Document Outline**. Also, right-clicking an item in the **Document Outline** displays the context menu for that item (the same context menu that appears when you right-click that same widget on the **Design Surface**).

Resource qualifiers and visualization options

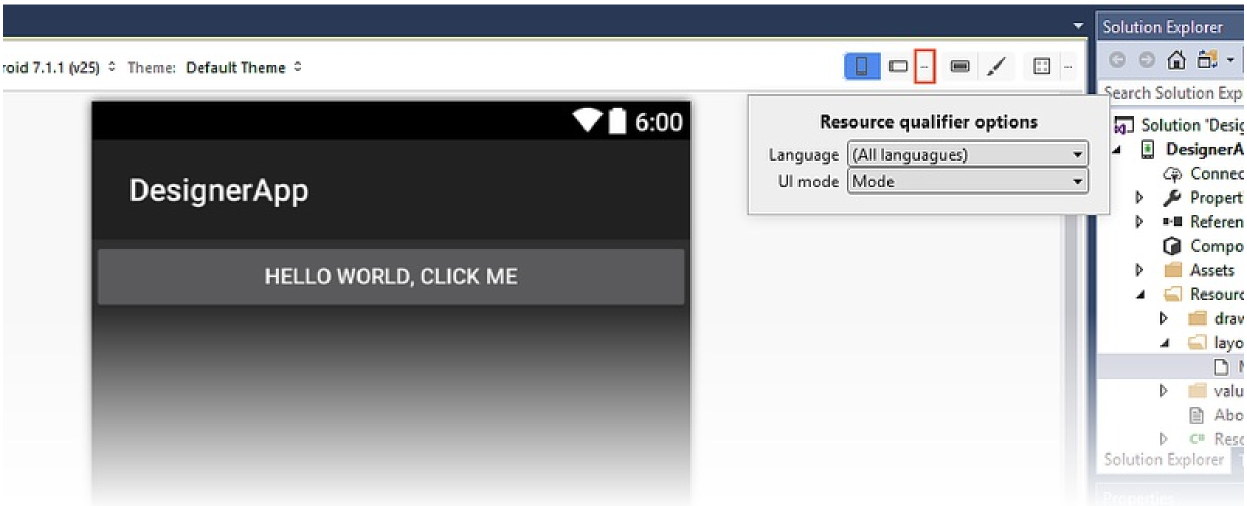
7/8/2021 • 7 minutes to read • Edit Online

This topic explains how to define resources that will be used only when some qualifier values are matched. A simple example is a language-qualified string resource. A string resource can be defined as the default, with other alternative resources defined to be used for additional languages. All resource types can be qualified, including the layout itself.

- Visual Studio
- Visual Studio for Mac

Resource qualifier options

Resource qualifier options can be accessed by clicking the ellipsis icon to the right of the **Landscape** mode button:



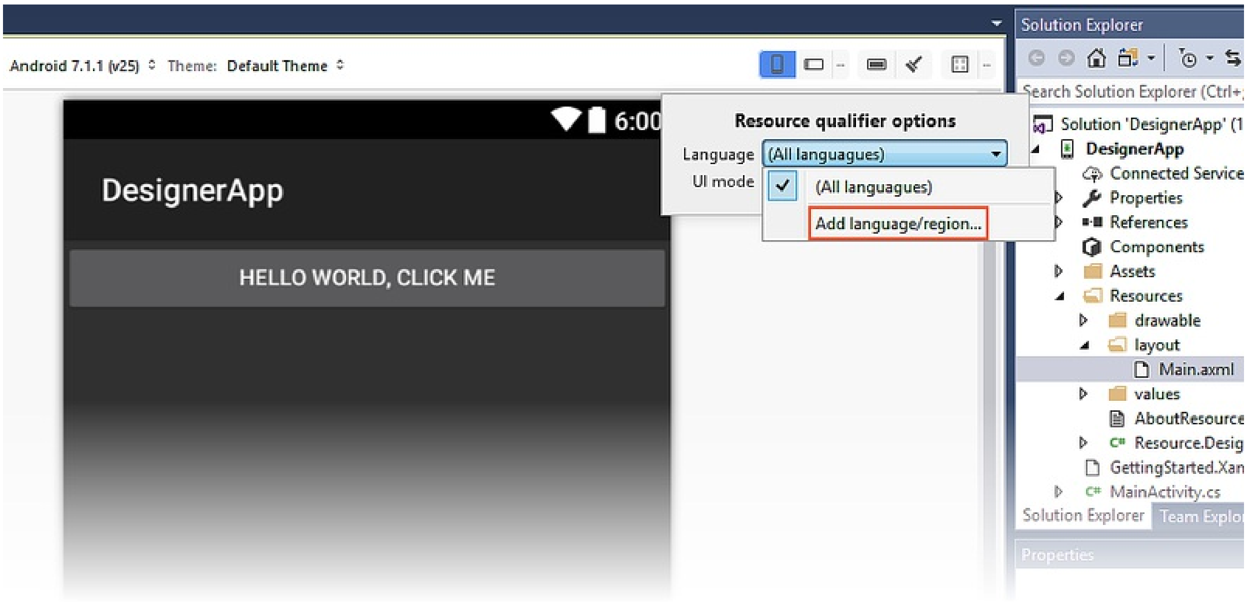
This dialog presents pull-down menus for the following resource qualifiers:

- **Language** – Displays available language resources and offers an option to add new language/region resources.
- **UI Mode** – Lists display modes (such as **Car Dock** and **Desk Dock**) as well as layout directions.

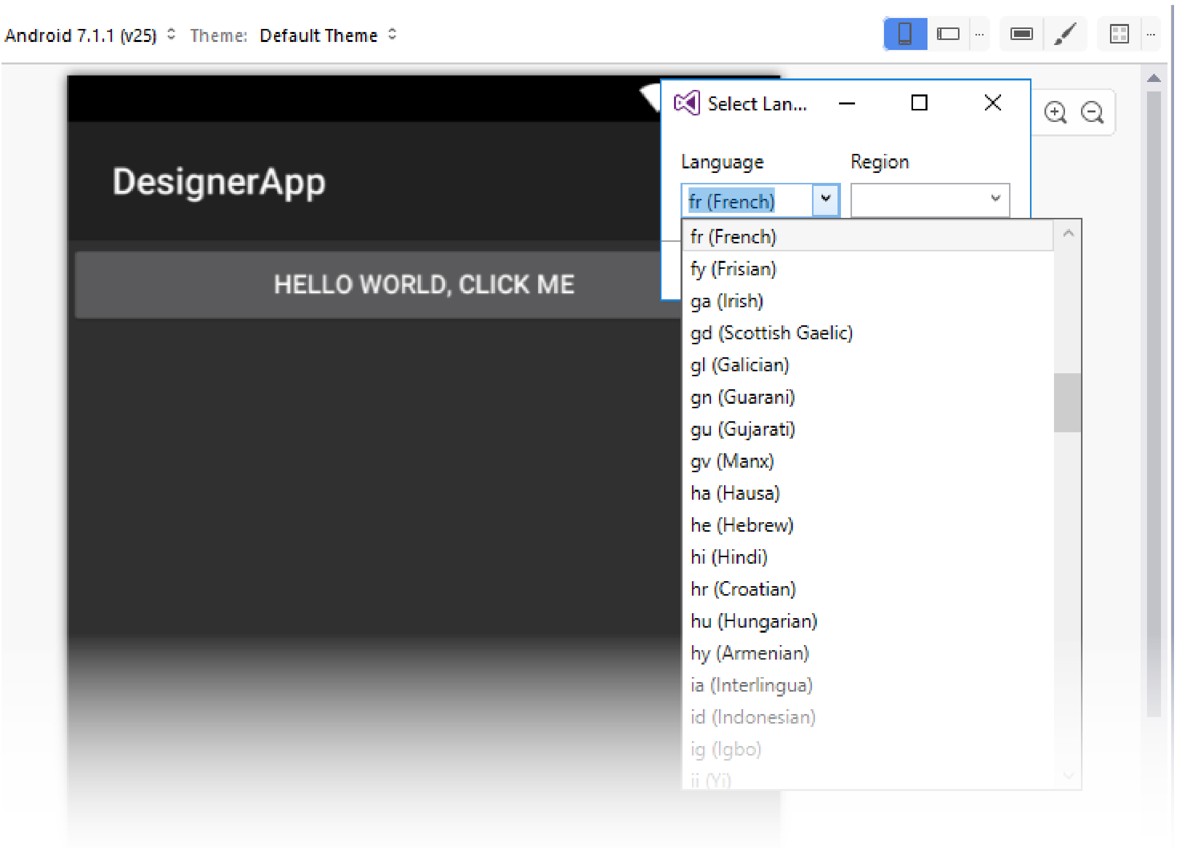
Each of these pull-down menus opens new dialog boxes where you can select and configure resource qualifiers (as explained next).

Language

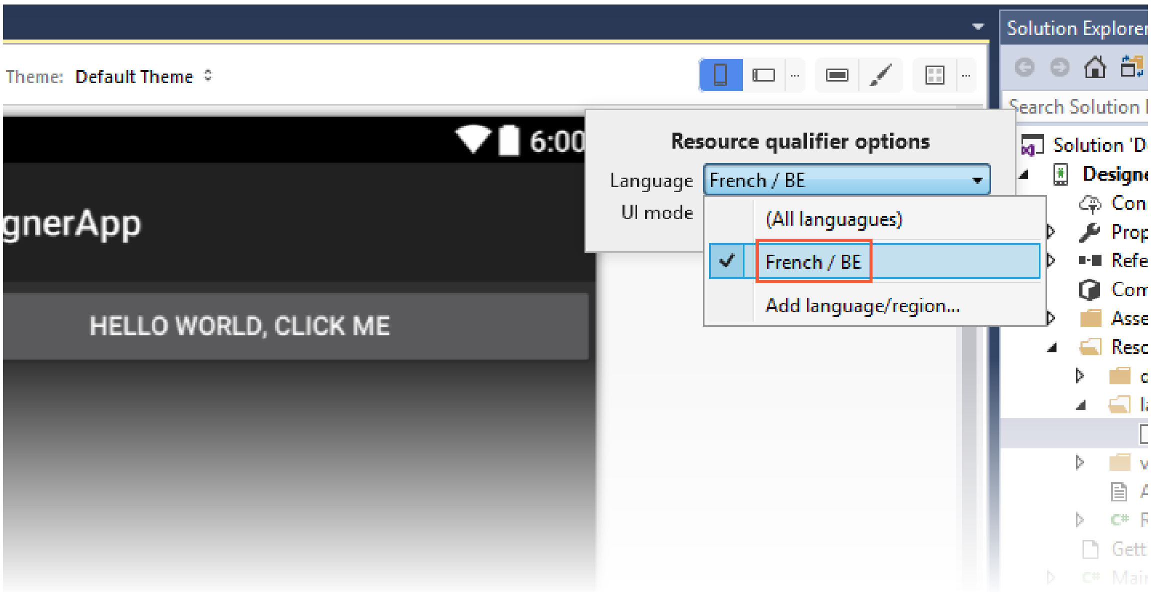
The **Language** pull-down menu lists only those languages that have resources defined (or **All languages**, which is the default). However, there is also an **Add language/region...** option that allows you to add a new language to the list:



When you click **Add language/region...**, the **Select Language** dialog opens to display drop-down lists of available languages and regions:



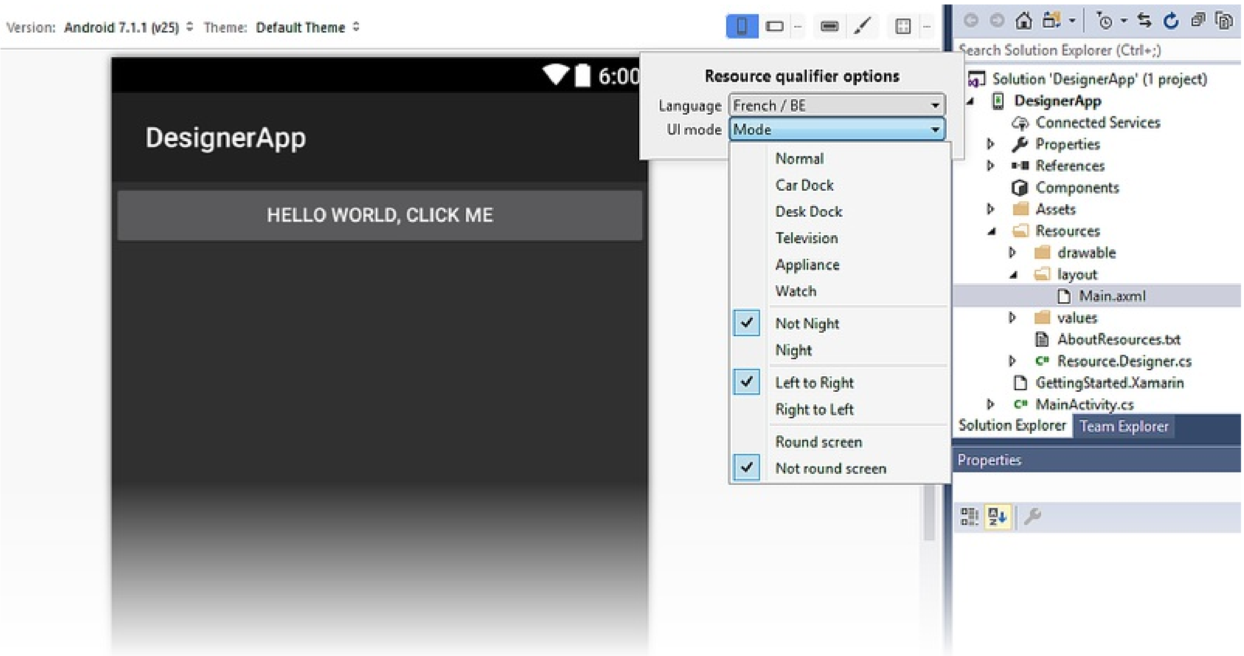
In this example, we have chosen **fr (French)** for the language and **BE** (Belgium) for the regional dialect of French. Note that the **Region** field is optional because many languages can be specified without regard for specific regions. When the **Language** pull-down menu is opened again, it displays the newly-added language/region resource:



Note that if you add a new language but you do not create new resources for it, the added language will no longer be shown the next time you open the project.

UI Mode

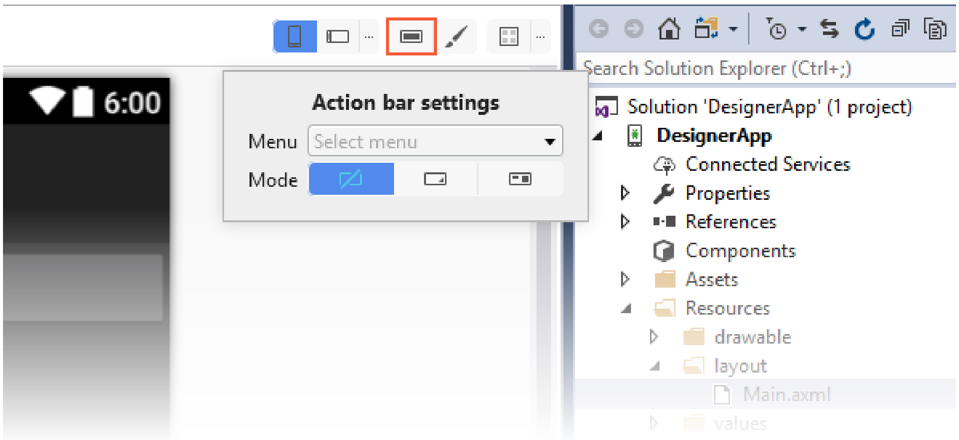
When you click the **UI Mode** pull-down menu, a list of modes is displayed, such as **Normal**, **Car Dock**, **Desk Dock**, **Television**, **Appliance**, and **Watch**:



Below this list are the night modes **Not Night** and **Night**, followed by the layout directions **Left to Right** and **Right to Left** (for information about **Left to Right** and **Right to Left** options, see LayoutDirection). The last items in the **Resource Qualifier Options** dialog are the **Round screens** (for use with Android Wear) or **Not Round screens**. For information about round and non-round screens, see Layouts. For more information about Android UI modes, see UiModeManager.

Action Bar settings

The **Action bar settings** icon is available to the left of the paintbrush (Theme Editor) icon:

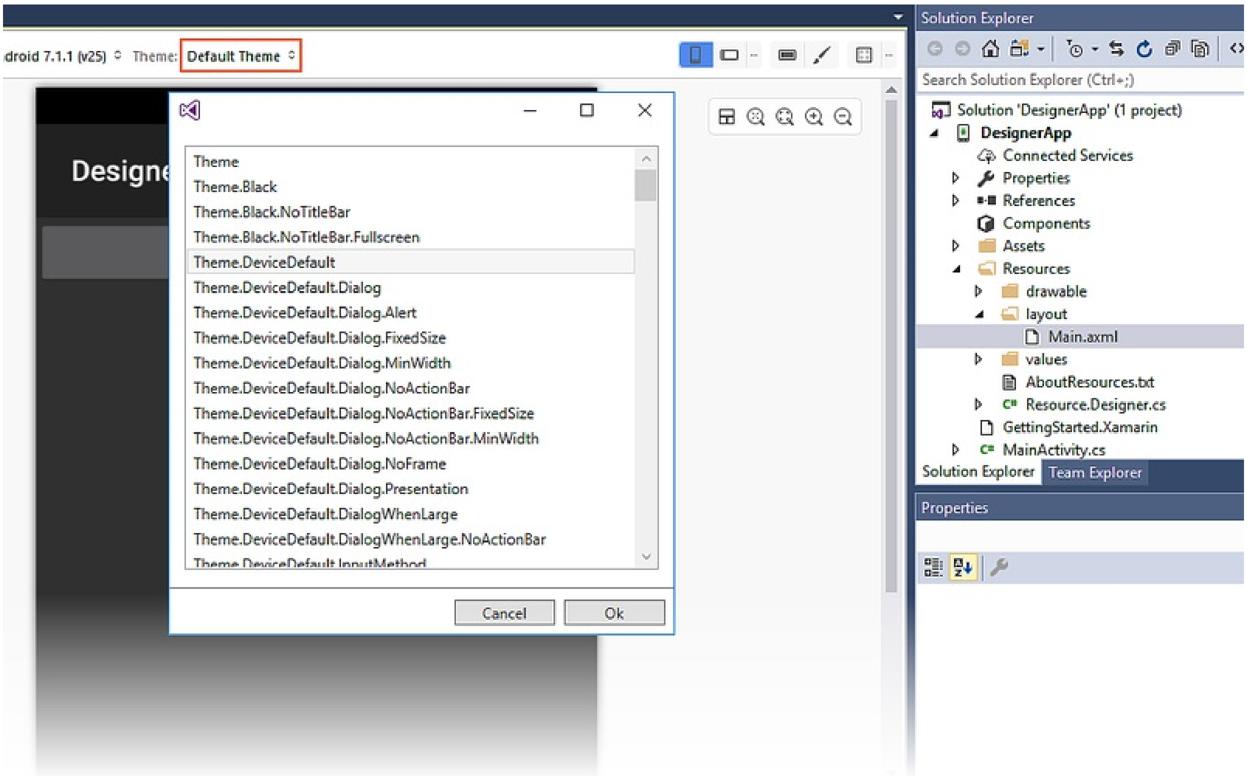


This icon opens a dialog popover that provides a way to select from one of three Action Bar modes:

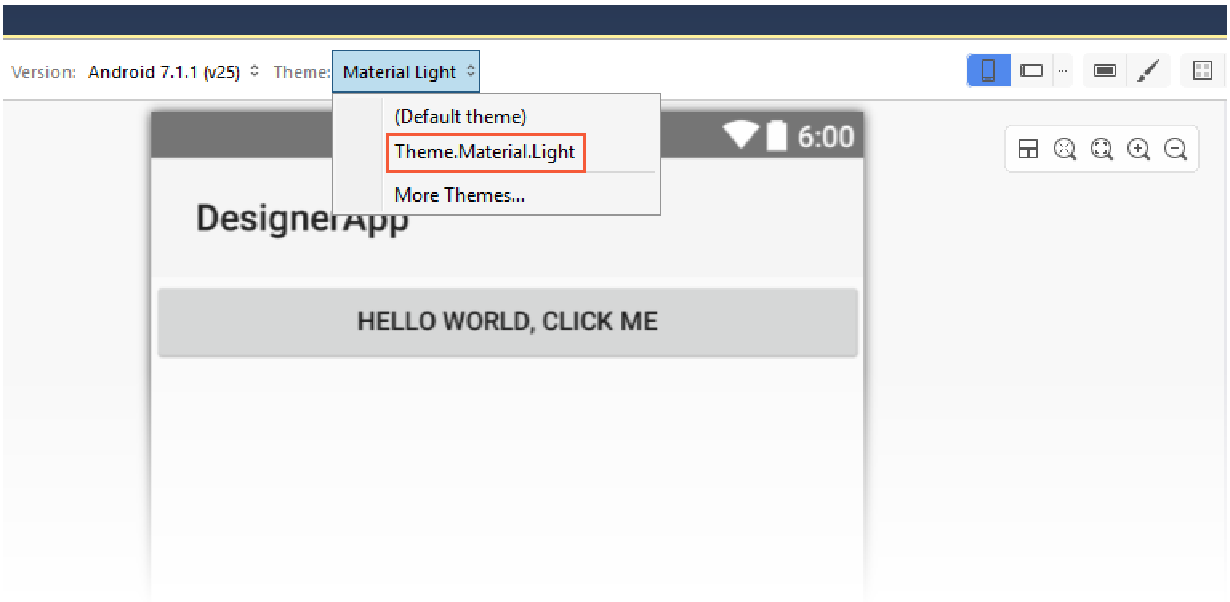
- **Standard** – Consists of either a logo or an icon and title text with an optional subtitle.
- **List** – List navigation mode. Instead of static title text, this mode presents a list menu for navigation within the activity (that is, it can be presented to the user as a dropdown list).
- **Tabs** – Tab navigation mode. Instead of static title text, this mode presents a series of tabs for navigation within the activity.

Themes

The **Theme** drop-down menu displays all of the themes defined in the project. Selecting **More Themes** opens a dialog with a list of all themes available from the installed Android SDK, as shown below:

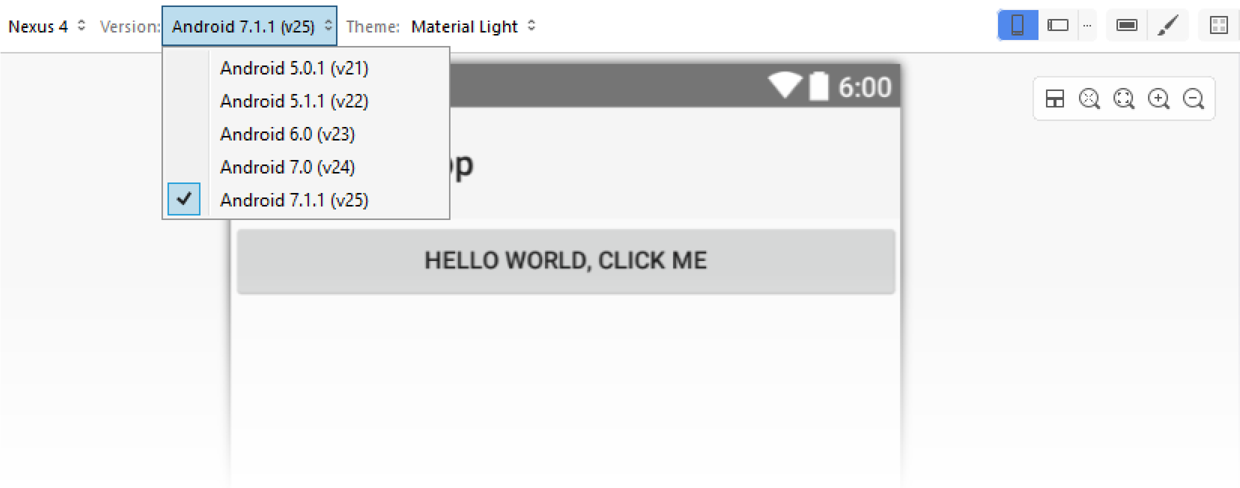


When a theme is selected, the Design Surface is updated to show the effect of the new theme. Note that this change is made permanent only if the **OK** button is clicked in the **Theme** dialog. Once a theme has been selected, it will be included in the **Theme** drop-down menu as shown below:



Android version

The Android **Version** selector sets the Android version that is used to render the layout in the Designer. The selector displays all versions that are compatible with the target framework version of the project:



The target framework version can be set in the project's settings under **Properties** > **Application** > **Compile using Android version**. For more information about target framework version, see Understanding Android API Levels.

The set of widgets available in the toolbox is determined by the target framework version of the project. This is also true for the properties available in the **Properties Window**. The available list of widgets is *not* determined by the value selected in the **Version** selector of the toolbar. For example, if you set the target version of the project to Android 4.4, you can still select Android 6.0 in the toolbar version selector to see what the project looks like in Android 6.0, but you won't be able to add widgets that are specific to Android 6.0 – you will still be limited to the widgets that are available in Android 4.4.

For more information about resource types, see Android Resources.

Alternative layout views

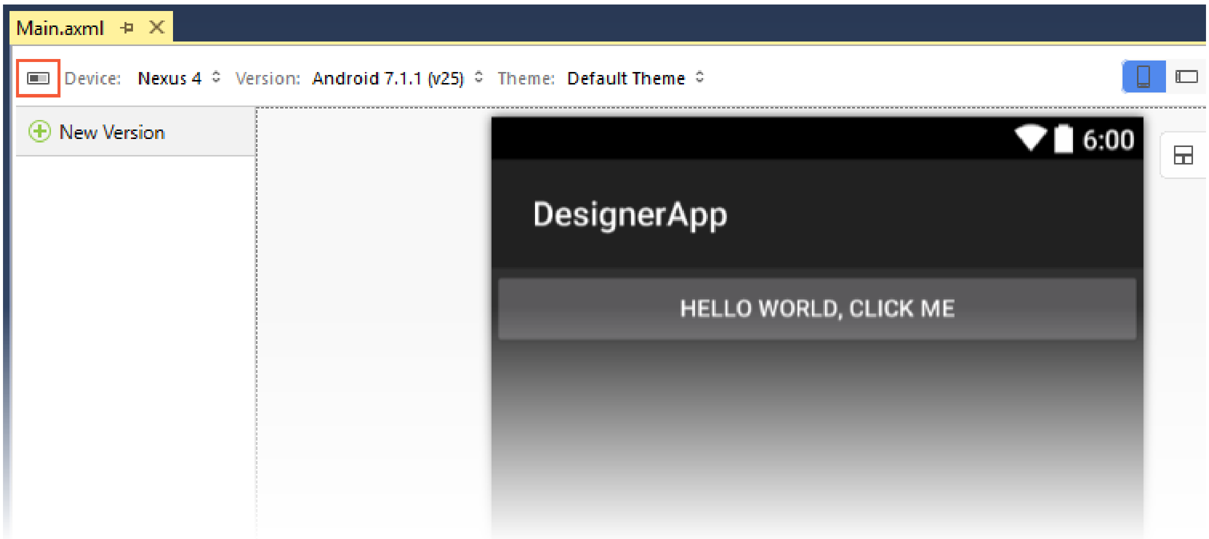
7/8/2021 • 10 minutes to read • Edit Online

This topic explains how you can version layouts by using resource qualifiers. For example, creating a version of a layout that is only used when the device is in landscape mode and a layout version that is only for portrait mode.

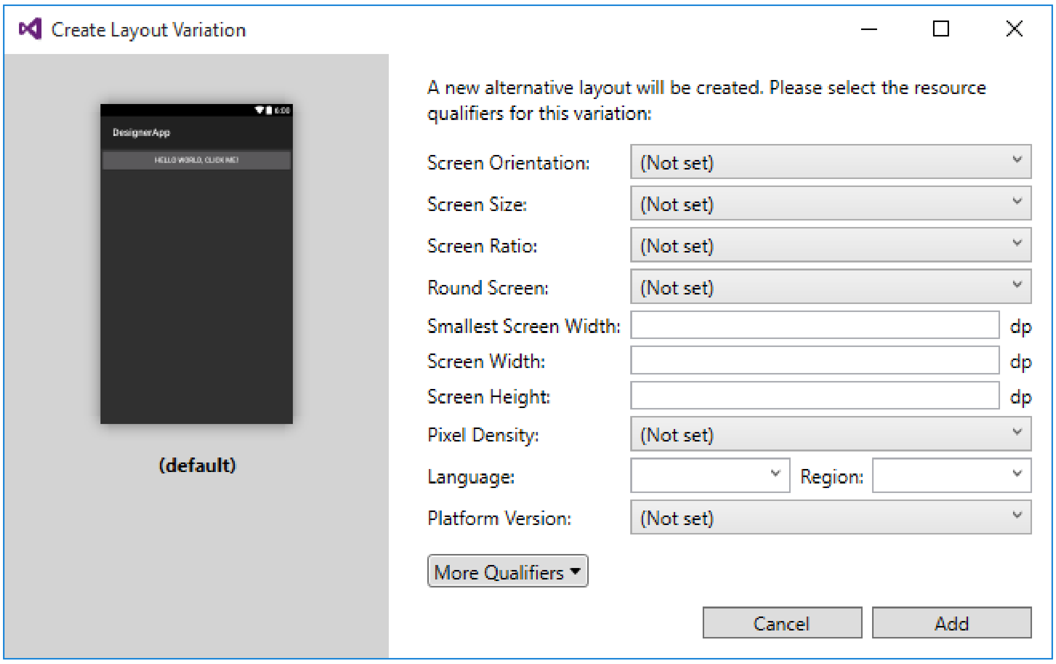
- Visual Studio
- Visual Studio for Mac

Creating alternative layouts

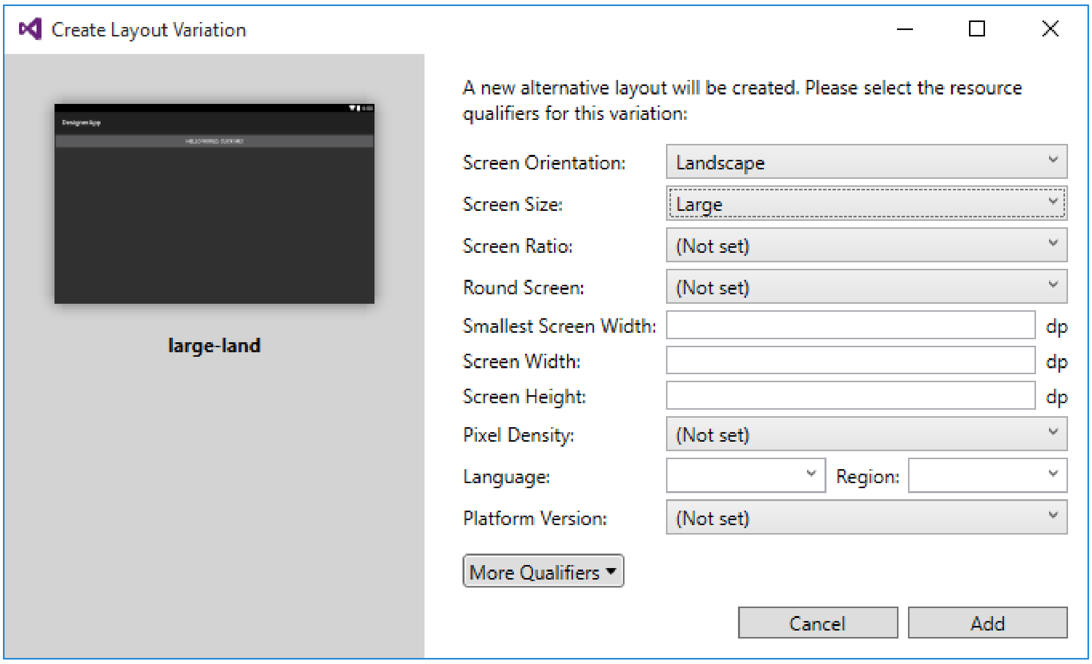
When you click the **Alternative Layout View** icon (to the left of **Device**), a preview pane opens to list the alternative layouts available in your project. If there are no alternative layouts, the **Default** view is presented:



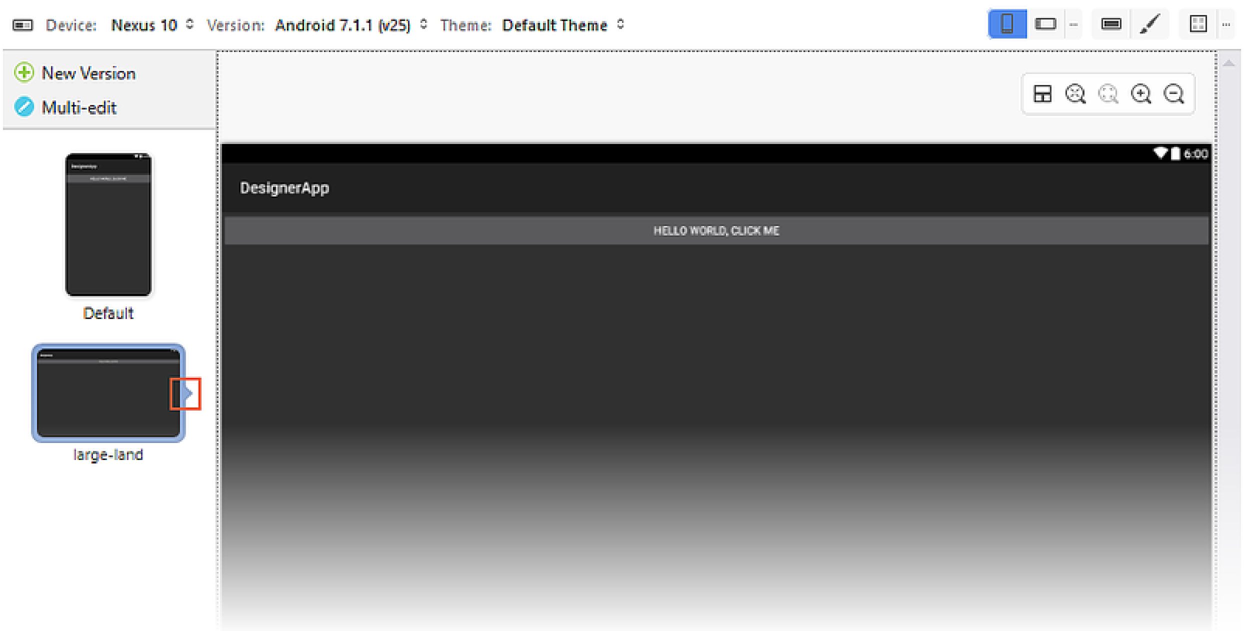
When you click the green plus sign next to **New Version**, the **Create Layout Variation** dialog opens so that you can select the resource qualifiers for this layout variation:



In the following example, the resource qualifier for **Screen Orientation** is set to **Landscape**, and the **Screen Size** is changed to **Large**. This creates a new layout version named **large-land**:



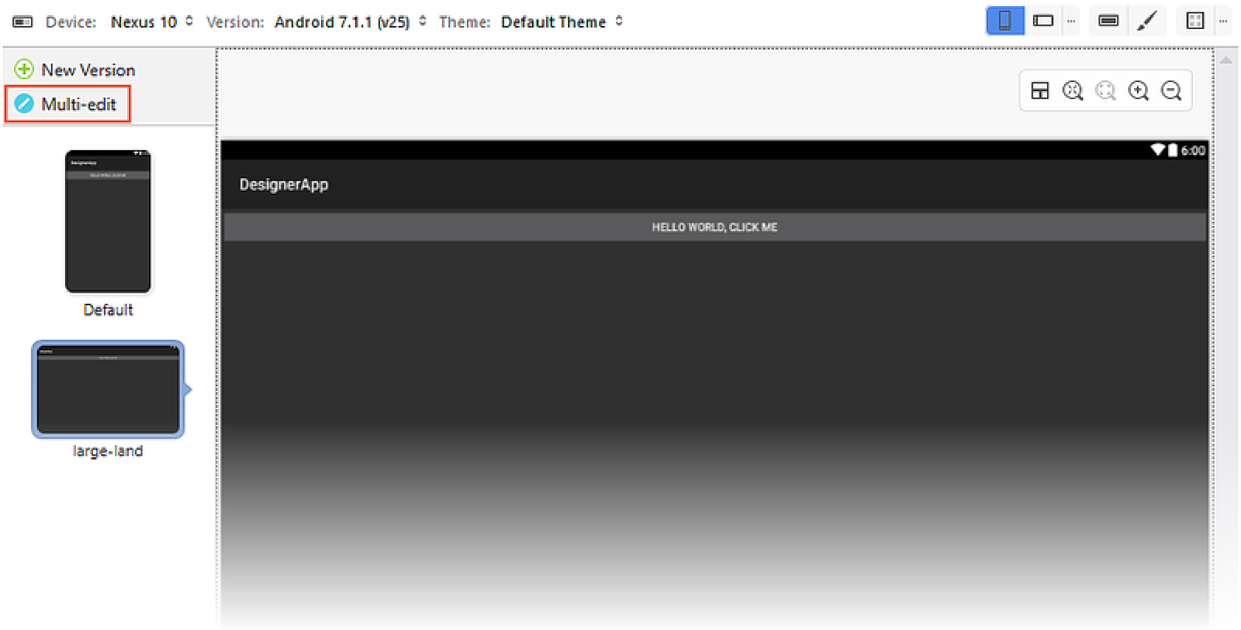
Note that the preview pane on the left displays the effects of the resource qualifier selections. Clicking **Add** creates the alternative layout and switches the Designer to that layout. The **Alternative Layout View** preview pane indicates which layout is loaded into the Designer via a small right pointer as indicated in the following screenshot:



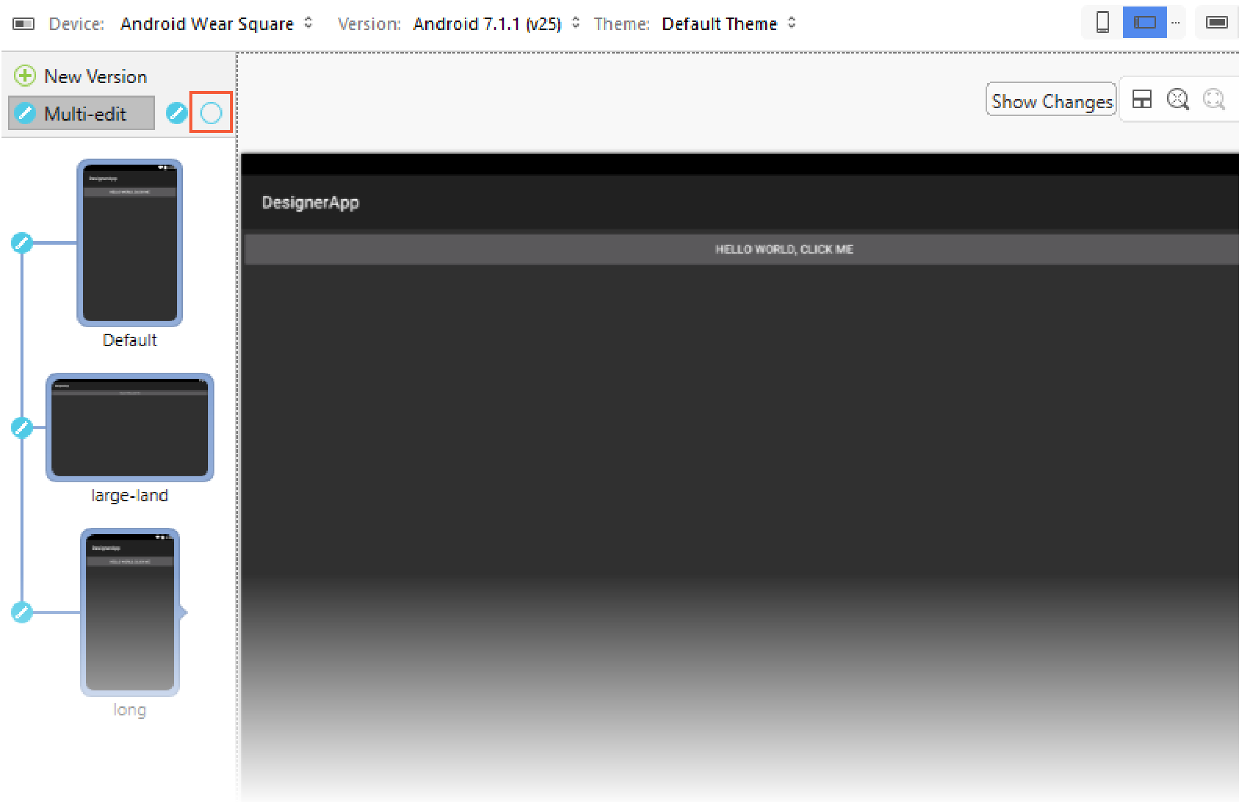
Editing alternative layouts

When you create alternative layouts, it is often desirable to make a single change that applies to all forked versions of a layout. For example, you may want to change the button text to yellow in all layouts. If you have a large number of layouts and you need to propagate a single change to all versions, maintenance can quickly become cumbersome and error-prone.

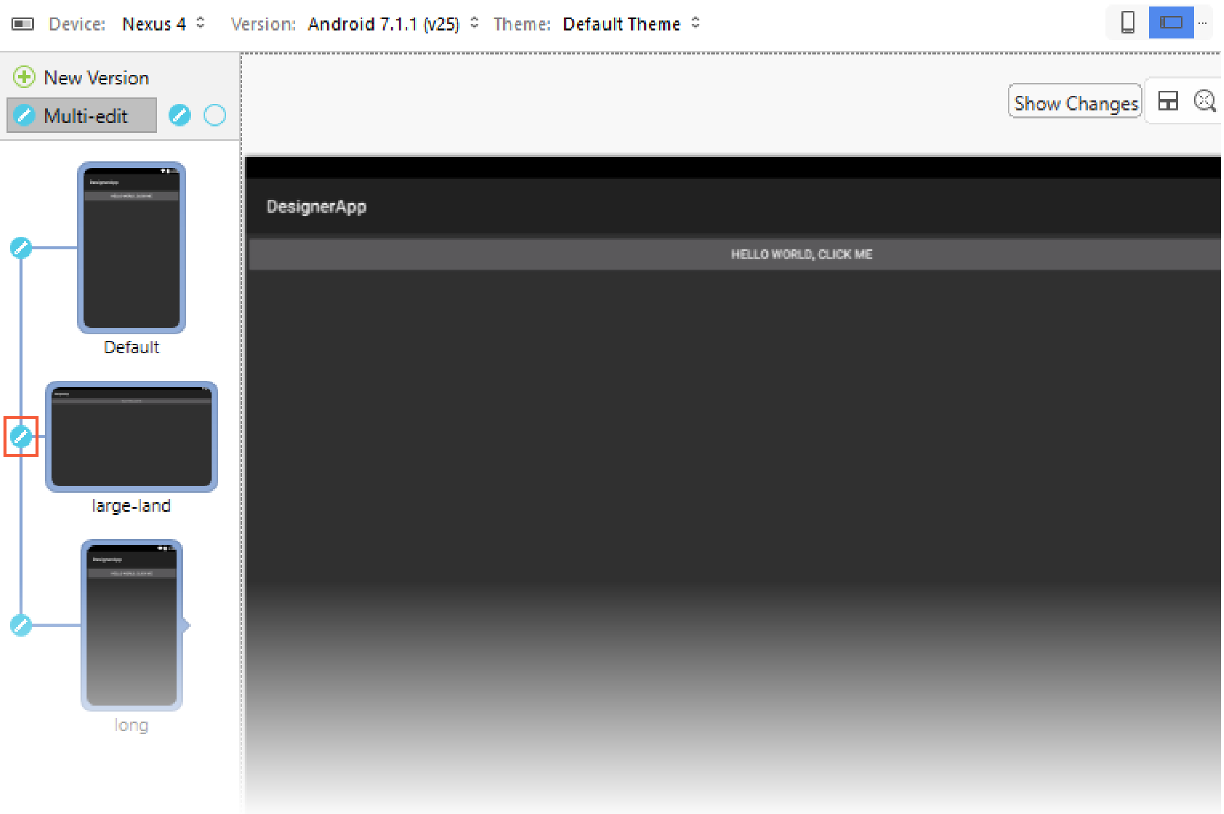
To simplify the maintenance of multiple layout versions, the Designer provides a **Multi-edit** mode that propagates your changes across one or more layouts. When more than one layout is present, the **Multi-edit** icon is displayed:



When you click the **Multi-edit** icon, lines appear that indicate that the layouts are linked (as shown below); that is, when you make a change to one layout, that change is propagated to any linked layouts. You can unlink all layouts by clicking the circled icon indicated in the following screenshot:



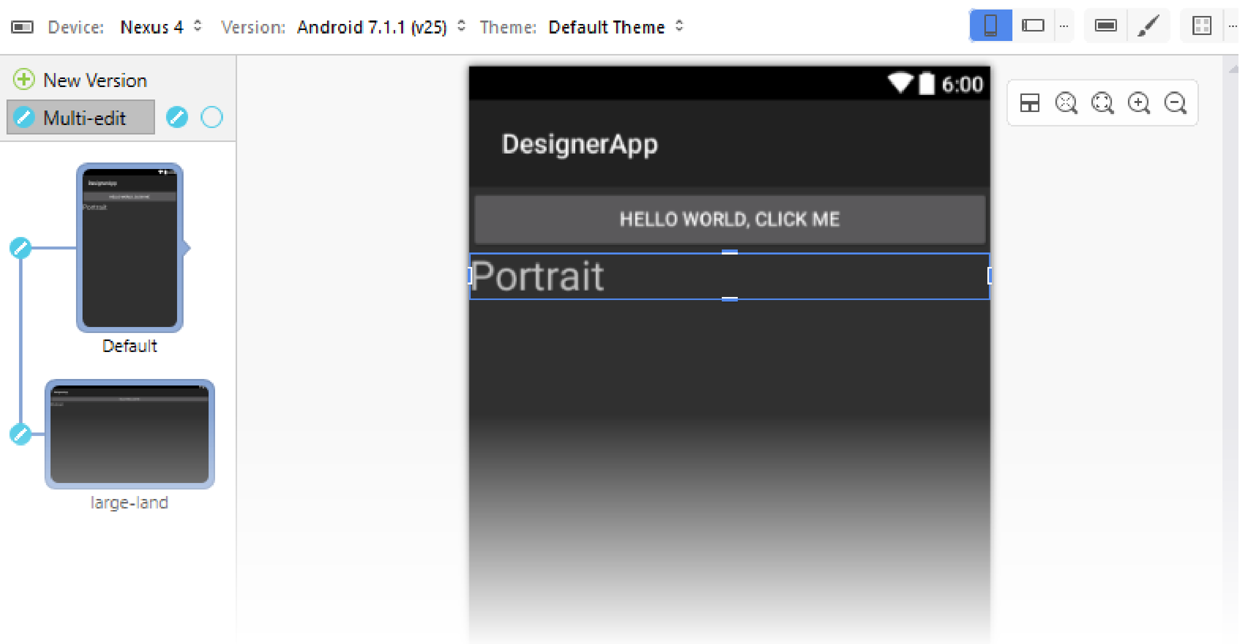
If you have more than two layouts, you can selectively toggle the edit button to the left of each layout preview to determine which layouts are linked together. For example, if you want to make a single change that propagates to the first and last of three layouts, you would first unlink the middle layout as shown here:



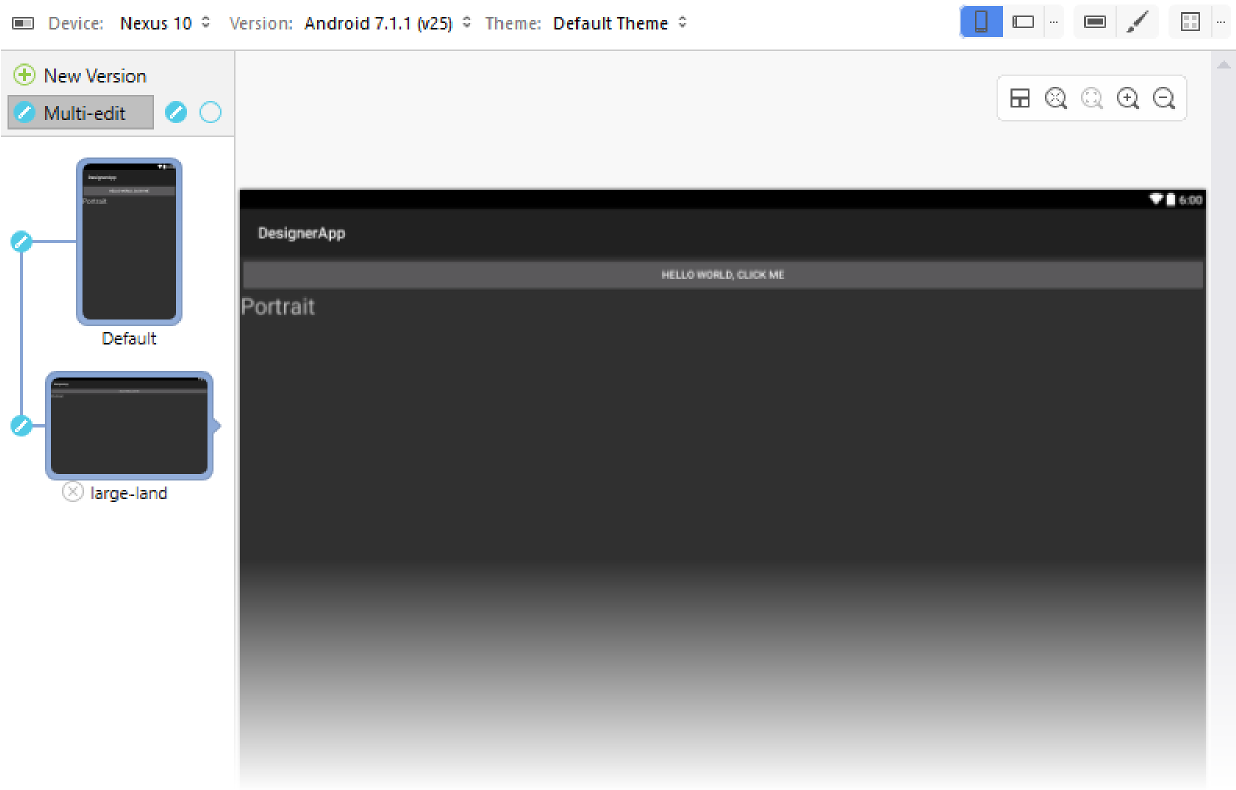
In this example, a change made to either the **Default** or **long** layout will be propagated to the other layout but not to the **large-land** layout.

Multi-Edit example

In general, when you make a change to one layout, that same change is propagated to all other linked layouts. For example, adding a new `TextView` widget to the **Default** layout and changing its text string to `Portrait` will cause the same change to be made to all linked layouts. Here is how it looks in the **Default** layout:



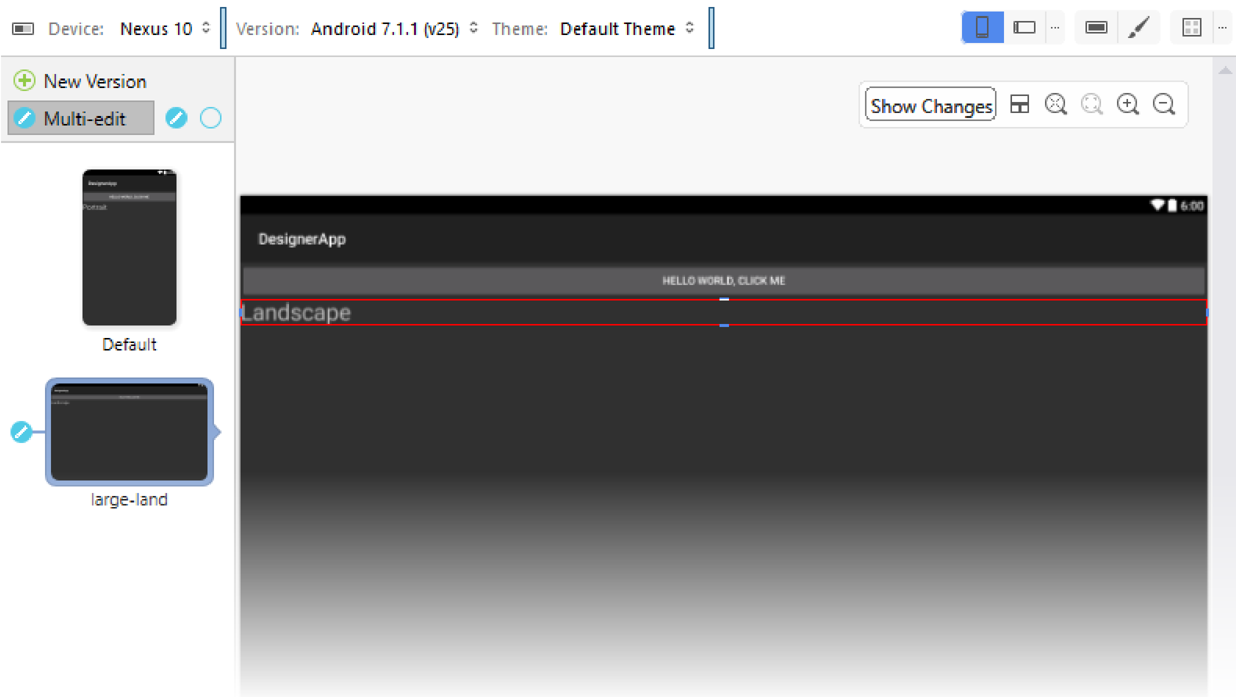
The `TextView` is also added to the **large-land** layout view because it is linked to the **Default** layout:



But what if you want to make a change that is local to only one layout (that is, you don't want the change to be propagated to any of the other layouts)? To do this, you must unlink the layout that you want to change before you modify it, as explained next.

Making local changes

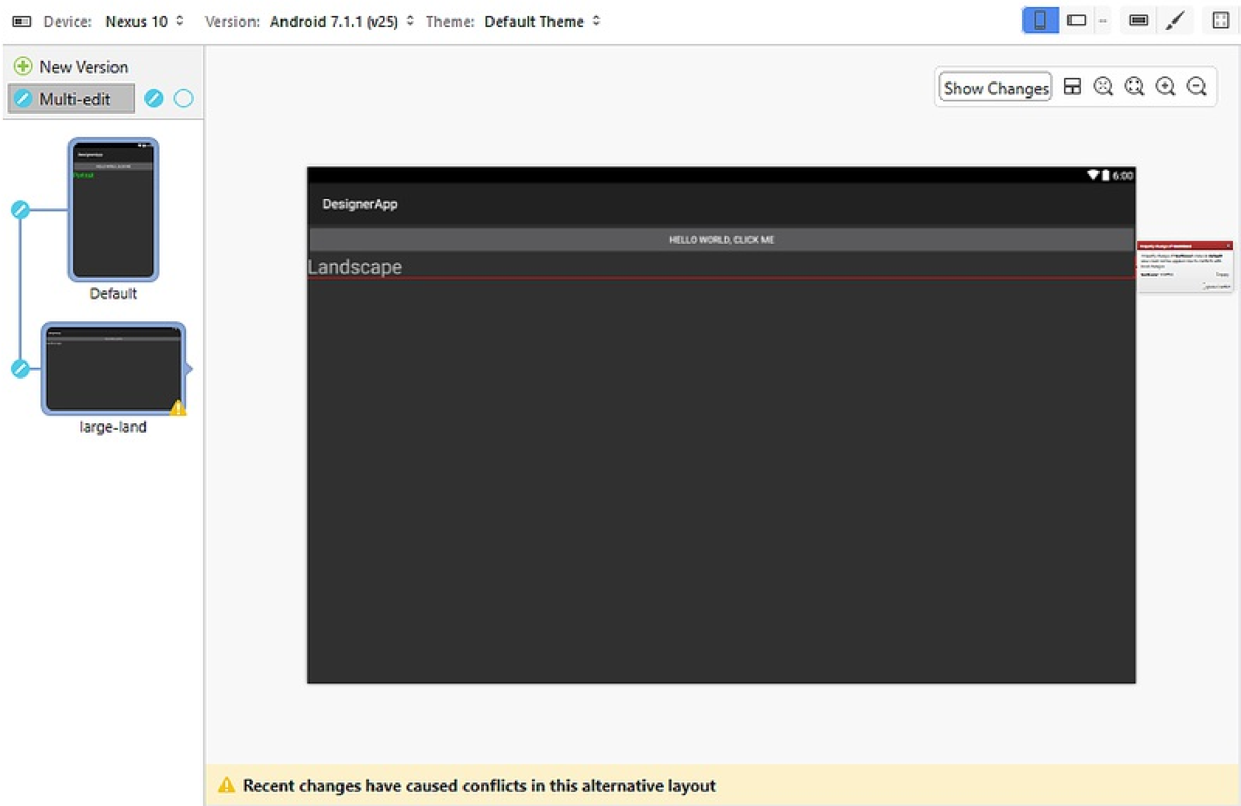
Suppose we want both layouts to have the added `TextView`, but we also want to change the text string in the **large-land** layout to `Landscape` rather than `Portrait`. If we make this change to **large-land** while both layouts are linked, the change will propagate back to the **Default** layout. Therefore, we must first unlink the two layouts before we make the change. When we modify the text in **large-land** to `Landscape`, the Designer marks this change with a red frame to indicate that the change is local to the **large-land** layout and is *not* propagated back to the **Default** layout:



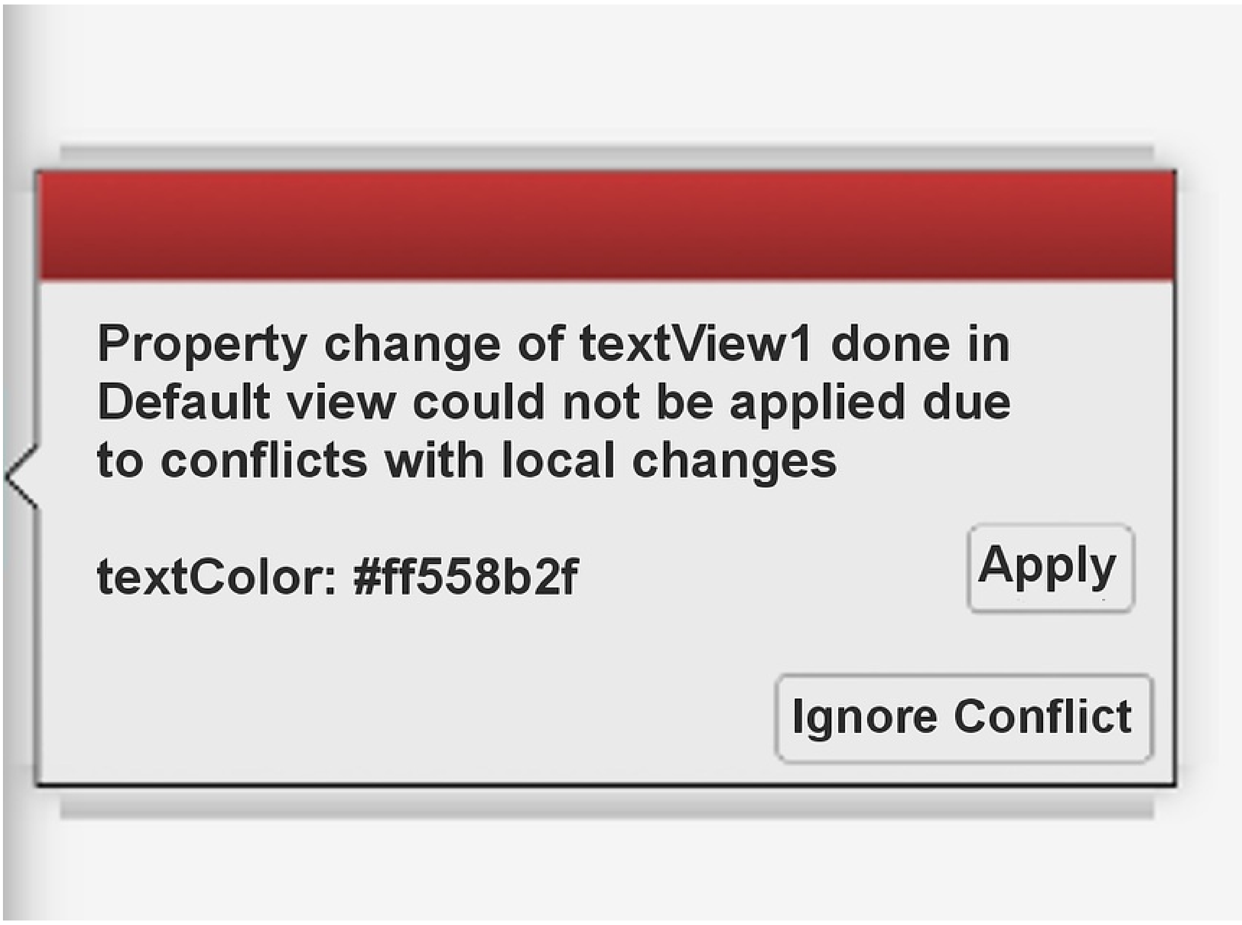
When you click the **Default** layout to view it, the `TextView` text string is still set to `Portrait`.

Handling conflicts

If you decide to change the color of the text in the **Default** layout to green, you'll see a warning icon appear on the linked layout. Clicking that layout opens the layout to reveal the conflict. The widget that caused the conflict is highlighted with a red frame and the following message is displayed: *Recent changes have caused conflicts in this alternative layout*.



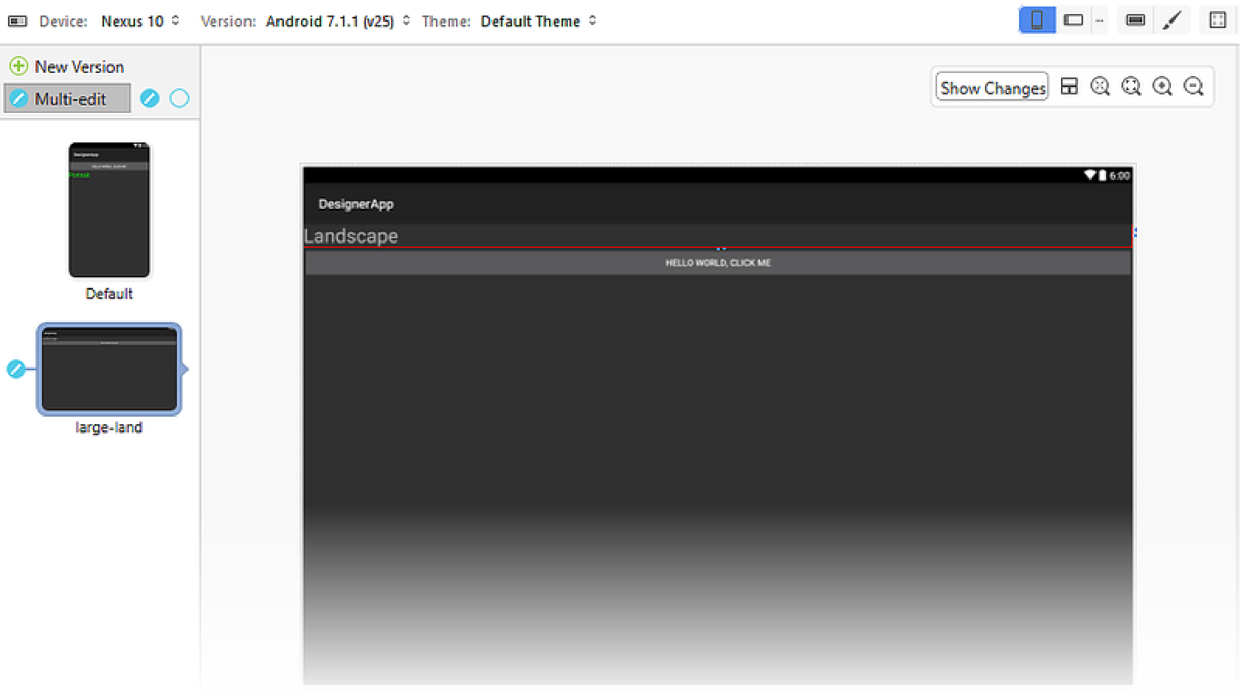
A *conflict box* is displayed on the right of the widget to explain the conflict:



The conflict box shows the list of properties that have changed and it lists their values. Clicking **Ignore Conflict** applies the property change only to this widget. Clicking **Apply** applies the property change to this widget as well as to the counterpart widget in the linked **Default** layout. If all property changes are applied, the conflict is automatically discarded.

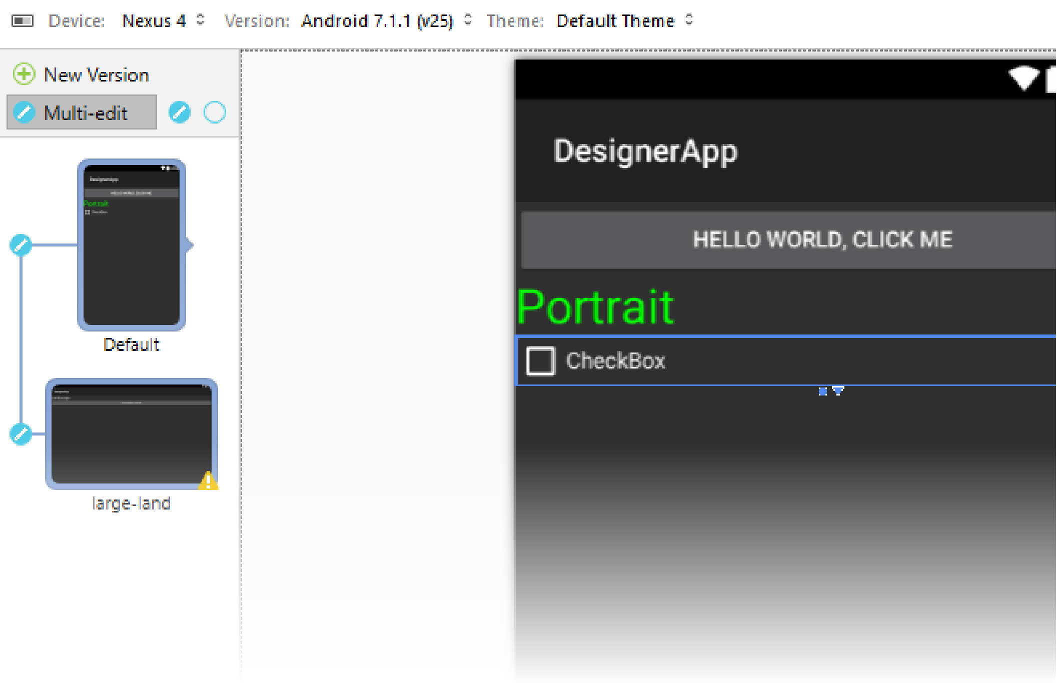
View group conflicts

Property changes are not the only source of conflicts. Conflicts can be detected when inserting or removing widgets. For example, when the **large-land** layout is unlinked from the **Default** layout, and the `TextView` in the **large-land** layout is dragged and dropped above the `Button`, the Designer marks the moved widget to indicate the conflict:

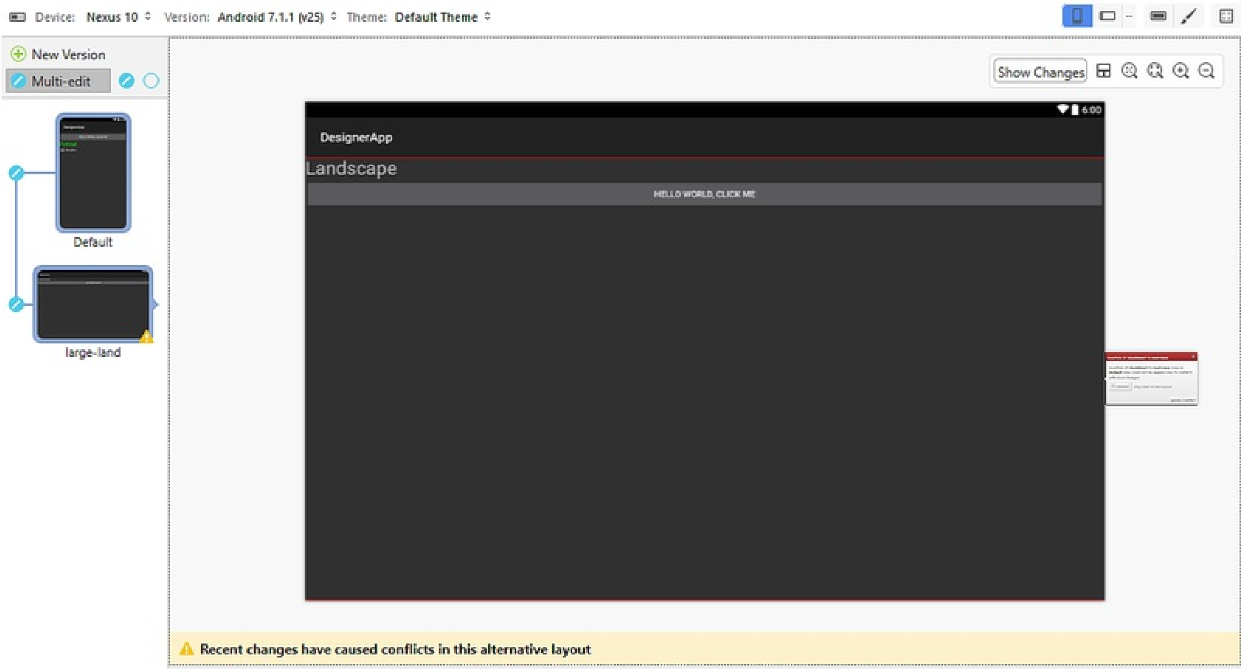


However, there is no marker on the `Button`. Although the position of the `Button` has changed, the `Button` shows no applied changes that are specific to the **large-land** configuration.

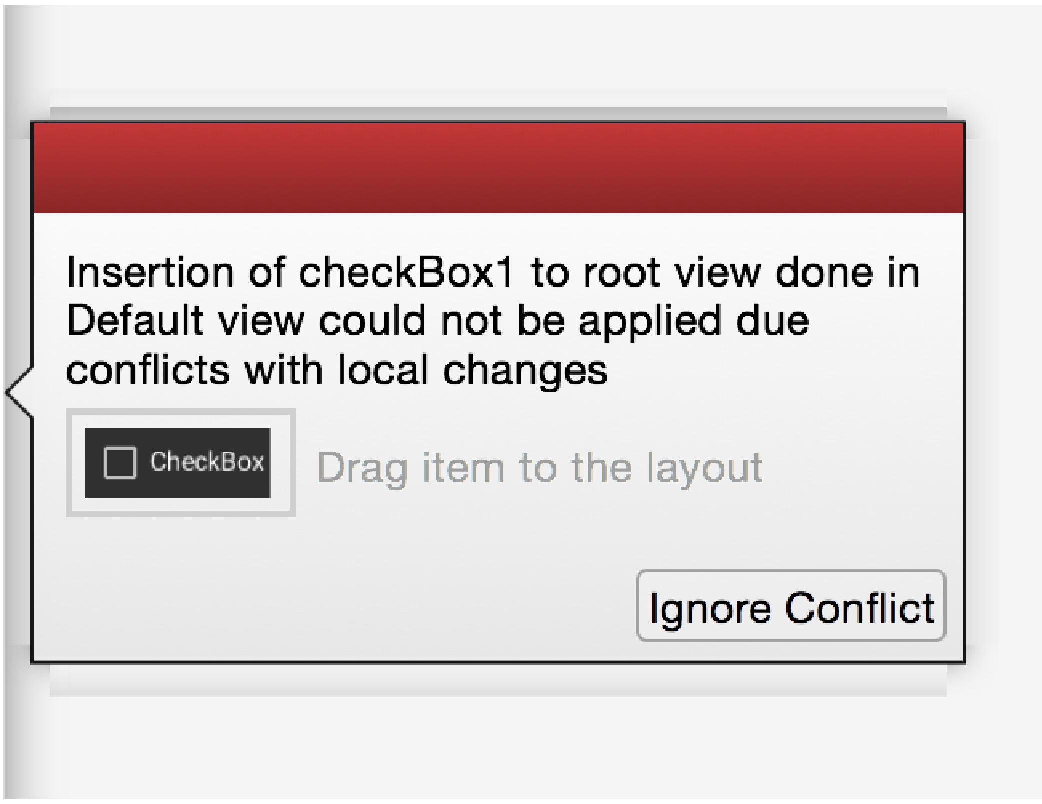
If a `CheckBox` is added to the **Default** layout, another conflict is generated, and a warning icon is displayed over the **large-land** layout:



Clicking the **large-land** layout reveals the conflict. The following message is displayed: *Recent changes have caused conflicts in this alternative layout*.

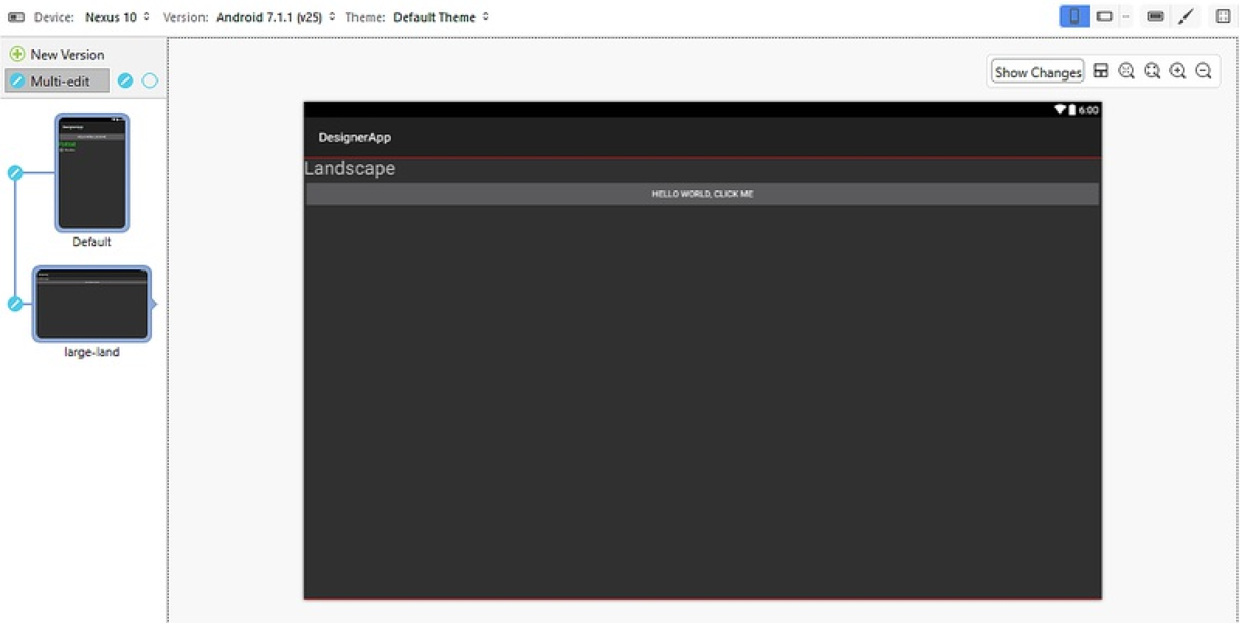


In addition, the conflict box displays the following message:



Adding the `CheckBox` causes a conflict because the **large-land** layout has changes in the `LinearLayout` that contains it. However, in this case the conflict box displays the widget that was just inserted into the **Default** layout (the `CheckBox`).

If you click **Ignore Conflict**, the Designer resolves the conflict, allowing the widget displayed in the conflict box to be dragged and dropped into the layout where the widget is missing (in this case, the **large-land** layout):



As seen in the previous example with the `Button`, the `CheckBox` does not have a red change marker because only the `LinearLayout` has changes that were applied in the **large-land** layout.

Conflict persistence

Conflicts are persisted in the layout file as XML comments, as shown here:

```
<!-- Widget Inserted Conflict | id:__root__ | @+id/checkBox1 -->
```

Therefore, when a project is closed and reopened, all the conflicts will still be there – even the ones that have been ignored.

Xamarin.Android Designer Material Design features

7/8/2021 • 11 minutes to read • Edit Online

This topic describes Designer features that make it easier for developers to create Material Design-compliant layouts. This section introduces and explains how to use the Material Grid, the Material Color Palette, the Typographic Scale, and the Theme Editor.

Evolve 2016: Everyone Can Create Beautiful Apps with Material Design

Overview

The Xamarin.Android Designer includes features that make it easier for you to create Material-Design-compliant layouts. If you are not familiar with Material Design, see the Material Design introduction.

- Visual Studio
- Visual Studio for Mac

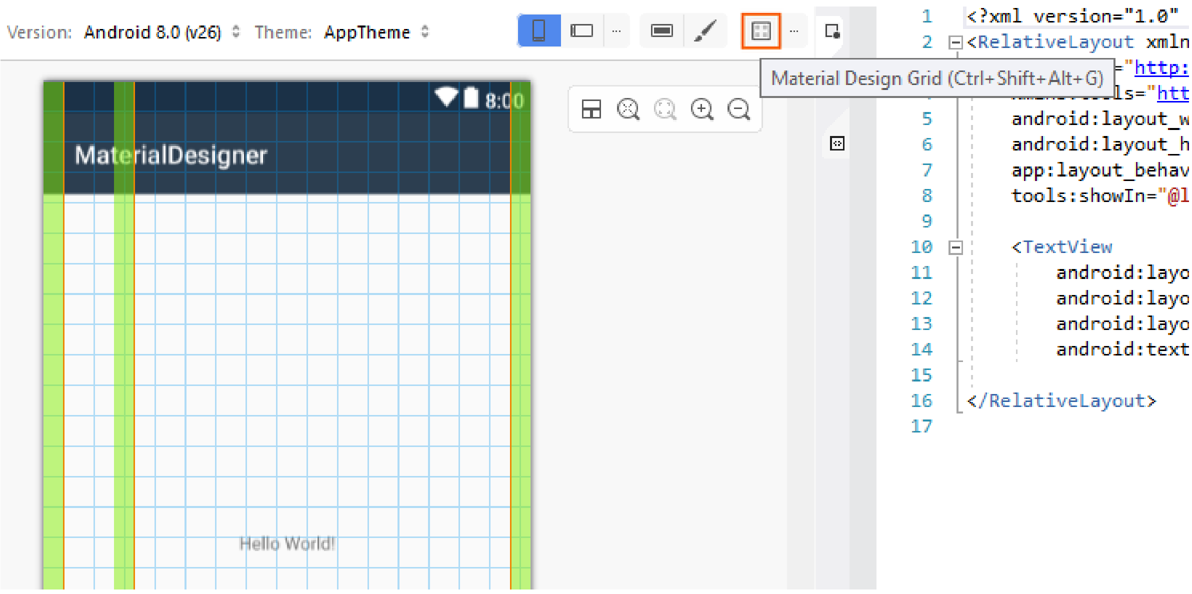
In this guide, we'll have a look at the following Designer features:

- *Material Grid* – An overlay on the Design Surface that shows a grid, spacing, and keylines to help you place layout widgets according to Material Design guidelines.
- *Theme Editor* – A small color resource editor that lets you set color information for a subset of a theme. For example, you can preview and modify Material colors such as `colorPrimary`, `colorPrimaryDark`, and `colorAccent`.

We'll have look at each of these features and provide examples of how to use them.

Material Design Grid

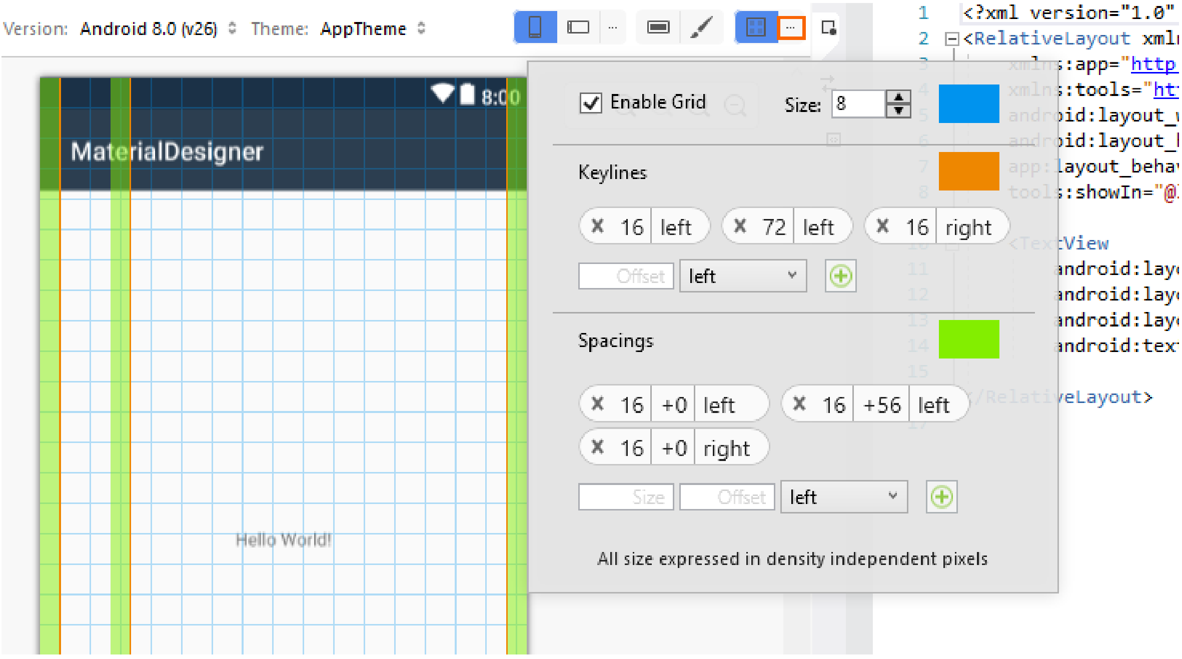
The Material Design Grid menu is available from the toolbar at the top of the Designer:



When you click the Material Design Grid icon, the Designer displays an overlay on the Design Surface that includes the following elements:

- Keylines (orange lines)
- Spacing (green areas)
- A grid (blue lines)

These elements can be seen in the previous screenshot. Each of these overlay items is configurable. When you click the ellipsis next to the Material Design Grid menu, a dialog popover opens that allows you to disable/enable the grid, configure the placement of keylines, and set spacings. Note that all values are expressed in `dp` (density-independent pixels):

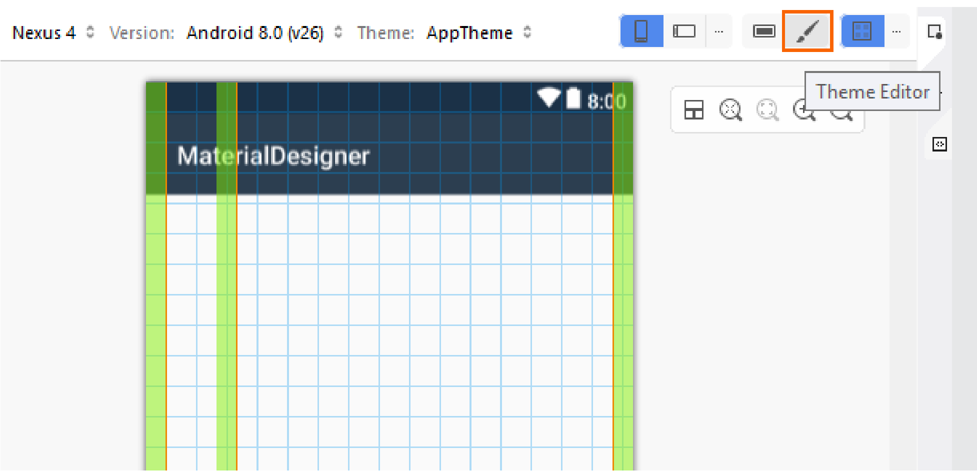


To add a new keyline, enter a new offset value in the **Offset** box, select a location (**left**, **top**, **right**, or **bottom**) and click the + icon to add the new keyline. Similarly, to add a new spacing, enter the size and offset (in dp) into the **Size** and **Offset** boxes, respectively. Select a location (**left**, **top**, **right**, or **bottom**) and click the + icon to add the new spacing.

When you change these configuration values, they are saved in the layout XML file and reused when you open the layout again.

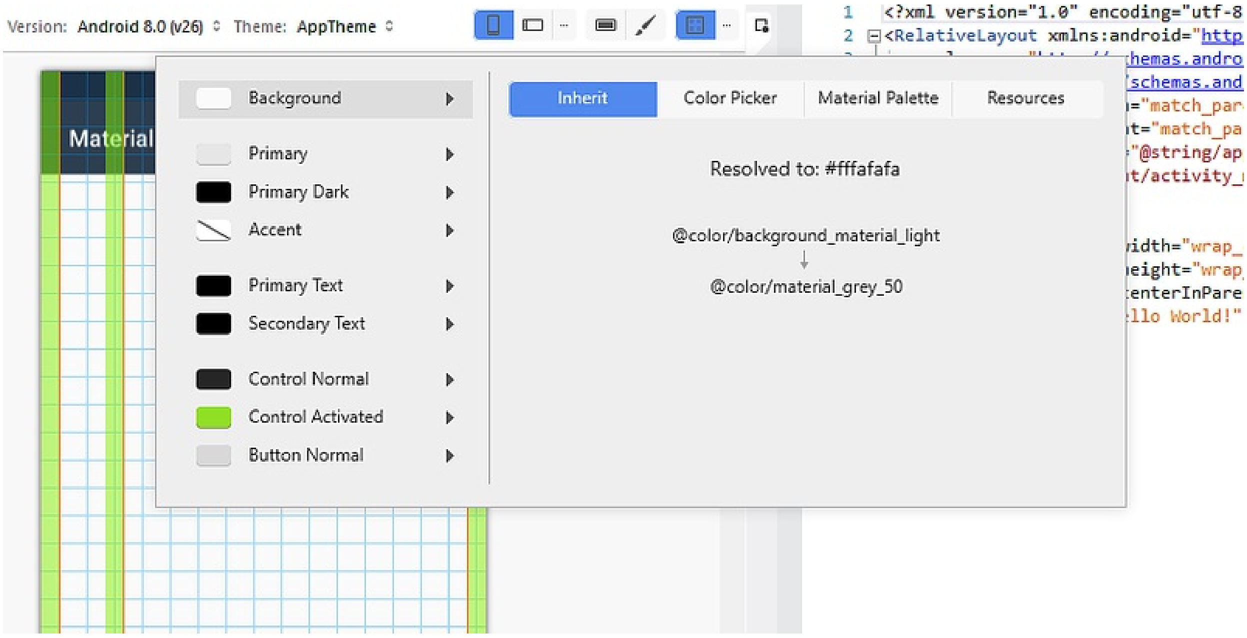
Theme Editor

The **Theme Editor** lets you customize color information for a subset of theme attributes. To open the **Theme Editor**, click the paintbrush icon on the toolbar:



Although the **Theme Editor** is accessible from the toolbar for all target Android versions and API levels, only a subset of the capabilities described below are available if the target API level is earlier than API 21 (Android 5.0 Lollipop).

The left-hand panel of the **Theme Editor** displays the list of colors that make up the currently selected theme (in this example, we are using the `Default Theme`):



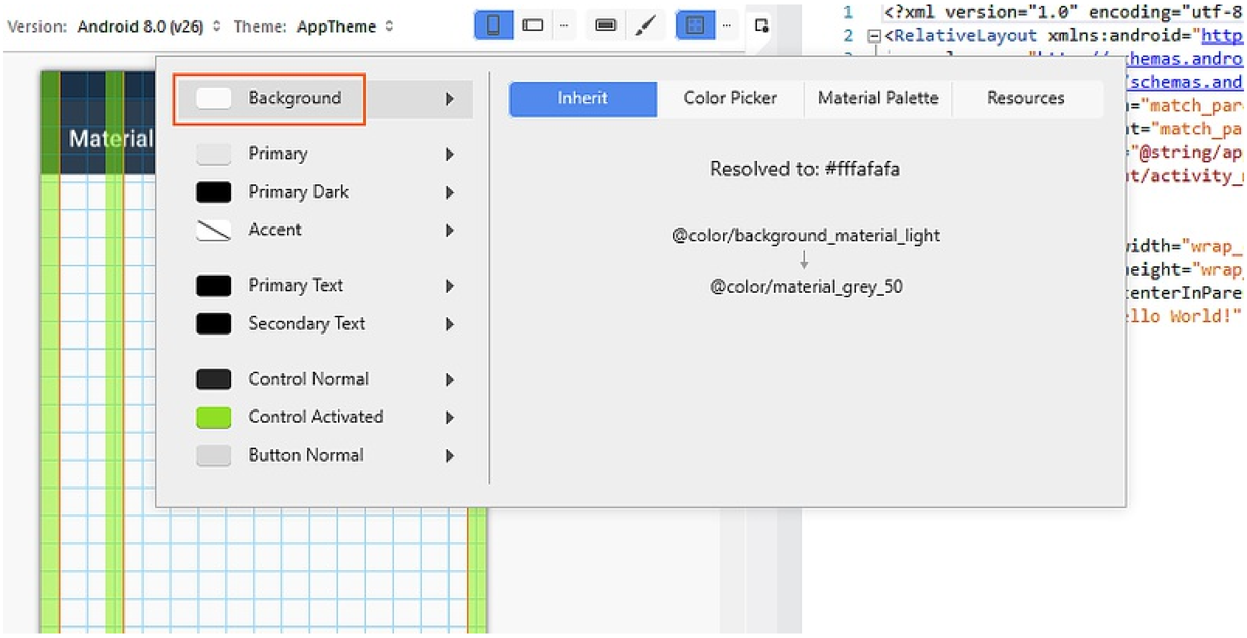
When you select a color on the left, the right-hand panel provides the following tabs to help you edit that color:

- **Inherit** – Displays a style inheritance diagram for the selected color and lists the resolved color and color code assigned to that theme color.
- **Color Picker** – Lets you change the selected color to any arbitrary value.
- **Material Palette** – Lets you change the selected color to a value that conforms to Material Design.
- **Resources** – Lets you change the selected color to one of the other existing color resources in the theme.

Let's look at each one of these tabs in detail.

Inherit tab

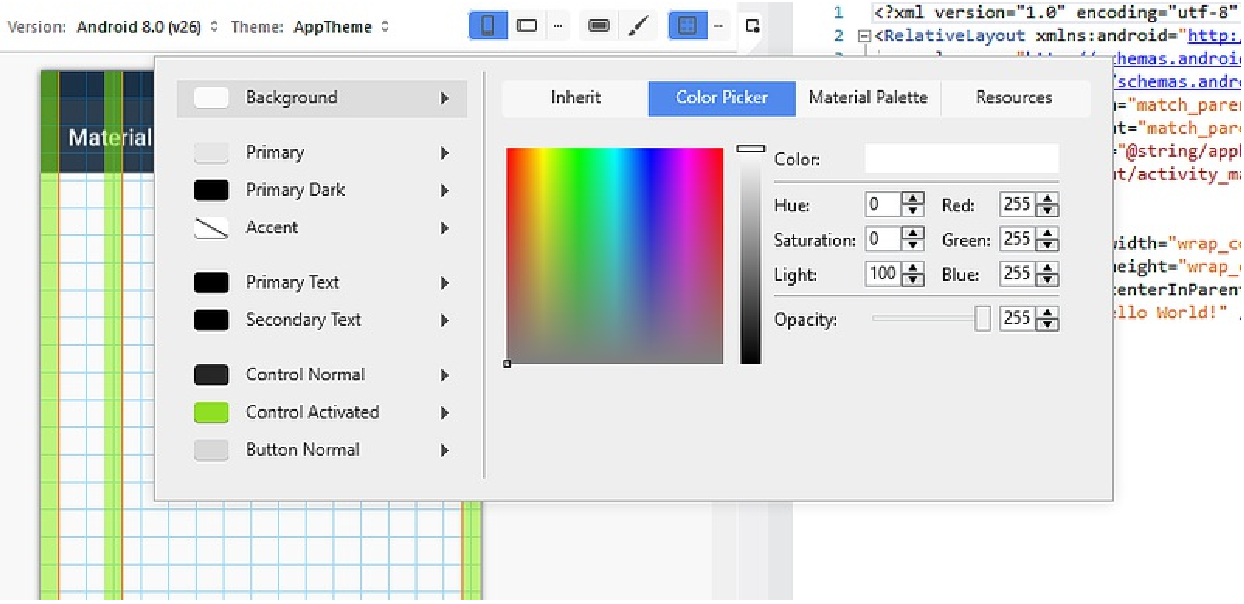
As seen in the following example, the **Inherit** tab lists the style inheritance for the **Background** color of the **Default Theme**:



In this example, the **Default Theme** inherits from a style that uses `@color/background_material_light` but overrides it with `color/material_grey_50`, which has a color code value of `#fffafafa`. For more information about style inheritance, see Styles and Themes.

Color Picker

The following screenshot illustrates the **Color Picker**:



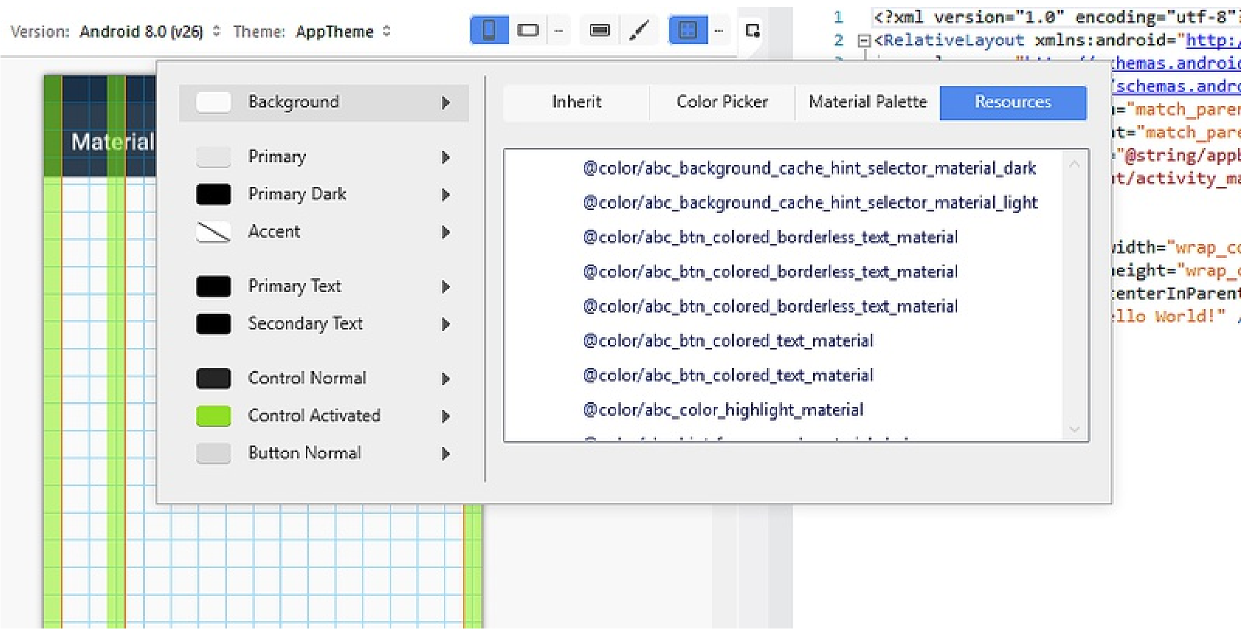
In this example, the **Background** color can be changed to any value through various means:

- Clicking a color directly.
- Entering hue, saturation, and brightness values.
- Entering RGB (red, green, blue) values in decimal.
- Setting the alpha (opacity) for the selected color.
- Entering the hexadecimal color code directly.

The color you choose in the Color Picker is *not* restricted to Material Design guidelines or to the set of available color resources.

Resources

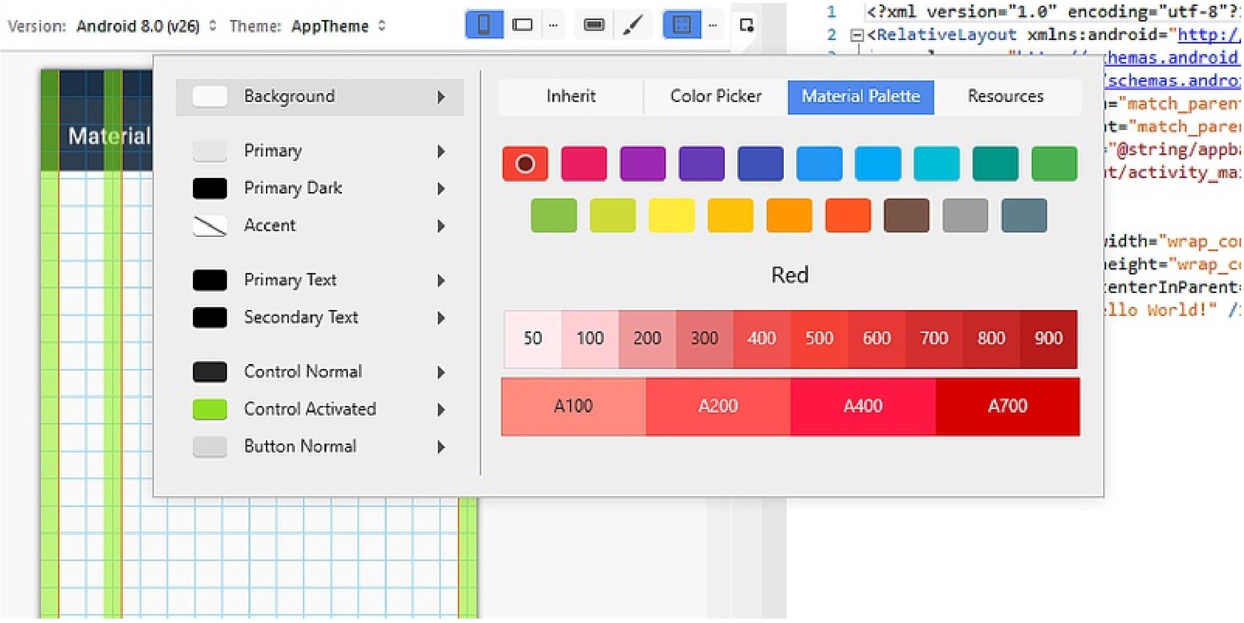
The **Resources** tab offers a list of color resources that are already present in the theme:



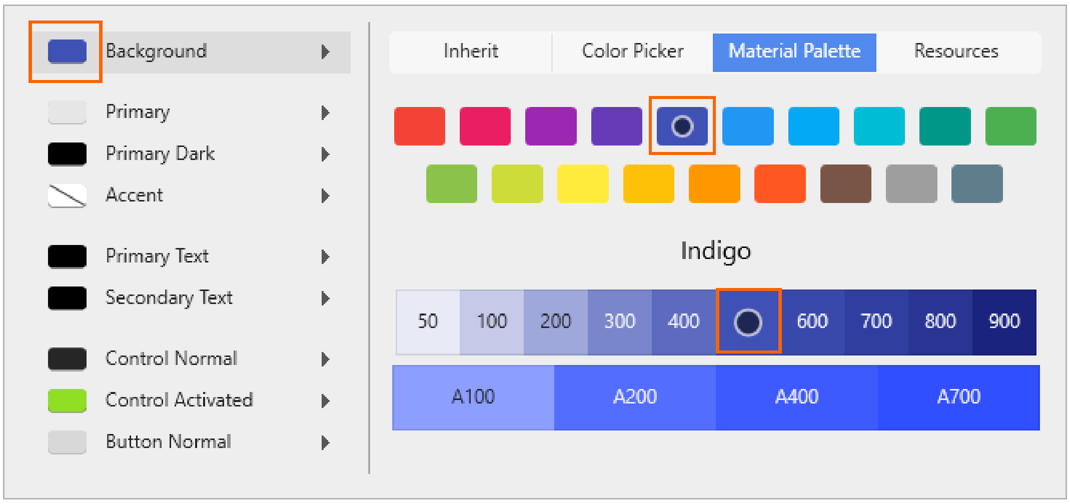
Using the **Resources** tab constrains your choices to this list of colors. Keep in mind that if you choose a color resource that is already assigned to another part of the theme, two adjacent elements of the UI may "run together" (because they have the same color) and become difficult for the user to distinguish.

Material Palette

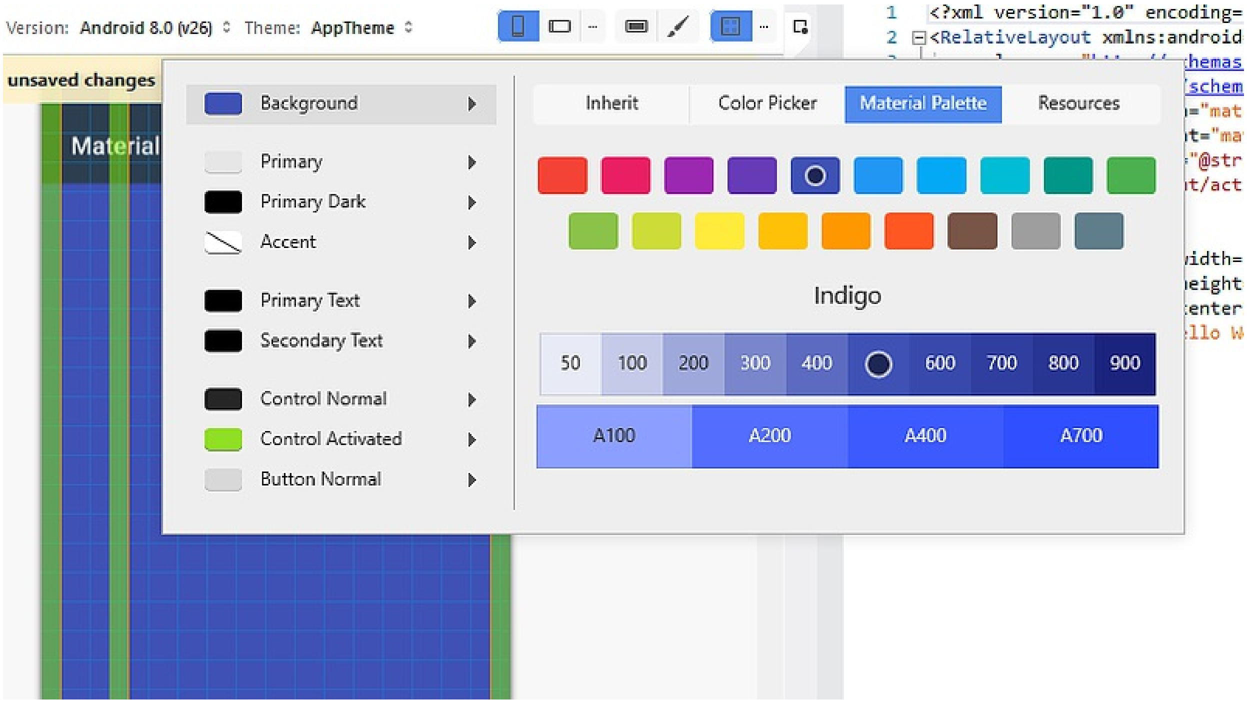
The **Material Palette** tab opens the **Material Design Color Palette**. Choosing a color value from this palette constrains your color choice so that it is consistent with Material Design guidelines:



The top of the color palette displays primary Material Design colors while the bottom of the palette displays a range of hues for the selected primary color. For example, when you select **Indigo**, a collection of **Indigo** hues is displayed at the bottom of the dialog. When you select a hue, the color of the property is changed to the selected hue. In the following example, the `Background Tint` of the button is changed to *Indigo 500*:



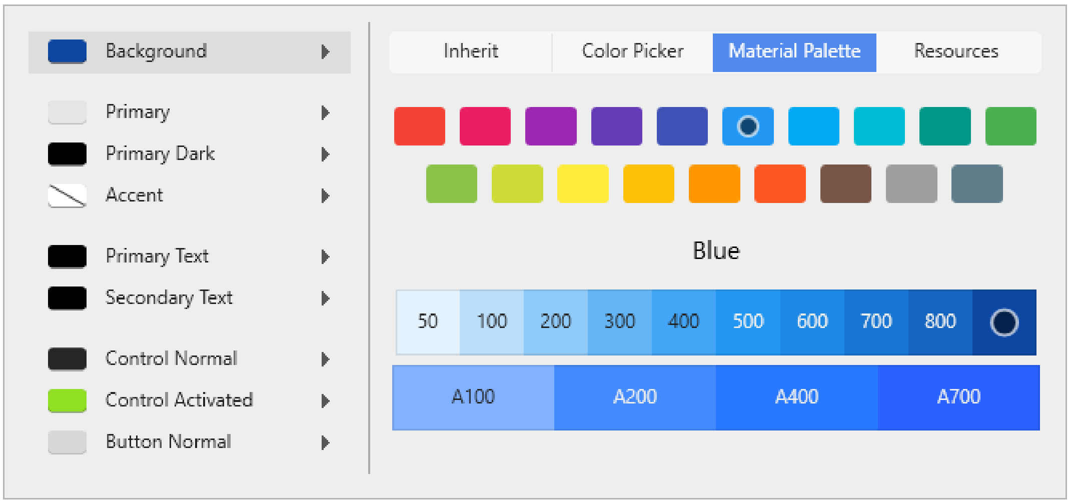
`Background Tint` is set to the color code for *Indigo 500* (`#ff3f51b5`), and the Designer updates the background color to reflect this change:



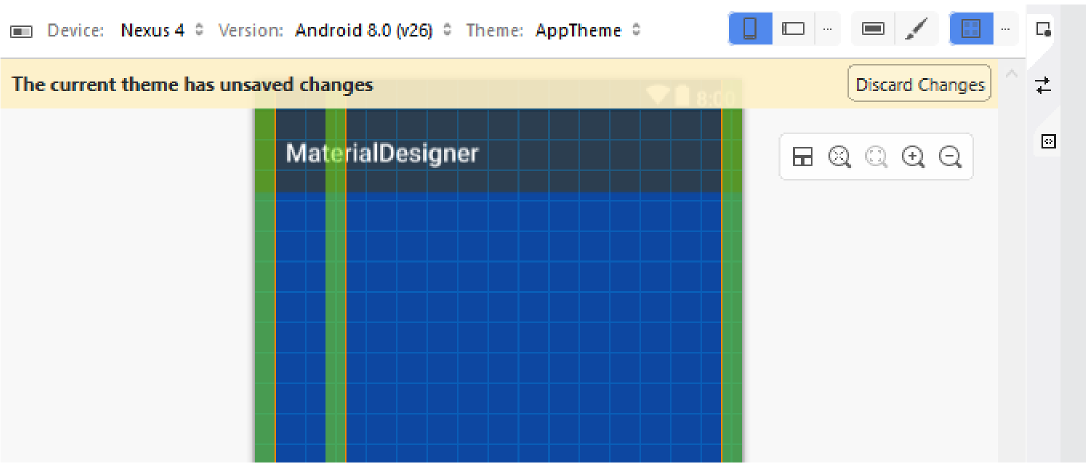
For more information about the Material Design color palette, see the Material Design Color Palette Guide.

Creating a new theme

In the following example, we'll use the Material Palette to create a new custom theme. First, we'll change the **Background** color to *Blue 900*:



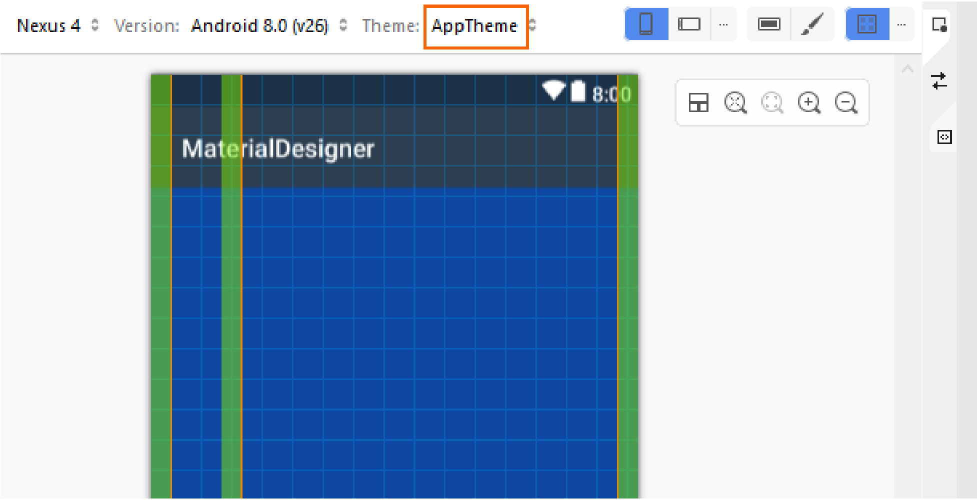
When a color resource is changed, a message pops up with the message, *The current theme has unsaved changes*:



The **Background** color in the Designer has changed to the new color selection, but this change has not yet been saved. At this point, you can do one of the following:

- Click **Discard Changes** to discard the new color choice (or choices) and revert the theme to its original state.
- Press CTRL+S to save your changes to the currently theme.

In the following example, CTRL+S was pressed so that the changes were saved to **AppTheme**:



Summary

This topic described the Material Design features available in the Xamarin.Android Designer. It explained how to enable and configure the Material Design Grid, and it explained how to use the Theme Editor to create new custom themes that conform to Material Design guidelines. For more information about Xamarin.Android support for Material Design, see Material Theme.

Related Links

- Material Theme
- Material Design introduction

Android layout diagnostics

7/8/2021 • 3 minutes to read • Edit Online

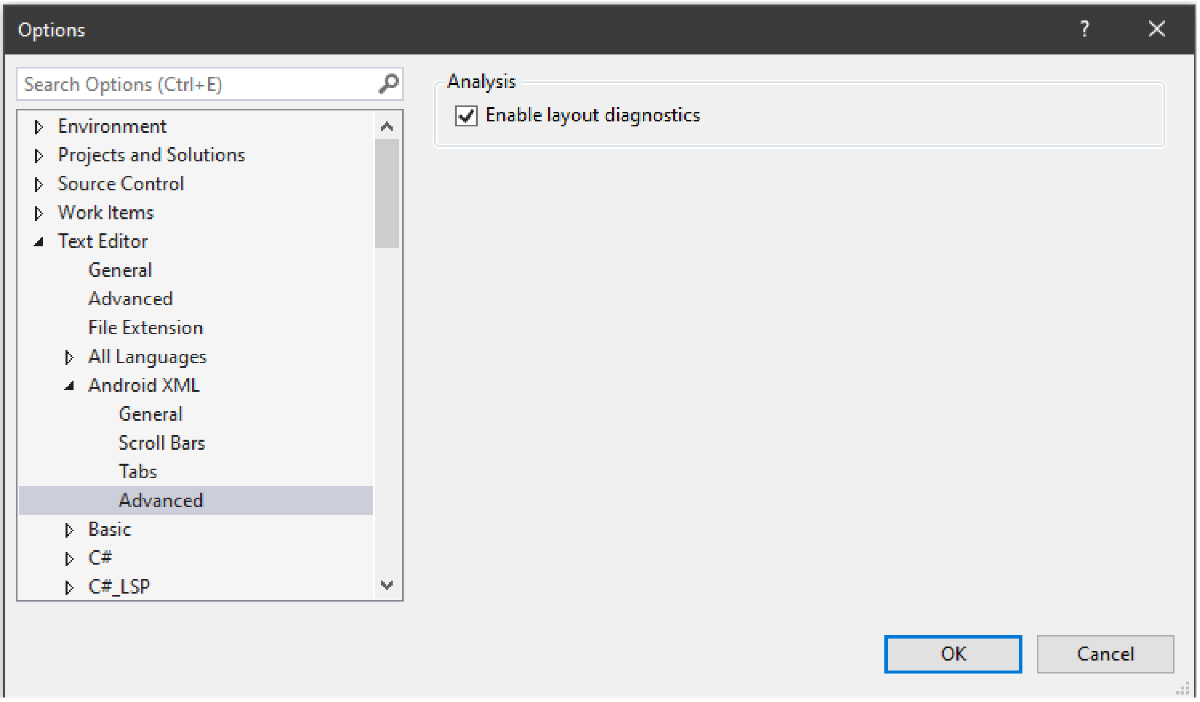
Android layout diagnostics are designed to help improve the quality of Android layout files by highlighting common quality issues and helpful optimizations. This feature is available for both Visual Studio 16.5+ and Visual Studio for Mac 8.5+.

A default set of analyzers is provided for a wide range of issues and each can be customized to cover a project's specific needs. The analyzers are loosely based on the Android linting system.

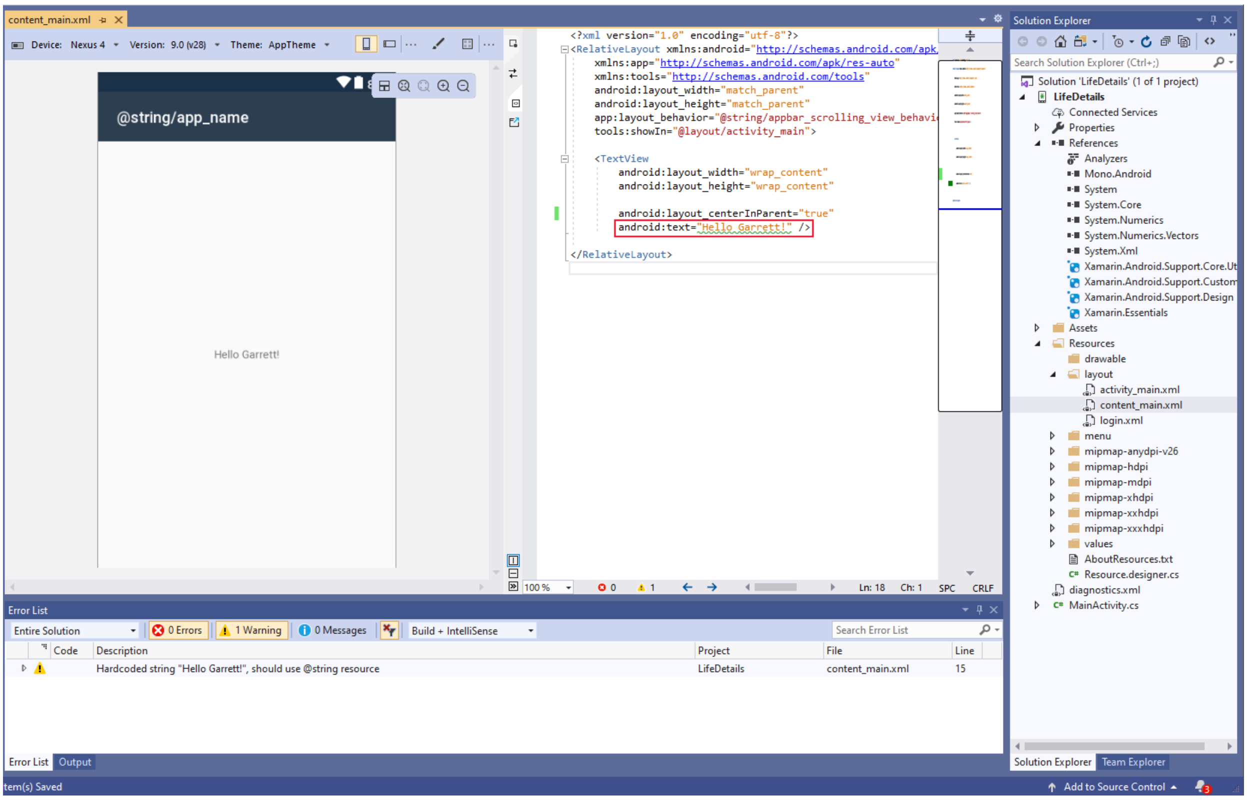
- Visual Studio
- Visual Studio for Mac

Enable Android layout diagnostics on Visual Studio 2019

Make sure the layout diagnostics setting, **Enable layout diagnostics**, is enabled. To access this options page, choose **Tools** > **Options**, and then choose **Text Editor** > **Android XML** > **Advanced**:



Once enabled, the Android layout editor will display issues:



Features

The following sections outline the available features in Android layout diagnostics.

Analyzers

Analyzers are used to help detect issues in layout files, reduce hardcoded values, improve performance, and flag errors. For a list of analyzers, see Android designer diagnostic analyzers

Diagnostic configuration

Analyzers can be configured using an XML file, allowing you to change the default severity level, ignore certain files, and pass in variables.

You can use a baseline file if you have a set of configurations you want to share across multiple Android apps. To use this feature, create a new configuration file and append `-baseline` to the file name. The baseline configurations are applied first, and then the remaining configuration files.

> **TIP**
>
> This can be useful if you want to ignore a set of issues on a new or existing Android app.

The format is:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <issue id="DuplicateIDs" severity="warning">
        <ignore path="Resources/layout/layout1.xml" />
    </issue>
    <issue id="HardcodedText" severity="informational">
        <ignore path="Resources/layout/layout1.xml" />
        <ignore path="Resource/layout/layout2.xml" />
    </issue>
    <issue id="TooManyViews">
        <variable name="MAX_VIEW_COUNT" value="12" />
    </issue>
    <issue id="TooDeepLayout">
        <variable name="MAX_DEPTH" value="12" />
    </issue>
</configuration>
```

NOTE

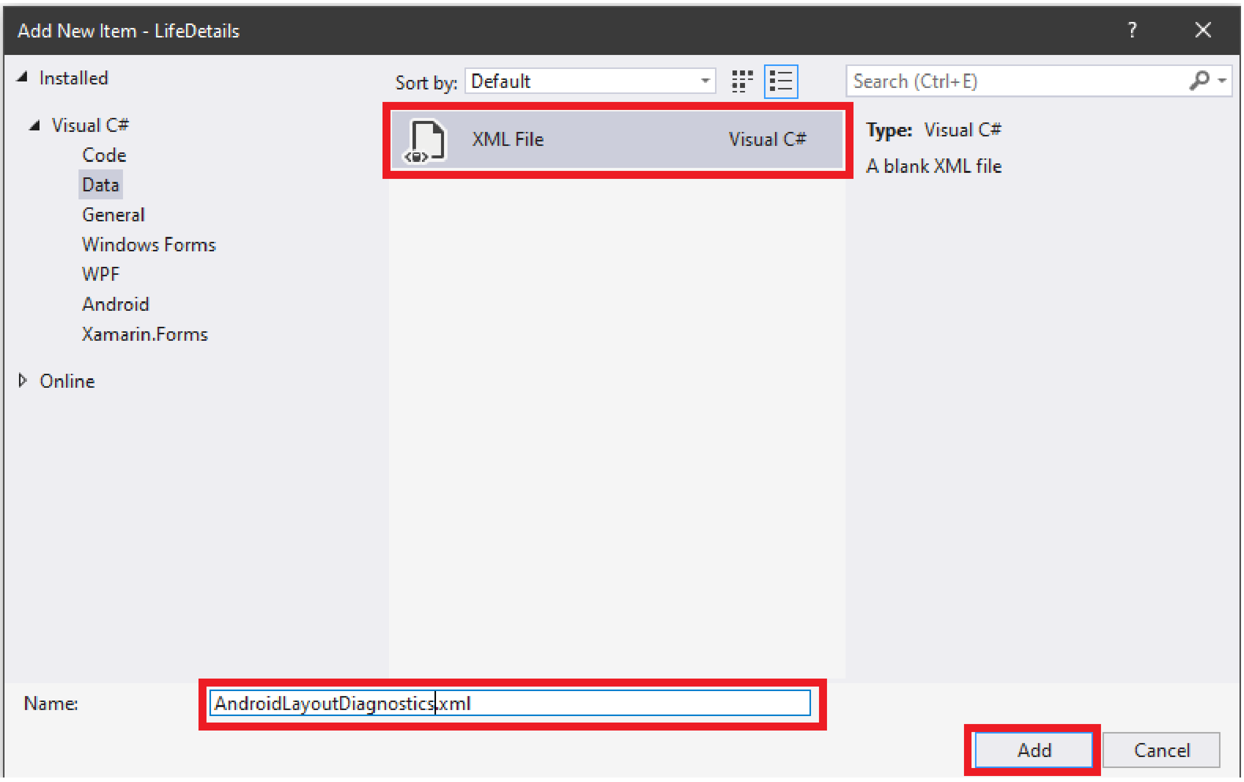
Currently the only variables are `MAX_VIEW_COUNT` (default: 80) and `MAX_DEPTH` (default: 10) for `TooManyViews` and `TooDeepLayout` respectively.

The severity levels are:

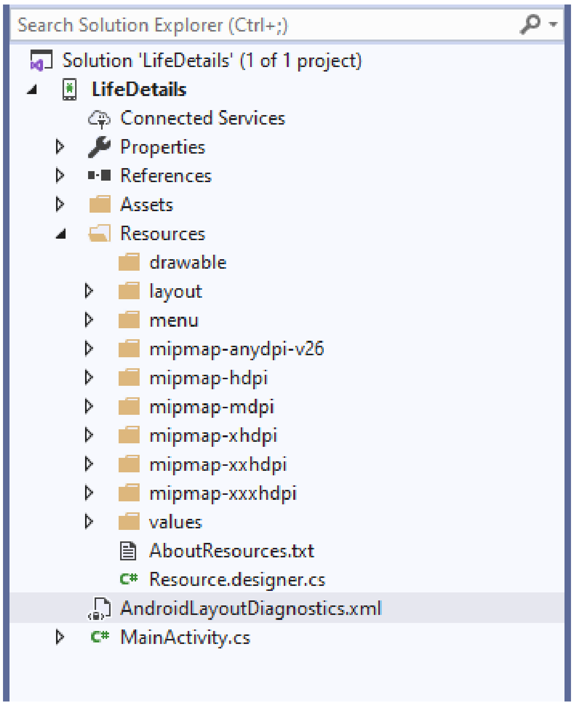
- Suggestion
- Info
- Warning
- Error
- Ignore

Add a configuration file

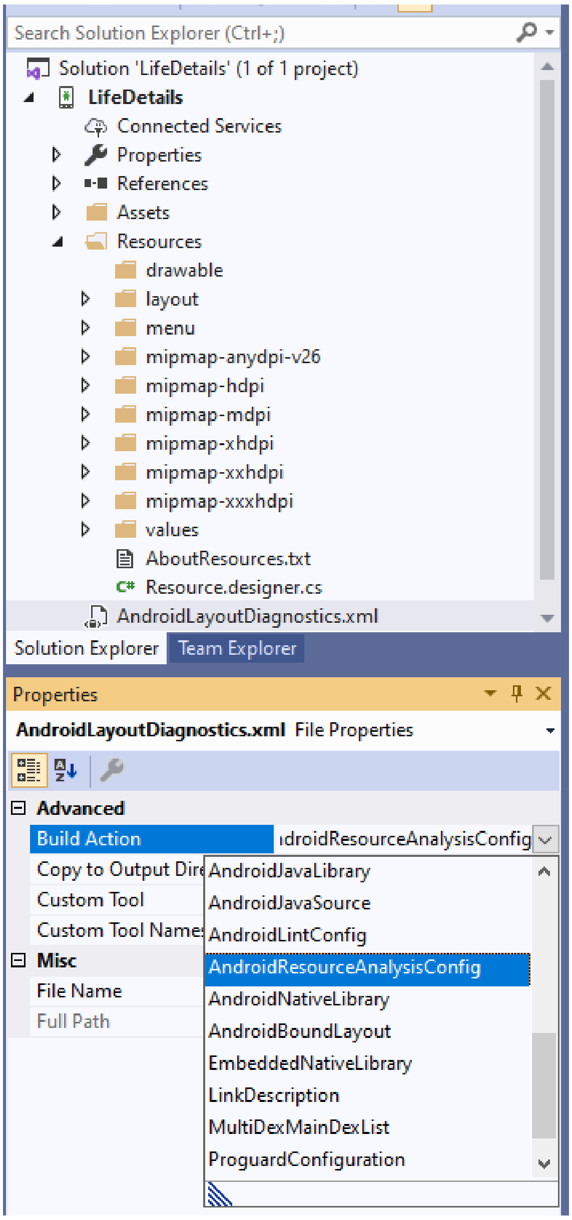
Create a new XML file in the root of an Android app project. The name of the file isn't important, but this example uses `AndroidLayoutDiagnostics.xml`:



Once the new XML file is added, it should appear in the Android app project tree:

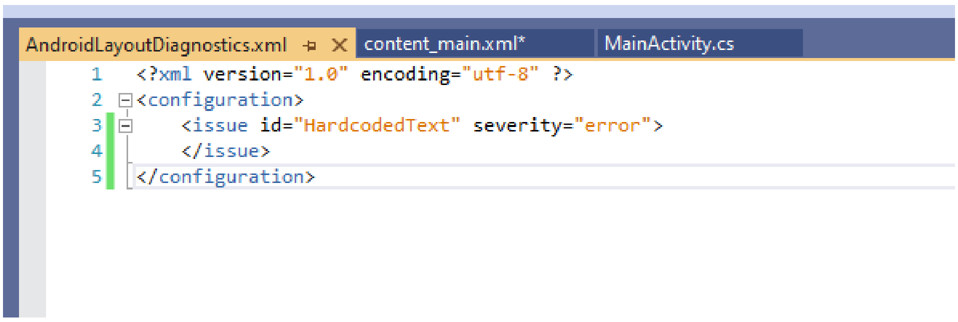


Make sure that the build action is set to **AndroidResourceAnalysisConfig** in the properties panel. The easiest way to pull up the property panel for the new file is to right-click on the file and select properties. Once the properties panel is showing, you should change the **Build Action** to **AndroidResourceAnalysisConfig**:

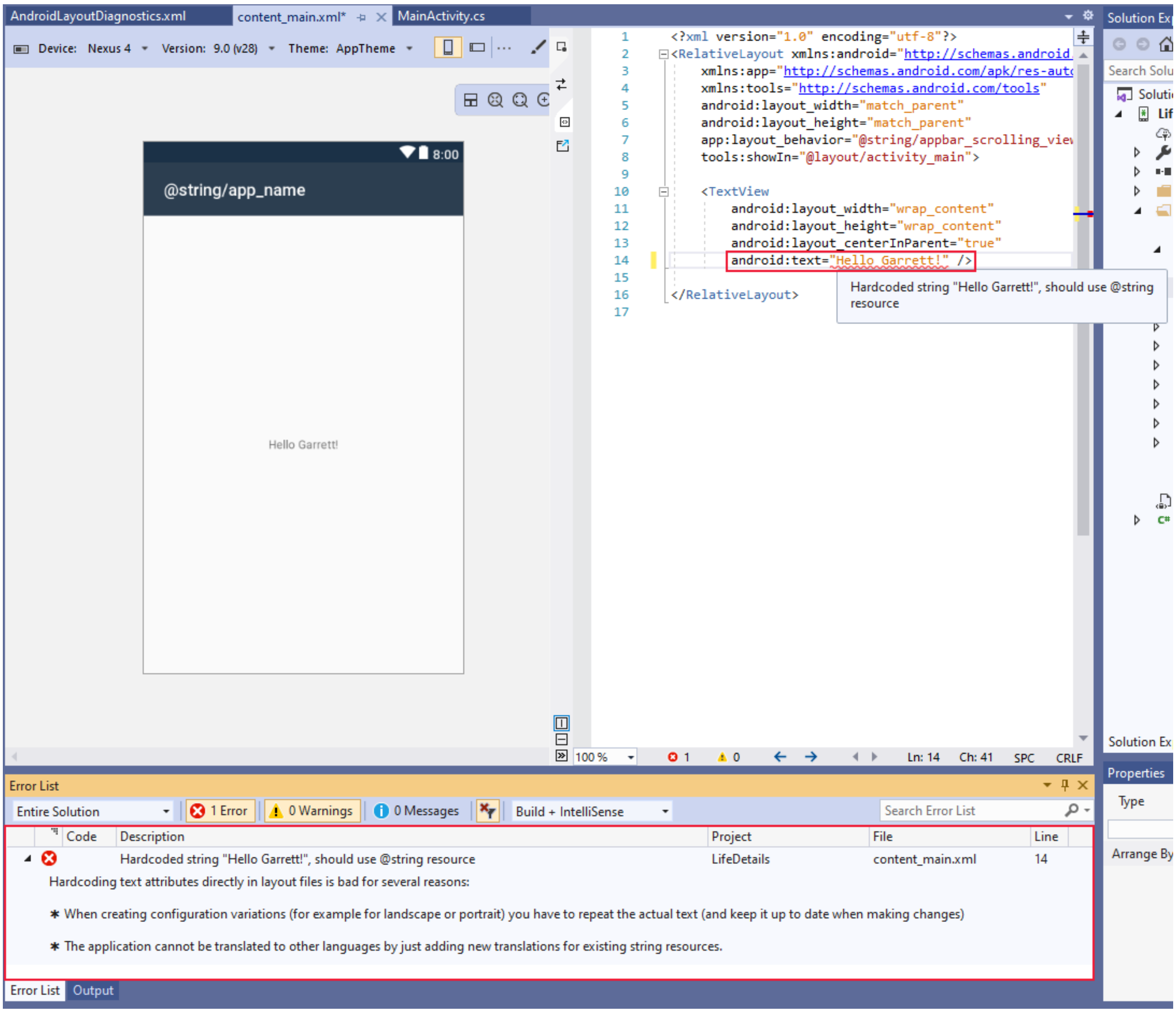


Now that you have a blank XML file you need to add the `<configuration>` root element. At this point, you can adjust the default behavior of any supported issues. If you want to ensure that hard-coded strings are treated as errors add:

```
<issue="HardcodedText" severity="error">
</issue>
```



Now that hard-coded text is considered an error, it's now flagged with a red squiggle in the layout editor:



> **NOTE**
>
> For any new configuration file changes to take effect, any currently open layout files need to be reopened.

Troubleshooting

Here are some possible common problems.

- Make sure there are no XML format error.
- Build action is set correctly to **AndroidResourceAnalysisConfig**.

Known issues

- The error pad isn't populated until after the file is changed the first time.

Related links

- Android Lint Checks
- Improve your code with lint checks

Android designer diagnostic analyzers

4/8/2020 • 3 minutes to read • Edit Online

This guide lists all of the currently supported Android layout diagnostic analyzers.

Accessibility

The following analyzers help to improve accessibility support:

ID	TITLE	SEVERITY	DESCRIPTION
ContentDescription	Image without `contentDescription`	Warning	Missing `contentDescription` attribute on image

Correctness

The following analyzers help fix correctness issues in a layout:

ID	TITLE	SEVERITY	DESCRIPTION	HELP
AdapterViewChildren	AdapterView with children	Warning	AdapterViews cannot have children in XML	Link
MissingId	Fragments should specify an `id` or `tag`	Warning	This `<fragment>` tag should specify an `id` or a `tag` to preserve state across activity restarts	Link
NestedScrollingVertical	Nested vertically scrolling elements	Warning	Nested scrolling widgets	
NestedScrollingHorizontal	Nested horizontally scrolling elements	Warning	Nested scrolling widgets	
ScrollViewSize	ScrollView children with wrong fill_parent/match_parent sizes	Warning	ScrollView children with wrong fill_parent/match_parent sizes	
ScrollViewCount	ScrollViews can have only one child	Warning	A scroll view can have only one child	
MissingAndroidNamespace	Missing Android namespace on attribute	Error	Missing Android XML namespace; your attribute will be interpreted as a custom attribute	
DuplicateIDs	Duplicate IDs	Error	Duplicate ids within a single layout	

ID	TITLE	SEVERITY	DESCRIPTION	HELP
IncludeLayoutParamsMissingWidthAndHeight	Missing both width and height	Error	Ignored layout params on include	Link
IncludeLayoutParamsMissingWidth	Missing width	Error	Ignored layout params on include	Link
IncludeLayoutParamsMissingHeight	Missing height	Error	Ignored layout params on include	Link
Orientation	Missing explicit orientation	Error	Missing explicit orientation	
Suspicious0dp	Suspicious 0dp dimension	Error	Suspicious 0dp dimension	
RequiredSizeWidth	Missing width attribute	Error	Missing attribute: layout_width	
RequiredSizeHeight	Missing height attribute	Error	Missing attribute: layout_height	
WebViewLayout	WebViews in wrap_content parents	Error		
WrongCase	Wrong case for view tag	Error	Wrong case for view tag	Link

Design

The following analyzers help to improve how you join layout files:

ID	TITLE	SEVERITY	DESCRIPTION
HardcodedColor	Hardcoded color	Info	Hardcoded color often leads to inconsistency
HardcodedSize	Hardcoded size	Info	Hardcoded size often leads to inconsistency
HardcodedText	Hardcoded text	Warning	Hardcoded text
UnresolvedResource	Unresolved resource URL	Warning	This resource URL cannot be resolved
XmlErrors	XML syntax error	Error	XML syntax error

Performance

The following analyzers help improve the performance of your layout:

ID	TITLE	SEVERITY	DESCRIPTION
NestedWeights	Nested layout weights	Warning	Nested weights are bad for performance
TooManyViews	Layout has too many views	Warning	Layout has too many views
TooDeepLayout	Layout hierarchy is too deep	Warning	Layout hierarchy is too deep
UselessParent	Useless parent layout	Warning	Useless parent layout
UselessLeaf	Useless leaf layout	Warning	This `%1$s` view is useless (no children, no `background`, no `id`, no `style`)

Usability

The following analyzers help improve layout usability for your customers:

ID	TITLE	SEVERITY	DESCRIPTION
NegativeMargin	Negative Margins	Warning	Negative Margins
MissingInputType	EditText with no inputType	Warning	No input type specified
InputTypePhone	EditText appears to be a phone number	Warning	The view name suggests this is a phone number, but it does not include `phone` in the `inputType`
InputTypeNumber	EditText appears to be a number	Warning	The view name suggests this is a number, but it does not include a numeric `inputType` (such as `numberDecimal`)
InputTypePassword	EditText appears to be a password	Warning	The view name suggests this is a password, but it does not include `password` in the `inputType` (such as `textVisiblePassword`)
InputTypePIN	EditText appears to be a PIN	Warning	The view name suggests this is a password (PIN), but it does not include `numberPassword` in the `inputType`

ID	TITLE	SEVERITY	DESCRIPTION
InputTypeEmail	EditText appears to be an email	Warning	The view name suggests this is an e-mail address, but it does not include `email` in the `inputType` (such as `textEmailAddress`)
InputTypeURI	EditText appears to be a URI	Warning	The view name suggests this is a URI, but it does not include `textUri` in the `inputType`
InputTypeDate	EditText appears to be a date	Warning	The view name suggests this is a date, but it does not include `date` in the `inputType` (such as `datetime`)

Material Theme

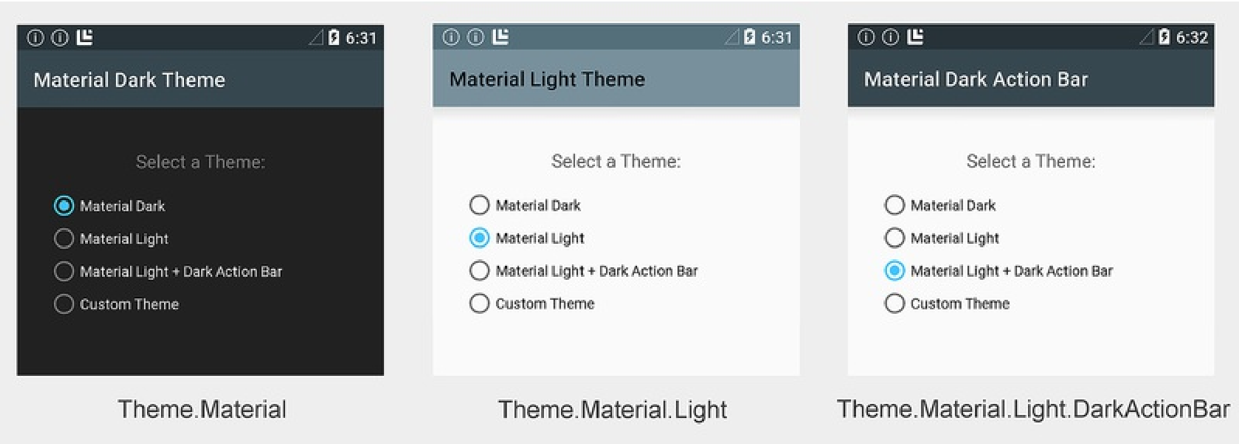
7/8/2021 • 9 minutes to read • Edit Online

Material Theme is a user interface style that determines the look and feel of views and activities starting with Android 5.0 (Lollipop). Material Theme is built into Android 5.0, so it is used by the system UI as well as by applications. Material Theme is not a "theme" in the sense of a system-wide appearance option that a user can dynamically choose from a settings menu. Rather, Material Theme can be thought of as a set of related built-in base styles that you can use to customize the look and feel of your app.

Android provides three Material Theme flavors:

- `Theme.Material` – Dark version of Material Theme; this is the default flavor in Android 5.0.
- `Theme.Material.Light` – Light version of Material Theme.
- `Theme.Material.Light.DarkActionBar` – Light version of Material Theme, but with a dark action bar.

Examples of these Material Theme flavors are displayed here:



You can derive from Material Theme to create your own theme, overriding some or all color attributes. For example, you can create a theme that derives from `Theme.Material.Light`, but overrides the app bar color to match the color of your brand. You can also style individual views; for example, you can create a style for CardView that has more rounded corners and uses a darker background color.

You can use a single theme for an entire app, or you can use different themes for different screens (activities) in an app. In the above screenshots, for example, a single app uses a different theme for each activity to demonstrate the built-in color schemes. Radio buttons switch the app to different activities, and, as a result, display different themes.

Because Material Theme is supported only on Android 5.0 and later, you cannot use it (or a custom theme derived from Material Theme) to theme your app for running on earlier versions of Android. However, you can configure your app to use Material Theme on Android 5.0 devices and gracefully fall back to an earlier theme when it runs on older versions of Android (see the Compatibility section of this article for details).

Requirements

The following is required to use the new Android 5.0 Material Theme features in Xamarin-based apps:

- **Xamarin.Android** – Xamarin.Android 4.20 or later must be installed and configured with either Visual Studio or Visual Studio for Mac.

- **Android SDK** – Android 5.0 (API 21) or later must be installed via the Android SDK Manager.
- **Java JDK 1.8** – JDK 1.7 can be used if you are specifically targeting API level 23 and earlier. JDK 1.8 is available from Oracle.

To learn how to configure an Android 5.0 app project, see Setting Up an Android 5.0 Project.

Using the Built-in Themes

The easiest way to use Material Theme is to configure your app to use a built-in theme without customization. If you don't want to explicitly configure a theme, your app will default to `Theme.Material` (the dark theme). If your app has only one activity, you can configure a theme at the activity level. If your app has multiple activities, you can configure a theme at the application level so that it uses the same theme across all activities, or you can assign different themes to different activities. The following sections explain how to configure themes at the app level and at the activity level.

Theming an Application

To configure an entire application to use a Material Theme flavor, set the `android:theme` attribute of the application node in **AndroidManifest.xml** to one of the following:

- `@android:style/Theme.Material` – Dark theme.
- `@android:style/Theme.Material.Light` – Light theme.
- `@android:style/Theme.Material.Light.DarkActionBar` – Light theme with dark action bar.

The following example configures the application *MyApp* to use the light theme:

```
<application android:label="MyApp"
             android:theme="@android:style/Theme.Material.Light">
</application>
```

Alternately, you can set the application `Theme` attribute in **AssemblyInfo.cs** (or **Properties.cs**). For example:

```
[assembly: Application(Theme="@android:style/Theme.Material.Light")]
```

When the application theme is set to `@android:style/Theme.Material.Light`, every activity in *MyApp* will be displayed using `Theme.Material.Light`.

Theming an Activity

To theme an activity, you add a `Theme` setting to the `[Activity]` attribute above your activity declaration and assign `Theme` to the Material Theme flavor that you want to use. The following example themes an activity with `Theme.Material.Light`:

```
[Activity(Theme = "@android:style/Theme.Material.Light",
          Label = "MyApp", MainLauncher = true, Icon = "@drawable/icon")]
```

Other activities in this app will use the default `Theme.Material` dark color scheme (or, if configured, the application theme setting).

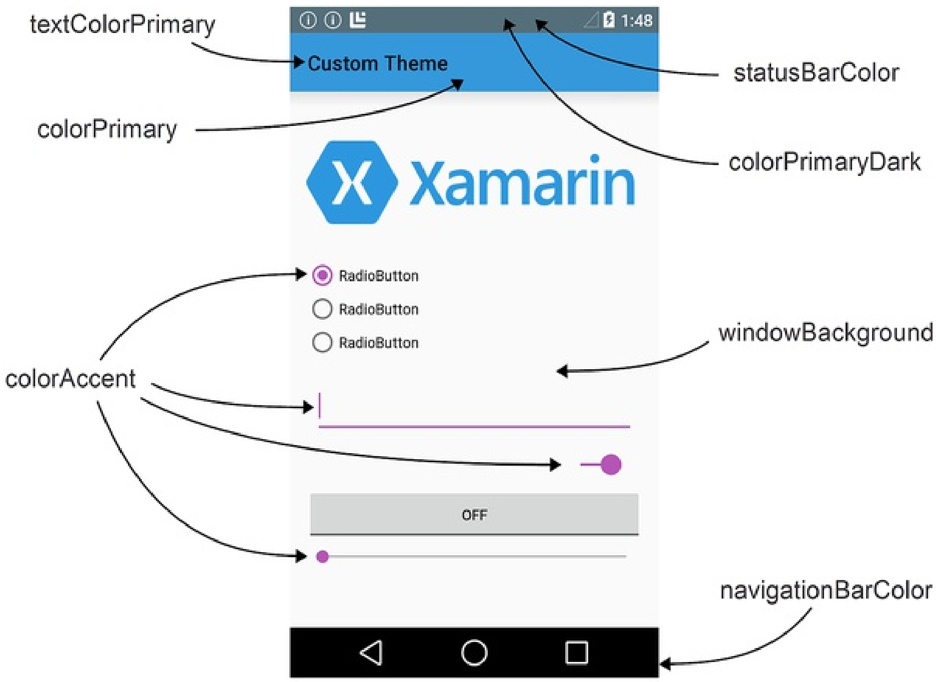
Using Custom Themes

You can enhance your brand by creating a custom theme that styles your app with your brand's colors. To create a custom theme, you define a new style that derives from a built-in Material Theme flavor, overriding the color attributes that you want to change. For example, you can define a custom theme that derives from `Theme.Material.Light.DarkActionBar` and changes the screen background color to beige instead of white.

Material Theme exposes the following layout attributes for customization:

- `colorPrimary` – The color of the app bar.
- `colorPrimaryDark` – The color of the status bar and contextual app bars; this is normally a dark version of `colorPrimary`.
- `colorAccent` – The color of UI controls such as check boxes, radio buttons, and edit text boxes.
- `windowBackground` – The color of the screen background.
- `textColorPrimary` – The color of UI text in the app bar.
- `statusBarColor` – The color of the status bar.
- `navigationBarColor` – The color of the navigation bar.

These screen areas are labeled in the following diagram:



By default, `statusBarColor` is set to the value of `colorPrimaryDark`. You can set `statusBarColor` to a solid color, or you can set it to `@android:color/transparent` to make the status bar transparent. The navigation bar can also be made transparent by setting `navigationBarColor` to `@android:color/transparent`.

Creating a Custom App Theme

You can create a custom app theme by creating and modifying files in the **Resources** folder of your app project. To style your app with a custom theme, use the following steps:

- Create a **colors.xml** file in **Resources/values** — you use this file to define your custom theme colors. For example, you can paste the following code into **colors.xml** to help you get started:

```
<?xml version="1.0" encoding="UTF-8" ?>
<resources>
    <color name="my_blue">#3498DB</color>
    <color name="my_green">#77D065</color>
    <color name="my_purple">#B455B6</color>
    <color name="my_gray">#738182</color>
</resources>
```

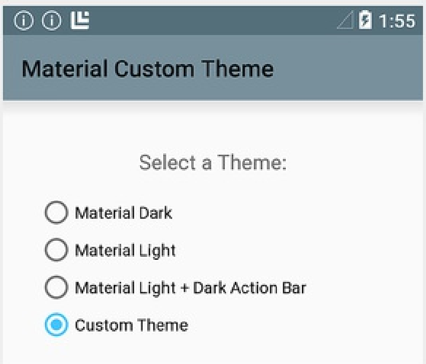
- Modify this example file to define the names and color codes for color resources that you will use in your custom theme.
- Create a **Resources/values-v21** folder. In this folder, create a **styles.xml** file:

Note that **Resources/values-v21** is specific to Android 5.0 – older versions of Android will not read files in this folder.

- Add a `resources` node to **styles.xml** and define a `style` node with the name of your custom theme. For example, here is a **styles.xml** file that defines *MyCustomTheme* (derived from the built-in `Theme.Material.Light` theme style):

```
<?xml version="1.0" encoding="UTF-8" ?>
<resources>
    <!-- Inherit from the light Material Theme -->
    <style name="MyCustomTheme" parent="android:Theme.Material.Light">
        <!-- Customizations go here -->
    </style>
</resources>
```

- At this point, an app that uses *MyCustomTheme* will display the stock `Theme.Material.Light` theme without customizations:

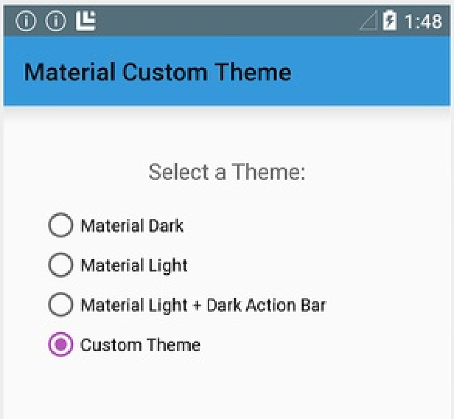


Before Custom Theme Settings

- Add color customizations to **styles.xml** by defining the colors of layout attributes that you want to change. For example, to change the app bar color to `my_blue` and change the color of UI controls to `my_purple`, add color overrides to **styles.xml** that refer to color resources configured in **colors.xml**:

```
<?xml version="1.0" encoding="UTF-8" ?>
<resources>
    <!-- Inherit from the light Material Theme -->
    <style name="MyCustomTheme" parent="android:Theme.Material.Light">
        <!-- Override the app bar color -->
        <item name="android:colorPrimary">@color/my_blue</item>
        <!-- Override the color of UI controls -->
        <item name="android:colorAccent">@color/my_purple</item>
    </style>
</resources>
```

With these changes in place, an app that uses *MyCustomTheme* will display an app bar color in `my_blue` and UI controls in `my_purple`, but use the `Theme.Material.Light` color scheme everywhere else:



After Custom Theme Settings

In this example, *MyCustomTheme* borrows colors from `Theme.Material.Light` for the background color, status bar, and text colors, but it changes the color of the app bar to `my_blue` and sets the color of the radio button to `my_purple`.

Creating a Custom View Style

Android 5.0 also makes it possible for you to style an individual view. After you create **colors.xml** and **styles.xml** (as described in the previous section), you can add a view style to **styles.xml**. To style an individual view, use the following steps:

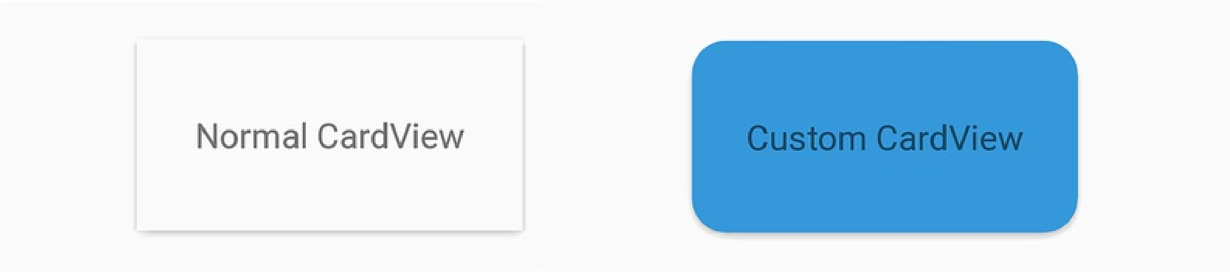
- Edit **Resources/values-v21/styles.xml** and add a `style` node with the name of your custom view style. Set the custom color attributes for your view within this `style` node. For example, to create a custom CardView style that has more rounded corners and uses `my_blue` as the card background color, add a `style` node to **styles.xml** (inside the `resources` node) and configure the background color and corner radius:

```
<!-- Theme an individual view: -->
<style name="CardView.MyBlue">

    <!-- Change the background color to Xamarin blue: -->
    <item name="cardBackgroundColor">@color/my_blue</item>

    <!-- Make the corners very round: -->
    <item name="cardCornerRadius">18dp</item>
</style>
```

- In your layout, set the `style` attribute for that view to match the custom style name that you chose in the previous step. For example:

```
<android.support.v7.widget.CardView
    style="@style/CardView.MyBlue"
    android:layout_width="200dp"
    android:layout_height="100dp"
    android:layout_gravity="center_horizontal">
```

The following screenshot provides an example of the default `CardView` (shown on the left) as compared to a `CardView` that has been styled with the custom `CardView.MyBlue` theme (shown on the right):



In this example, the custom `CardView` is displayed with the background color `my_blue` and an 18dp corner radius.

Compatibility

To style your app so that it uses Material Theme on Android 5.0 but automatically reverts to a downward-compatible style on older Android versions, use the following steps:

- Define a custom theme in **Resources/values-v21/styles.xml** that derives from a Material Theme style. For example:

```
<resources>
    <style name="MyCustomTheme" parent="android:Theme.Material.Light">
        <!-- Your customizations go here -->
    </style>
</resources>
```

- Define a custom theme in **Resources/values/styles.xml** that derives from an older theme, but uses the same theme name as above. For example:

```
<resources>
    <style name="MyCustomTheme" parent="android:Theme.Holo.Light">
        <!-- Your customizations go here -->
    </style>
</resources>
```

- In **AndroidManifest.xml**, configure your app with the custom theme name. For example:

```
<application android:label="MyApp"
             android:theme="@style/MyCustomTheme">
</application>
```

- Alternately, you can style a specific activity using your custom theme:

```
[Activity(Label = "MyActivity", Theme = "@style/MyCustomTheme")]
```

If your theme uses colors defined in a **colors.xml** file, be sure to place this file in **Resources/values** (rather than **Resources/values-v21**) so that both versions of your custom theme can access your color definitions.

When your app runs on an Android 5.0 device, it will use the theme definition specified in **Resources/values-v21/styles.xml**. When this app runs on older Android devices, it will automatically fall back to the theme definition specified in **Resources/values/styles.xml**.

For more information about theme compatibility with older Android versions, see Alternate Resources.

Summary

This article introduced the new Material Theme user interface style included in Android 5.0 (Lollipop). It described the three built-in Material Theme flavors that you can use to style your app, it explained how to create a custom theme for branding your app, and it provided an example of how to theme an individual view. Finally, this article explained how to use Material Theme in your app while maintaining downward compatibility with older versions of Android.

Related Links

- ThemeSwitcher (sample)
- Introduction to Lollipop
- CardView
- Alternate Resources
- Android Lollipop
- Android Pie Developer
- Material Design
- Material Design Principles
- Maintaining Compatibility

User Profile

7/8/2021 • 2 minutes to read • Edit Online

Android has supported enumerating contacts with the ContactsContract provider since API Level 5. For example, listing contacts is as simple as using the ContactContracts.Contacts class as shown in the following code example:

```
// Get the URI for the user's contacts:
var uri = ContactsContract.Contacts.ContentUri;

// Setup the "projection" (columns we want) for only the ID and display name:
string[] projection = {
    ContactsContract.Contacts.InterfaceConsts.Id,
    ContactsContract.Contacts.InterfaceConsts.DisplayName };

// Use a CursorLoader to retrieve the user's contacts data:
CursorLoader loader = new CursorLoader(this, uri, projection, null, null, null);
ICursor cursor = (ICursor)loader.LoadInBackground();

// Print the contact data to the console if reading back succeeds:
if (cursor != null)
{
    if (cursor.MoveToFirst())
    {
        do
        {
            Console.WriteLine("Contact ID: {0}, Contact Name: {1}",
                               cursor.GetString(cursor.GetColumnIndex(projection[0])),
                               cursor.GetString(cursor.GetColumnIndex(projection[1])));
        } while (cursor.MoveToNext());
    }
}
```

Beginning with Android 4 (API Level 14), the ContactsContact.Profile class is available through the `ContactsContract` provider. The `ContactsContact.Profile` provides access to the personal profile for the owner of a device, which includes contact data such as the device owner's name and phone number.

Required Permissions

To read and write contact data, applications must request the `READ_CONTACTS` and `WRITE_CONTACTS` permissions, respectively. Additionally, to read and edit the user profile, applications must request the `READ_PROFILE` and `WRITE_PROFILE` permissions.

Updating Profile Data

Once these permissions have been set, an application can use normal Android techniques to interact with the user profile's data. For example, to update the profile's display name, call ContentResolver.Update with a `Uri` retrieved through the ContactsContract.Profile.ContentRawContactsUri property, as shown below:

```
var values = new ContentValues ();
values.Put (ContactsContract.Contacts.InterfaceConsts.DisplayName, "John Doe");

// Update the user profile with the name "John Doe":
ContentResolver.Update (ContactsContract.Profile.ContentRawContactsUri, values, null, null);
```

Reading Profile Data

Issuing a query to the ContactsContact.Profile.ContentUri reads back the profile data. For example, the following code will read the user profile's display name:

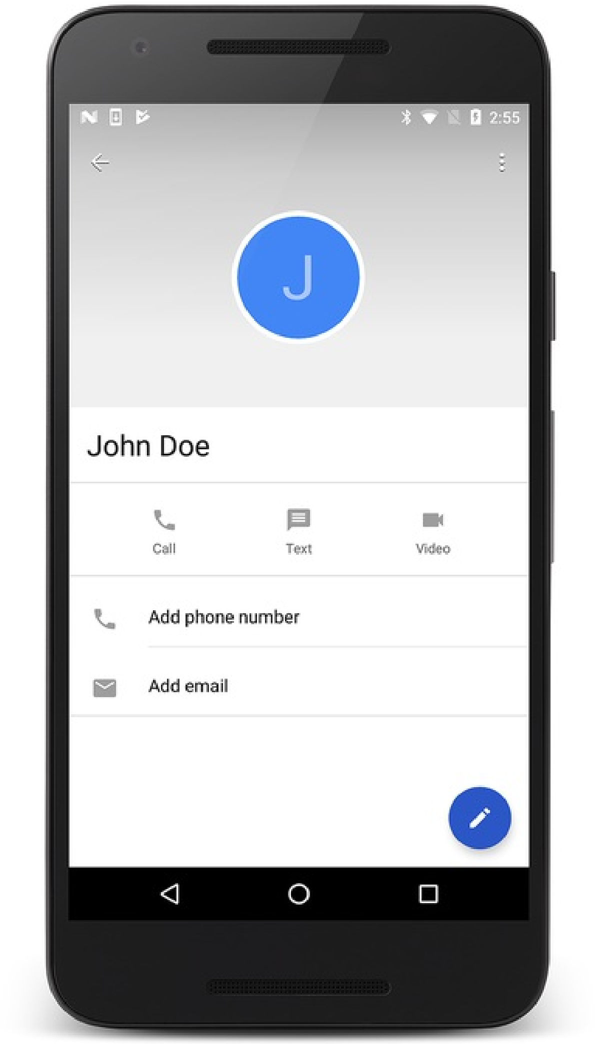
```
// Read the profile
var uri = ContactsContract.Profile.ContentUri;

// Setup the "projection" (column we want) for only the display name:
string[] projection = {
    ContactsContract.Contacts.InterfaceConsts.DisplayName };

// Use a CursorLoader to retrieve the data:
CursorLoader loader = new CursorLoader(this, uri, projection, null, null, null);
ICursor cursor = (ICursor)loader.LoadInBackground();
if (cursor != null)
{
    if (cursor.MoveToFirst ())
    {
        Console.WriteLine(cursor.GetString (cursor.GetColumnIndex (projection [0])));
    }
}
```

Navigating to the User Profile

Finally, to navigate to the user profile, create an Intent with an `ActionView` action and a `ContactsContract.Profile.ContentUri` then pass it to the `StartActivity` method like this:

```
var intent = new Intent (Intent.ActionView,
    ContactsContract.Profile.ContentUri);
StartActivity (intent);
```

When running the above code, the user profile is displayed as illustrated in the following screenshot:



Working with the user profile is similar to interacting with other data in Android, and it offers an additional level of device personalization.

Related Links

- ContactsProviderDemo (sample)

Splash Screen

7/8/2021 • 6 minutes to read • Edit Online

Download the sample

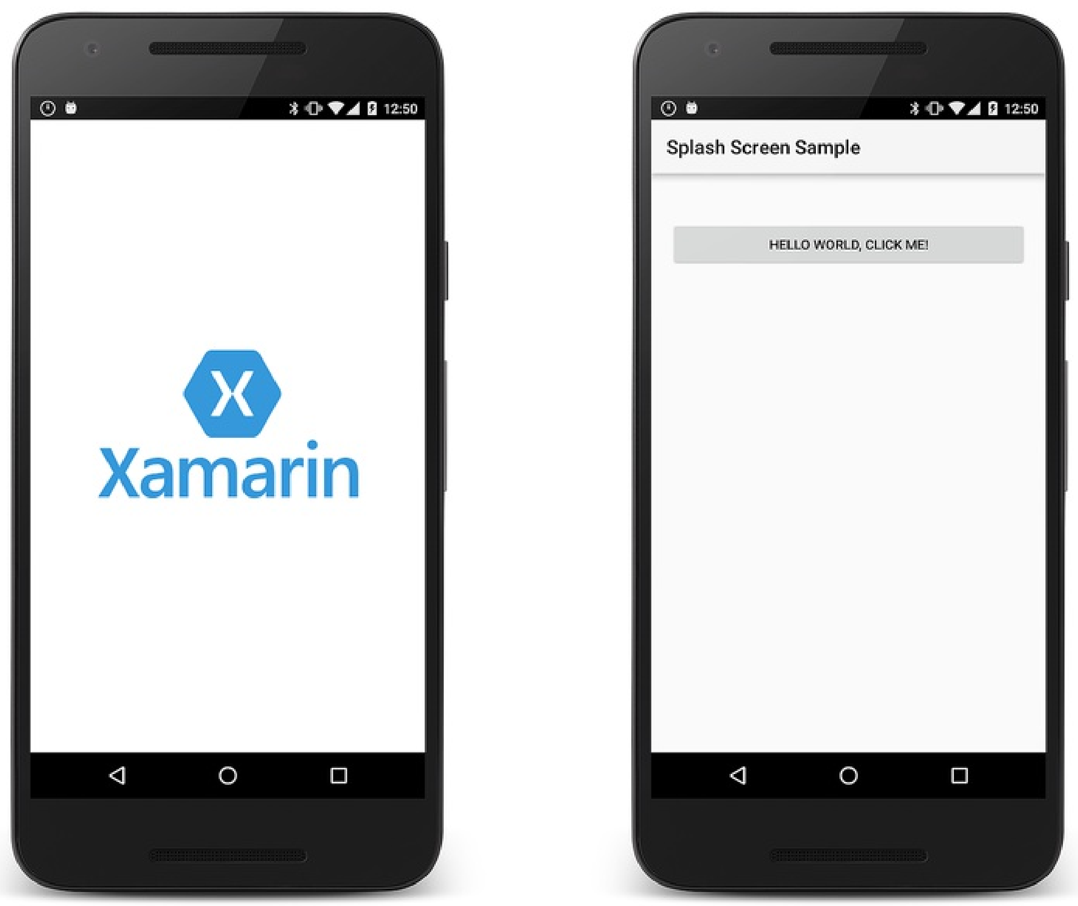
An Android app takes some time to start up, especially when the app is first launched on a device. A splash screen may display start up progress to the user or to indicate branding.

Overview

An Android app takes some time to start up, especially during the first time the app is run on a device (sometimes this is referred to as a *cold start*). The splash screen may display start up progress to the user, or it may display branding information to identify and promote the application.

This guide discusses one technique to implement a splash screen in an Android application. It covers the following steps:

1. Creating a drawable resource for the splash screen.
2. Defining a new theme that will display the drawable resource.
3. Adding a new Activity to the application that will be used as the splash screen defined by the theme created in the previous step.



Requirements

This guide assumes that the application targets Android API level 21 or higher. The application must also have the **Xamarin.Android.Support.v4** and **Xamarin.Android.Support.v7.AppCompat** NuGet packages added to the project.

All of the code and XML in this guide may be found in the SplashScreen sample project for this guide.

Implementing A Splash Screen

The quickest way to render and display the splash screen is to create a custom theme and apply it to an Activity that exhibits the splash screen. When the Activity is rendered, it loads the theme and applies the drawable resource (referenced by the theme) to the background of the activity. This approach avoids the need for creating a layout file.

The splash screen is implemented as an Activity that displays the branded drawable, performs any initializations, and starts up any tasks. Once the app has bootstrapped, the splash screen Activity starts the main Activity and removes itself from the application back stack.

Creating a Drawable for the Splash Screen

The splash screen will display an XML drawable in the background of the splash screen Activity. It is necessary to use a bitmapped image (such as a PNG or JPG) for the image to display.

The sample application defines a drawable called **splash_screen.xml**. This drawable uses a Layer List to center the splash screen image in the application as shown in the following xml:

```
<?xml version="1.0" encoding="utf-8"?>
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">
  <item>
    <color android:color="@color/splash_background"/>
  </item>
  <item>
    <bitmap
        android:src="@drawable/splash_logo"
        android:tileMode="disabled"
        android:gravity="center"/>
  </item>
</layer-list>
```

This `layer-list` centers the splash image on a background color specified by the `@color/splash_background` resource. The sample application defines this color in the **Resources/values/colors.xml** file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  ...
  <color name="splash_background">#FFFFFF</color>
</resources>
```

For more information about `Drawable` objects see the Google documentation on Android Drawable.

Implementing a Theme

To create a custom theme for the splash screen Activity, edit (or add) the file **values/styles.xml** and create a new `style` element for the splash screen. A sample **values/style.xml** file is shown below with a `style` named **MyTheme.Splash**:

```
<resources>
  <style name="MyTheme.Base" parent="Theme.AppCompat.Light">
  </style>

  <style name="MyTheme" parent="MyTheme.Base">
  </style>

  <style name="MyTheme.Splash" parent ="Theme.AppCompat.Light.NoActionBar">
    <item name="android:windowBackground">@drawable/splash_screen</item>
    <item name="android:windowNoTitle">true</item>
    <item name="android:windowFullscreen">true</item>
    <item name="android:windowContentOverlay">@null</item>
    <item name="android:windowActionBar">true</item>
  </style>
</resources>
```

MyTheme.Splash is very spartan – it declares the window background, explicitly removes the title bar from the window, and declares that it is full-screen. If you want to create a splash screen that emulates the UI of your app before the activity inflates the first layout, you can use `windowContentOverlay` rather than `windowBackground` in your style definition. In this case, you must also modify the **splash_screen.xml** drawable so that it displays an emulation of your UI.

Create a Splash Activity

Now we need a new Activity for Android to launch that has our splash image and performs any startup tasks. The following code is an example of a complete splash screen implementation:

```
[Activity(Theme = "@style/MyTheme.Splash", MainLauncher = true, NoHistory = true)]
public class SplashActivity : AppCompatActivity
{
    static readonly string TAG = "X:" + typeof(SplashActivity).Name;

    public override void OnCreate(Bundle savedInstanceState, PersistableBundle persistentState)
    {
        base.OnCreate(savedInstanceState, persistentState);
        Log.Debug(TAG, "SplashActivity.OnCreate");
    }

    // Launches the startup task
    protected override void OnResume()
    {
        base.OnResume();
        Task startupWork = new Task(() => { SimulateStartup(); });
        startupWork.Start();
    }

    // Simulates background work that happens behind the splash screen
    async void SimulateStartup ()
    {
        Log.Debug(TAG, "Performing some startup work that takes a bit of time.");
        await Task.Delay (8000); // Simulate a bit of startup work.
        Log.Debug(TAG, "Startup work is finished - starting MainActivity.");
        StartActivity(new Intent(Application.Context, typeof (MainActivity)));
    }
}
```

`SplashActivity` explicitly uses the theme that was created in the previous section, overriding the default theme of the application. There is no need to load a layout in `OnCreate` as the theme declares a drawable as the background.

It is important to set the `NoHistory=true` attribute so that the Activity is removed from the back stack. To prevent the back button from canceling the startup process, you can also override `OnBackPressed` and have it do nothing:

```
public override void OnBackPressed() { }
```

The startup work is performed asynchronously in `OnResume`. This is necessary so that the startup work does not slow down or delay the appearance of the launch screen. When the work has completed, `SplashActivity` will launch `MainActivity` and the user may begin interacting with the app.

This new `SplashActivity` is set as the launcher activity for the application by setting the `MainLauncher` attribute to `true`. Because `SplashActivity` is now the launcher activity, you must edit `MainActivity.cs`, and remove the `MainLauncher` attribute from `MainActivity`:

```
[Activity(Label = "@string/ApplicationName")]
public class MainActivity : AppCompatActivity
{
    // Code omitted for brevity
}
```

Landscape Mode

The splash screen implemented in the previous steps will display correctly in both portrait and landscape mode. However, in some cases it is necessary to have separate splash screens for portrait and landscape modes (for example, if the splash image is full-screen).

To add a splash screen for landscape mode, use the following steps:

1. In the **Resources/drawable** folder, add the landscape version of the splash screen image you want to use. In this example, **splash_logo_land.png** is the landscape version of the logo that was used in the above examples (it uses white lettering instead of blue).

2. In the **Resources/drawable** folder, create a landscape version of the `layer-list` drawable that was defined earlier (for example, **splash_screen_land.xml**). In this file, set the bitmap path to the landscape version of the splash screen image. In the following example, **splash_screen_land.xml** uses **splash_logo_land.png**:

   ```
   <?xml version="1.0" encoding="utf-8"?>
   <layer-list xmlns:android="http://schemas.android.com/apk/res/android">
     <item>
       <color android:color="@color/splash_background"/>
     </item>
     <item>
       <bitmap
           android:src="@drawable/splash_logo_land"
           android:tileMode="disabled"
           android:gravity="center"/>
     </item>
   </layer-list>
   ```

3. Create the **Resources/values-land** folder if it doesn't already exist.

4. Add the files **colors.xml** and **style.xml** to **values-land** (these can be copied and modified from the existing **values/colors.xml** and **values/style.xml** files).

5. Modify **values-land/style.xml** so that it uses the landscape version of the drawable for `windowBackground`. In this example, **splash_screen_land.xml** is used:

```
<resources>
  <style name="MyTheme.Base" parent="Theme.AppCompat.Light">
  </style>
    <style name="MyTheme" parent="MyTheme.Base">
  </style>
  <style name="MyTheme.Splash" parent ="Theme.AppCompat.Light.NoActionBar">
    <item name="android:windowBackground">@drawable/splash_screen_land</item>
    <item name="android:windowNoTitle">true</item>
    <item name="android:windowFullscreen">true</item>
    <item name="android:windowContentOverlay">@null</item>
    <item name="android:windowActionBar">true</item>
  </style>
</resources>
```

6. Modify **values-land/colors.xml** to configure the colors you want to use for the landscape version of the splash screen. In this example, the splash background color is changed to blue for landscape mode:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="primary">#2196F3</color>
  <color name="primaryDark">#1976D2</color>
  <color name="accent">#FFC107</color>
  <color name="window_background">#F5F5F5</color>
  <color name="splash_background">#3498DB</color>
</resources>
```

7. Build and run the app again. Rotate the device to landscape mode while the splash screen is still displayed. The splash screen changes to the landscape version:

Note that the use of a landscape-mode splash screen does not always provide a seamless experience. By default, Android launches the app in portrait mode and transitions it to landscape mode even if the device is already in landscape mode. As a result, if the app is launched while the device is in landscape mode, the device briefly presents the portrait splash screen and then animates rotation from the portrait to the landscape splash screen. Unfortunately, this initial portrait-to-landscape transition takes place even when `ScreenOrientation = Android.Content.PM.ScreenOrientation.Landscape` is specified in the splash Activity's flags.

The best way to work around this limitation is to create a single splash screen image that renders correctly in both portrait and landscape modes.

Summary

This guide discussed one way to implement a splash screen in a Xamarin.Android application; namely, applying a custom theme to the launch activity.

Related Links

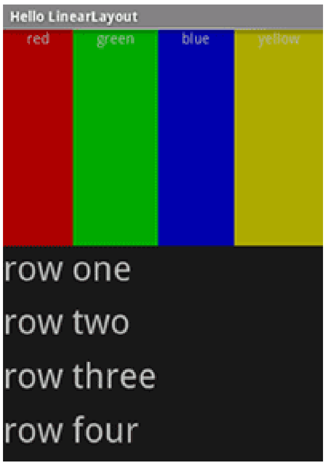
- SplashScreen (sample)
- layer-list Drawable
- Material Design Patterns - Launch Screens

Xamarin.Android Layouts

7/8/2021 • 2 minutes to read • Edit Online

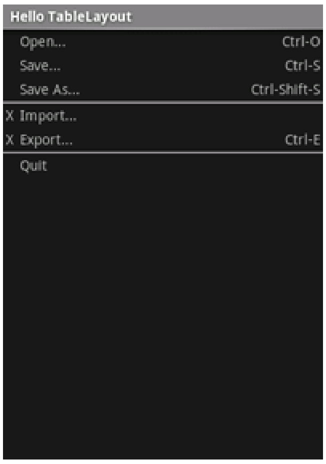
Layouts are used to arrange the elements that make up the UI interface of a screen (such as an Activity). The following sections explain how to use the most commonly-used layouts in Xamarin.Android apps.

- LinearLayout is a view group that displays child view elements in a linear direction, either vertically or horizontally.

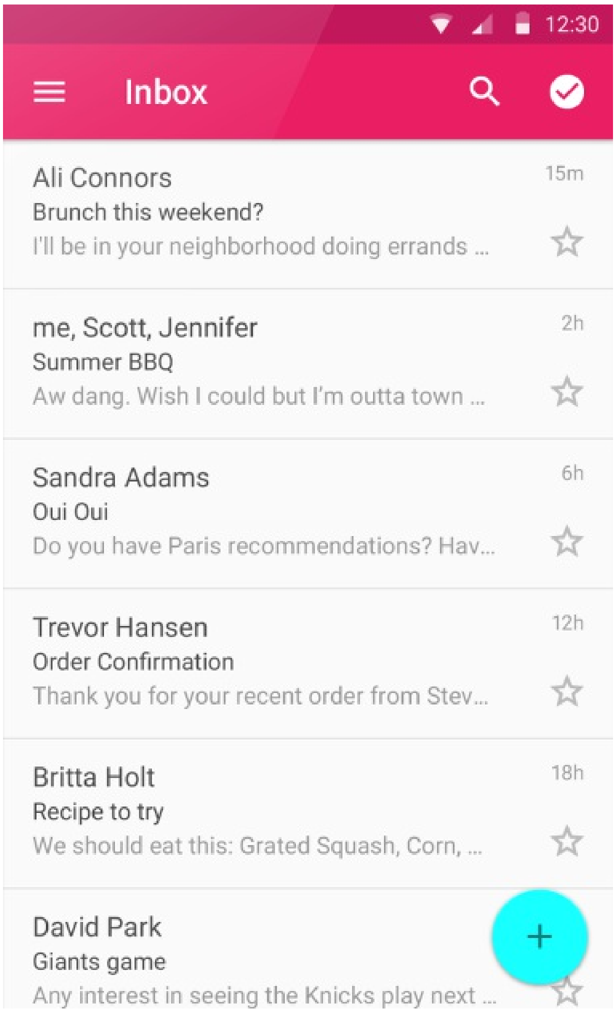


- RelativeLayout is view group that displays child view elements in a relative position. The position of a view can be specified as relative to sibling elements.

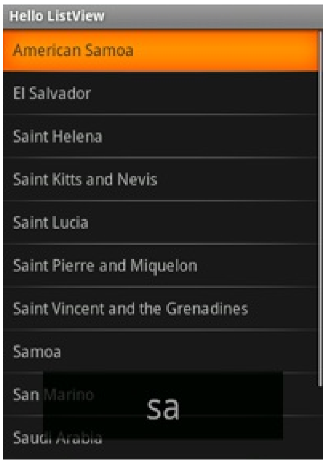
- TableLayout is a view group that displays child view elements in rows and columns.



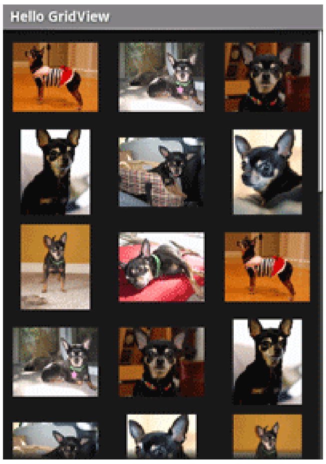
- RecyclerView is a UI element that displays a collection of items in a list or a grid, enabling the user to scroll through the collection.



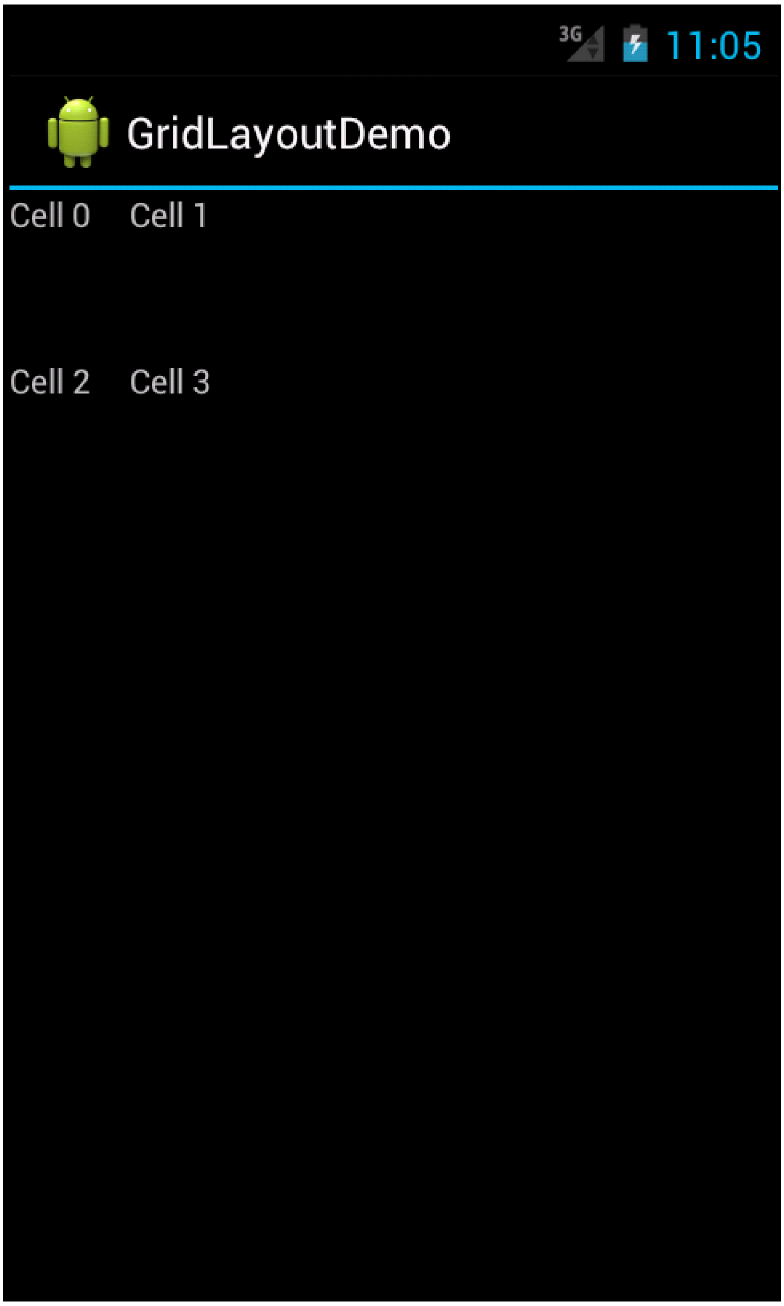
- ListView is a view group that creates a list of scrollable items. The list items are automatically inserted into the list using a list adapter. The `ListView` is an important UI component of Android applications because it is used everywhere from short lists of menu options to long lists of contacts or internet favorites. It provides a simple way to present a scrolling list of rows that can either be formatted with a built-in style or customized extensively. A ListView instance requires an Adapter to feed it with data contained in row views.



- GridView is a UI element that displays items in a two-dimensional grid that can be scrolled.



- GridLayout is a view group that supports laying out views in a 2D grid, similar to an HTML table.



- Tabbed Layouts are a popular user interface pattern in mobile applications because of their simplicity and usability. They provide a consistent, easy way to navigate between various screens in an application.


Artists
Albums
Songs
This is the Artists tab

Xamarin.Android LinearLayout

7/8/2021 • 2 minutes to read • Edit Online

`LinearLayout` is a `ViewGroup` that displays child `View` elements in a linear direction, either vertically or horizontally.

You should be careful about over-using the `LinearLayout`. If you begin nesting multiple `LinearLayout`s, you may want to consider using a `RelativeLayout` instead.

Start a new project named **HelloLinearLayout**.

Open **Resources/Layout/Main.axml** and insert the following:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation=    "vertical"
    android:layout_width=    "match_parent"
    android:layout_height=    "match_parent"    >

  <LinearLayout
      android:orientation=    "horizontal"
      android:layout_width=    "match_parent"
      android:layout_height=    "match_parent"
      android:layout_weight=    "1"    >
      <TextView
          android:text=    "red"
          android:gravity=    "center_horizontal"
          android:background=    "#aa0000"
          android:layout_width=    "wrap_content"
          android:layout_height=    "match_parent"
          android:layout_weight=    "1"    />
      <TextView
          android:text=    "green"
          android:gravity=    "center_horizontal"
          android:background=    "#00aa00"
          android:layout_width=    "wrap_content"
          android:layout_height=    "match_parent"
          android:layout_weight=    "1"    />
      <TextView
          android:text=    "blue"
          android:gravity=    "center_horizontal"
          android:background=    "#0000aa"
          android:layout_width=    "wrap_content"
          android:layout_height=    "match_parent"
          android:layout_weight=    "1"    />
      <TextView
          android:text=    "yellow"
          android:gravity=    "center_horizontal"
          android:background=    "#aaaa00"
          android:layout_width=    "wrap_content"
          android:layout_height=    "match_parent"
          android:layout_weight=    "1"    />
  </LinearLayout>

  <LinearLayout
    android:orientation=    "vertical"
    android:layout_width=    "match_parent"
    android:layout_height=    "match_parent"
    android:layout_weight=    "1"    >
    <TextView
        android:text=    "row one"
```

```
            android:textSize=    "15pt"
            android:layout_width=    "match_parent"
            android:layout_height=    "wrap_content"
            android:layout_weight=    "1"    />
        <TextView
            android:text=    "row two"
            android:textSize=    "15pt"
            android:layout_width=    "match_parent"
            android:layout_height=    "wrap_content"
            android:layout_weight=    "1"    />
        <TextView
            android:text=    "row three"
            android:textSize=    "15pt"
            android:layout_width=    "match_parent"
            android:layout_height=    "wrap_content"
            android:layout_weight=    "1"    />
        <TextView
            android:text=    "row four"
            android:textSize=    "15pt"
            android:layout_width=    "match_parent"
            android:layout_height=    "wrap_content"
           android:layout_weight=    "1"    />
    </LinearLayout>

</LinearLayout>
```

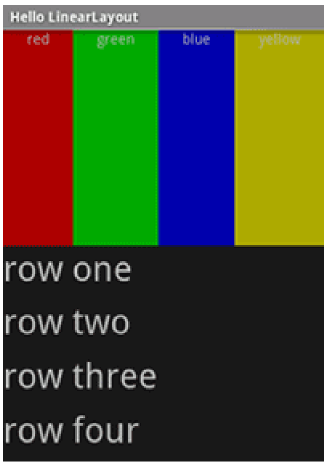
Carefully inspect this XML. There is a root `LinearLayout` that defines its orientation to be vertical – all child `View`s (of which it has two) will be stacked vertically. The first child is another `LinearLayout` that uses a horizontal orientation and the second child is a `LinearLayout` that uses a vertical orientation. Each of these nested `LinearLayout`s contain several `TextView` elements, which are oriented with each other in the manner defined by their parent `LinearLayout`.

Now open **HelloLinearLayout.cs** and be sure it loads the **Resources/Layout/Main.axml** layout in the `OnCreate()` method:

```
protected override void OnCreate (Bundle savedInstanceState)
{
    base.OnCreate (savedInstanceState);
    SetContentView (Resource.Layout.Main);
}
```

The `SetContentView(int)`) method loads the layout file for the `Activity`, specified by the resource ID – `Resources.Layout.Main` refers to the **Resources/Layout/Main.axml** layout file.

Run the application. You should see the following:



Notice how the XML attributes define each View's behavior. Try experimenting with different values for `android:layout_weight` to see how the screen real estate is distributed based on the weight of each element. See the Common Layout Objects document for more about how `LinearLayout` handles the `android:layout_weight` attribute.

References

- `LinearLayout`
- `TextView`

Xamarin.Android RelativeLayout

7/8/2021 • 2 minutes to read • Edit Online

`RelativeLayout` is a `ViewGroup` that displays child `View` elements in relative positions. The position of a `View` can be specified as relative to sibling elements (such as to the left-of or below a given element) or in positions relative to the `RelativeLayout` area (such as aligned to the bottom, left of center).

A `RelativeLayout` is a very powerful utility for designing a user interface because it can eliminate nested `ViewGroup` s. If you find yourself using several nested `LinearLayout` groups, you may be able to replace them with a single `RelativeLayout` .

Start a new project named **HelloRelativeLayout**.

Open the **Resources/Layout/Main.axml** file and insert the following:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/label"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Type here:"/>
    <EditText
        android:id="@+id/entry"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="@android:drawable/editbox_background"
        android:layout_below="@id/label"/>
    <Button
        android:id="@+id/ok"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/entry"
        android:layout_alignParentRight="true"
        android:layout_marginLeft="10dip"
        android:text="OK" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toLeftOf="@id/ok"
        android:layout_alignTop="@id/ok"
        android:text="Cancel" />
</RelativeLayout>
```

Notice each of the `android:layout_*` attributes, such as `layout_below` , `layout_alignParentRight` , and `layout_toLeftOf` . When using a `RelativeLayout` , you can use these attributes to describe how you want to position each `View` . Each one of these attributes define a different kind of relative position. Some attributes use the resource ID of a sibling `View` to define its own relative position. For example, the last `Button` is defined to lie to the left-of and aligned-with-the-top-of the `View` identified by the ID `ok` (which is the previous `Button`).

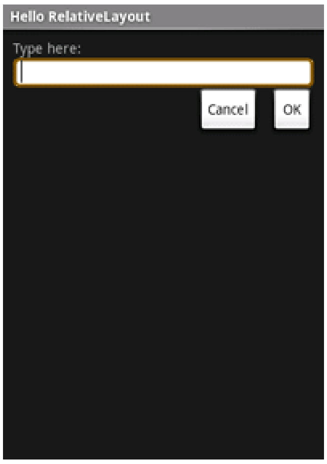
All of the available layout attributes are defined in `RelativeLayout.LayoutParams` .

Make sure you load this layout in the `OnCreate()` method:

```
protected override void OnCreate (Bundle savedInstanceState)
{
    base.OnCreate (savedInstanceState);
    SetContentView (Resource.Layout.Main);
}
```

The `SetContentView(int)` method loads the layout file for the `Activity`, specified by the resource ID — `Resource.Layout.Main` refers to the **Resources/Layout/Main.axml** layout file.

Run the application. You should see the following layout:



Resources

- `RelativeLayout`
- `RelativeLayout.LayoutParams`
- `TextView`
- `EditText`
- `Button`

Xamarin.Android TableLayout

7/8/2021 • 2 minutes to read • Edit Online

`TableLayout` is a `ViewGroup` that displays child `View` elements in rows and columns.

Start a new project named **HelloTableLayout**.

Open the **Resources/Layout/content_main.xml** file and insert the following:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:stretchColumns="1">

    <TableRow
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_column="1"
            android:text="Open..."
            android:padding="3dip"/>
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Ctrl-O"
            android:gravity="right"
            android:padding="3dip"/>
    </TableRow>

    <TableRow
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_column="1"
            android:text="Save..."
            android:padding="3dip"/>
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Ctrl-S"
            android:gravity="right"
            android:padding="3dip"/>
    </TableRow>

    <TableRow
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_column="1"
            android:text="Save As..."
            android:padding="3dip"/>
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Ctrl-Shift-S"
```

```
            android:gravity="right"
            android:padding="3dip"/>
    </TableRow>

    <View
        android:layout_width="wrap_content"
        android:layout_height="2dip"
        android:background="#FF909090"/>

    <TableRow
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="X"
            android:padding="3dip"/>
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Import..."
            android:padding="3dip"/>
    </TableRow>

    <TableRow
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="X"
            android:padding="3dip"/>
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Export..."
            android:padding="3dip"/>
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Ctrl-E"
            android:gravity="right"
            android:padding="3dip"/>
    </TableRow>

    <View
        android:layout_width="wrap_content"
        android:layout_height="2dip"
        android:background="#FF909090"/>

    <TableRow
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_column="1"
            android:text="Quit"
            android:padding="3dip"/>
    </TableRow>
</TableLayout>
```

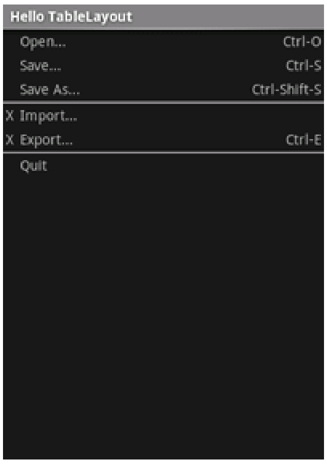
Notice how this resembles the structure of an HTML table. The `TableLayout` element is like the HTML `<table>` element; `TableRow` is like a `<tr>` element; but for the cells, you can use any kind of `View` element. In this example, a `TextView` is used for each cell. In between some of the rows, there is also a basic `View`, which is used to draw a horizontal line.

Make sure your **HelloTableLayout** Activity loads this layout in the `OnCreate()` method:

```
protected override void OnCreate (Bundle savedInstanceState)
{
    base.OnCreate (savedInstanceState);
    SetContentView (Resource.Layout.Main);
}
```

The `SetContentView(int)`) method loads the layout file for the `Activity`, specified by the resource ID — `Resource.Layout.Main` refers to the **Resources/Layout/Main.axml** layout file.

Run the application. You should see the following:



References

- `TableLayout`
- `TableRow`
- `TextView`

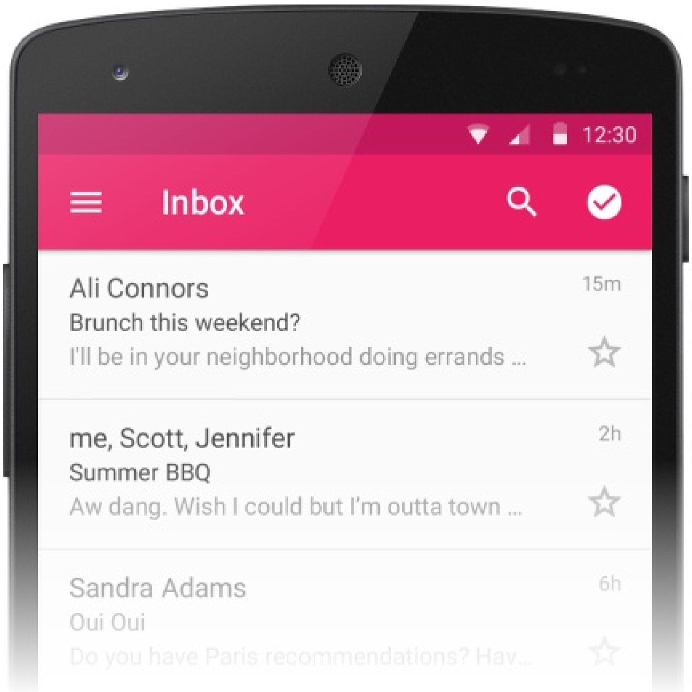
RecyclerView

7/8/2021 • 3 minutes to read • Edit Online

RecyclerView is a view group for displaying collections; it is designed to be a more flexible replacement for older view groups such as ListView and GridView. This guide explains how to use and customize RecyclerView in Xamarin.Android applications.

RecyclerView

Many apps need to display collections of the same type (such as messages, contacts, images, or songs); often, this collection is too large to fit on the screen, so the collection is presented in a small window that can smoothly scroll through all items in the collection. `RecyclerView` is an Android widget that displays a collection of items in a list or a grid, enabling the user to scroll through the collection. The following is a screenshot of an example app that uses `RecyclerView` to display email inbox contents in a vertical scrolling list:



`RecyclerView` offers two compelling features:

- It has a flexible architecture that lets you modify its behavior by plugging in your preferred components.
- It is efficient with large collections because it reuses item views and requires the use of *view holders* to cache view references.

This guide explains how to use `RecyclerView` in Xamarin.Android applications; it explains how to add the `RecyclerView` package to your Xamarin.Android project, and it describes how `RecyclerView` functions in a typical application. Real code examples are provided to show you how to integrate `RecyclerView` into your application, how to implement item-view click, and how to refresh `RecyclerView` when its underlying data changes. This guide assumes that you are familiar with Xamarin.Android development.

Requirements

Although `RecyclerView` is often associated with Android 5.0 Lollipop, it is offered as a support library – `RecyclerView` works with apps that target API level 7 (Android 2.1) and later. The following is required to use `RecyclerView` in Xamarin-based applications:

- **Xamarin.Android** – Xamarin.Android 4.20 or later must be installed and configured with either Visual Studio or Visual Studio for Mac.

- Your app project must include the **Xamarin.Android.Support.v7.RecyclerView** package. For more information about installing NuGet packages, see Walkthrough: Including a NuGet in your project.

Overview

`RecyclerView` can be thought of as a replacement for the `ListView` and `GridView` widgets in Android. Like its predecessors, `RecyclerView` is designed to display a large data set in a small window, but `RecyclerView` offers more layout options and is better optimized for displaying large collections. If you are familiar with `ListView`, there are several important differences between `ListView` and `RecyclerView`:

- `RecyclerView` is slightly more complex to use: you have to write more code to use `RecyclerView` compared to `ListView`.

- `RecyclerView` does not provide a predefined adapter; you must implement the adapter code that accesses your data source. However, Android includes several predefined adapters that work with `ListView` and `GridView`.

- `RecyclerView` does not offer an item-click event when a user taps an item; instead, item-click events are handled by helper classes. By contrast, `ListView` offers an item-click event.

- `RecyclerView` enhances performance by recycling views and by enforcing the view-holder pattern, which eliminates unnecessary layout resource lookups. Use of the view-holder pattern is optional in `ListView`.

- `RecyclerView` is based on a modular design that makes it easier to customize. For example, you can plug in a different layout policy without significant code changes to your app. By contrast, `ListView` is relatively monolithic in structure.

- `RecyclerView` includes built-in animations for item add and remove. `ListView` animations require some additional effort on the part of the app developer.

Sections

RecyclerView Parts and Functionality

This topic explains how the `Adapter`, `LayoutManager`, and `ViewHolder` work together as helper classes to support `RecyclerView`. It provides a high-level overview of each of these helper classes and explains how you use them in your app.

A Basic RecyclerView Example

This topic builds on the information provided in RecyclerView Parts and Functionality by providing real code examples of how the various `RecyclerView` elements are implemented to build a real-world photo-browsing app.

Extending the RecyclerView Example

This topic adds additional code to the example app presented in A Basic RecyclerView Example to demonstrate how to handle item-click events and update `RecyclerView` when the underlying data source changes.

Summary

This guide introduced the Android `RecyclerView` widget; it explained how to add the `RecyclerView` support library to Xamarin.Android projects, how `RecyclerView` recycles views, how it enforces the view-holder pattern for efficiency, and how the various helper classes that make up `RecyclerView` collaborate to display collections. It provided example code to demonstrate how `RecyclerView` is integrated into an application, it explained how to tailor `RecyclerView`'s layout policy by plugging in different layout managers, and it described how to handle item click events and notify `RecyclerView` of data source changes.

For more information about `RecyclerView`, see the RecyclerView class reference.

Related Links

- RecyclerViewer (sample)
- Introduction to Lollipop
- RecyclerView

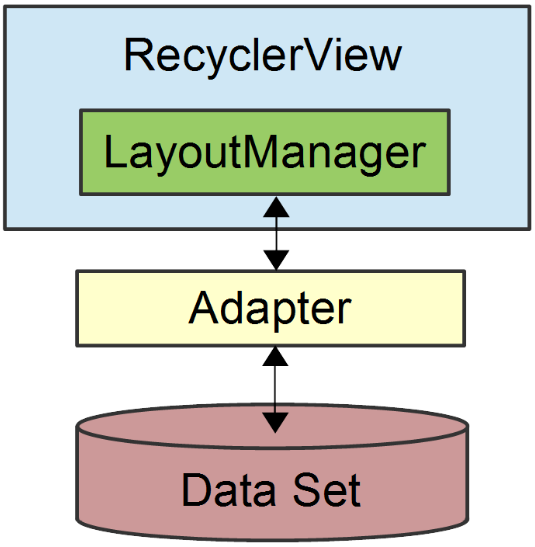
RecyclerView Parts and Functionality

7/8/2021 • 8 minutes to read • Edit Online

`RecyclerView` handles some tasks internally (such as the scrolling and recycling of views), but it is essentially a manager that coordinates helper classes to display a collection. `RecyclerView` delegates tasks to the following helper classes:

- `Adapter` – Inflates item layouts (instantiates the contents of a layout file) and binds data to views that are displayed within a `RecyclerView`. The adapter also reports item-click events.
- `LayoutManager` – Measures and positions item views within a `RecyclerView` and manages the policy for view recycling.
- `ViewHolder` – Looks up and stores view references. The view holder also helps with detecting item-view clicks.
- `ItemDecoration` – Allows an app to add special drawing and layout offsets to specific views for drawing dividers between items, highlights, and visual grouping boundaries.
- `ItemAnimator` – Defines the animations that take place during item actions or as changes are made to the adapter.

The relationship between the `RecyclerView`, `LayoutManager`, and `Adapter` classes is depicted in the following diagram:



As this figure illustrates, the `LayoutManager` can be thought of as the intermediary between the `Adapter` and the `RecyclerView`. The `LayoutManager` makes calls into `Adapter` methods on behalf of the `RecyclerView`. For example, the `LayoutManager` calls an `Adapter` method when it is time to create a new view for a particular item position in the `RecyclerView`. The `Adapter` inflates the layout for that item and creates a `ViewHolder` instance (not shown) to cache references to the views at that position. When the `LayoutManager` calls the `Adapter` to bind a particular item to the data set, the `Adapter` locates the data for that item, retrieves it from the data set, and copies it to the associated item view.

When using `RecyclerView` in your app, creating derived types of the following classes is required:

- `RecyclerView.Adapter` – Provides a binding from your app's data set (which is specific to your app) to item views that are displayed within the `RecyclerView`. The adapter knows how to associate each item-view position in the `RecyclerView` to a specific location in the data source. In addition, the adapter handles the layout of the contents within each individual item view and creates the view holder for each view. The adapter also reports item-click events that are detected by the item view.

- `RecyclerView.ViewHolder` – Caches references to the views in your item layout file so that resource lookups are not repeated unnecessarily. The view holder also arranges for item-click events to be forwarded to the adapter when a user taps the view-holder's associated item view.

- `RecyclerView.LayoutManager` – Positions items within the `RecyclerView`. You can use one of several predefined layout managers or you can implement your own custom layout manager. `RecyclerView` delegates the layout policy to the layout manager, so you can plug in a different layout manager without having to make significant changes to your app.

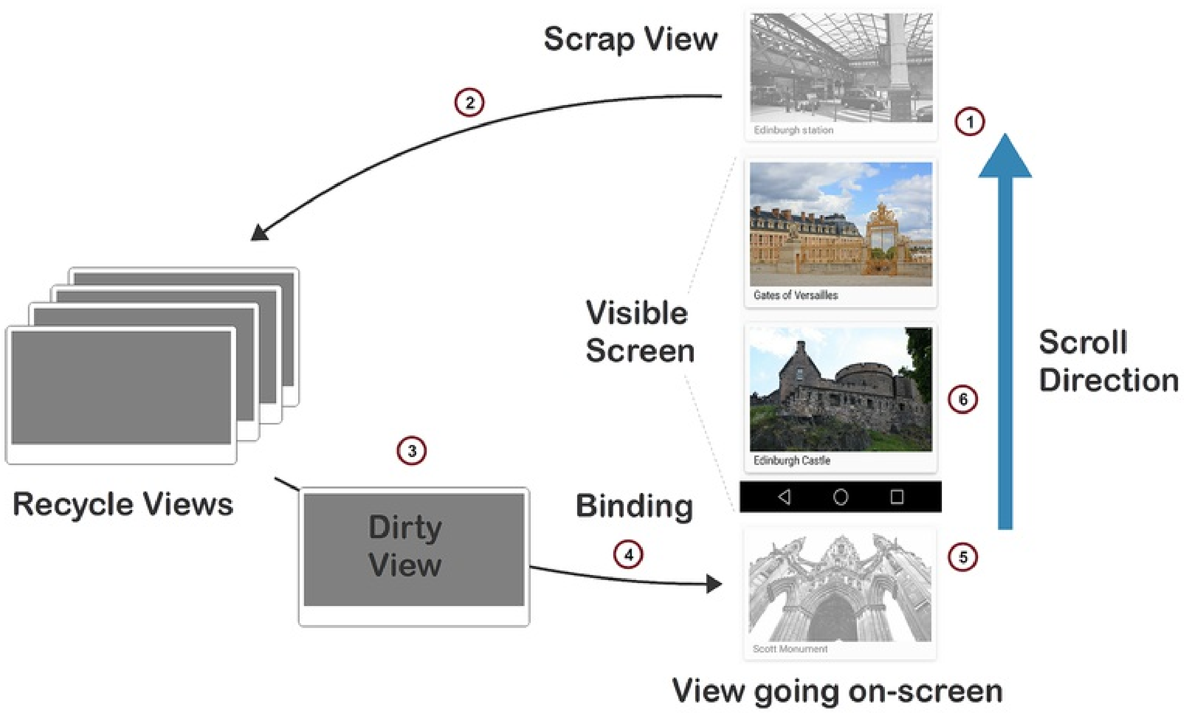
Also, you can optionally extend the following classes to change the look and feel of `RecyclerView` in your app:

- `RecyclerView.ItemDecoration`
- `RecyclerView.ItemAnimator`

If you do not extend `ItemDecoration` and `ItemAnimator`, `RecyclerView` uses default implementations. This guide does not explain how to create custom `ItemDecoration` and `ItemAnimator` classes; for more information about these classes, see RecyclerView.ItemDecoration and RecyclerView.ItemAnimator.

How View Recycling Works

`RecyclerView` does not allocate an item view for every item in your data source. Instead, it allocates only the number of item views that fit on the screen and it reuses those item layouts as the user scrolls. When the view first scrolls out of sight, it goes through the recycling process illustrated in the following figure:



1. When a view scrolls out of sight and is no longer displayed, it becomes a *scrap view*.

2. The scrap view is placed in a pool and becomes a *recycle view*. This pool is a cache of views that display the same type of data.

3. When a new item is to be displayed, a view is taken from the recycle pool for reuse. Because this view

must be re-bound by the adapter before being displayed, it is called a *dirty view*.

4. The dirty view is recycled: the adapter locates the data for the next item to be displayed and copies this data to the views for this item. References for these views are retrieved from the view holder associated with the recycled view.

5. The recycled view is added to the list of items in the `RecyclerView` that are about to go on-screen.

6. The recycled view goes on-screen as the user scrolls the `RecyclerView` to the next item in the list. Meanwhile, another view scrolls out of sight and is recycled according to the above steps.

In addition to item-view reuse, `RecyclerView` also uses another efficiency optimization: view holders. A *view holder* is a simple class that caches view references. Each time the adapter inflates an item-layout file, it also creates a corresponding view holder. The view holder uses `FindViewById` to get references to the views inside the inflated item-layout file. These references are used to load new data into the views every time the layout is recycled to show new data.

The Layout Manager

The layout manager is responsible for positioning items in the `RecyclerView` display; it determines the presentation type (a list or a grid), the orientation (whether items are displayed vertically or horizontally), and which direction items should be displayed (in normal order or in reverse order). The layout manager is also responsible for calculating the size and position of each item in the **RecycleView** display.

The layout manager has an additional purpose: it determines the policy for when to recycle item views that are no longer visible to the user. Because the layout manager is aware of which views are visible (and which are not), it is in the best position to decide when a view can be recycled. To recycle a view, the layout manager typically makes calls to the adapter to replace the contents of a recycled view with different data, as described previously in How View Recycling Works.

You can extend `RecyclerView.LayoutManager` to create your own layout manager, or you can use a predefined layout manager. `RecyclerView` provides the following predefined layout managers:

- `LinearLayoutManager` – Arranges items in a column that can be scrolled vertically, or in a row that can be scrolled horizontally.
- `GridLayoutManager` – Displays items in a grid.
- `StaggeredGridLayoutManager` – Displays items in a staggered grid, where some items have different heights and widths.

To specify the layout manager, instantiate your chosen layout manager and pass it to the `SetLayoutManager` method. Note that you *must* specify the layout manager – `RecyclerView` does not select a predefined layout manager by default.

For more information about the layout manager, see the RecyclerView.LayoutManager class reference.

The View Holder

The view holder is a class that you define for caching view references. The adapter uses these view references to bind each view to its content. Every item in the `RecyclerView` has an associated view holder instance that caches the view references for that item. To create a view holder, use the following steps to define a class to hold the exact set of views per item:

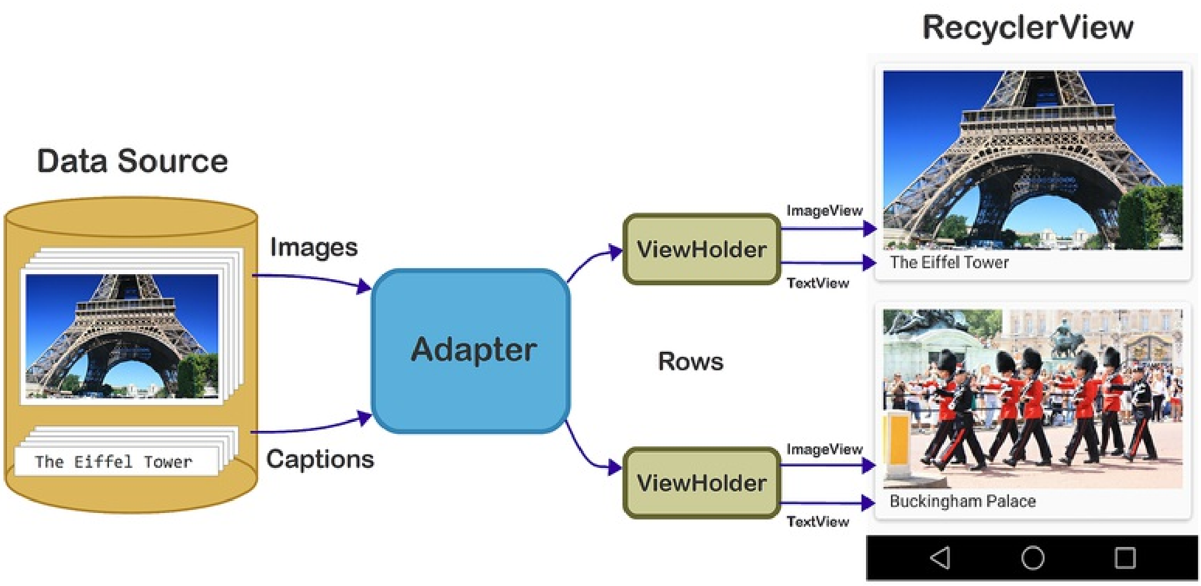
1. Subclass `RecyclerView.ViewHolder`.
2. Implement a constructor that looks up and stores the view references.
3. Implement properties that the adapter can use to access these references.

A detailed example of a `ViewHolder` implementation is presented in A Basic RecyclerView Example. For more information about `RecyclerView.ViewHolder`, see the RecyclerView.ViewHolder class reference.

The Adapter

Most of the "heavy-lifting" of the `RecyclerView` integration code takes place in the adapter. `RecyclerView` requires that you provide an adapter derived from `RecyclerView.Adapter` to access your data source and populate each item with content from the data source. Because the data source is app-specific, you must implement adapter functionality that understands how to access your data. The adapter extracts information from the data source and loads it into each item in the `RecyclerView` collection.

The following drawing illustrates how the adapter maps content in a data source through view holders to individual views within each row item in the `RecyclerView`:



The adapter loads each `RecyclerView` row with data for a particular row item. For row position *P*, for example, the adapter locates the associated data at position *P* within the data source and copies this data to the row item at position *P* in the `RecyclerView` collection. In the above drawing, for example, the adapter uses the view holder to lookup the references for the `ImageView` and `TextView` at that position so it doesn't have to repeatedly call `FindViewById` for those views as the user scrolls through the collection and reuses views.

When you implement an adapter, you must override the following `RecyclerView.Adapter` methods:

- `OnCreateViewHolder` – Instantiates the item layout file and view holder.
- `OnBindViewHolder` – Loads the data at the specified position into the views whose references are stored in the given view holder.
- `ItemCount` – Returns the number of items in the data source.

The layout manager calls these methods while it is positioning items within the `RecyclerView`.

Notifying RecyclerView of Data Changes

`RecyclerView` does not automatically update its display when the contents of its data source changes; the adapter must notify `RecyclerView` when there is a change in the data set. The data set can change in many ways; for example, the contents within an item can change or the overall structure of the data may be altered. `RecyclerView.Adapter` provides a number of methods that you can call so that `RecyclerView` responds to data changes in the most efficient manner:

- `NotifyItemChanged` – Signals that the item at the specified position has changed.
- `NotifyItemRangeChanged` – Signals that the items in the specified range of positions have changed.
- `NotifyItemInserted` – Signals that the item in the specified position has been newly inserted.
- `NotifyItemRangeInserted` – Signals that the items in the specified range of positions have been newly inserted.
- `NotifyItemRemoved` – Signals that the item in the specified position has been removed.
- `NotifyItemRangeRemoved` – Signals that the items in the specified range of positions have been removed.
- `NotifyDataSetChanged` – Signals that the data set has changed (forces a full update).

If you know exactly how your data set has changed, you can call the appropriate methods above to refresh `RecyclerView` in the most efficient manner. If you do not know exactly how your data set has changed, you can call `NotifyDataSetChanged`, which is far less efficient because `RecyclerView` must refresh all the views that are visible to the user. For more information about these methods, see RecyclerView.Adapter.

In the next topic, A Basic RecyclerView Example, an example app is implemented to demonstrate real code examples of the parts and functionality outlined above.

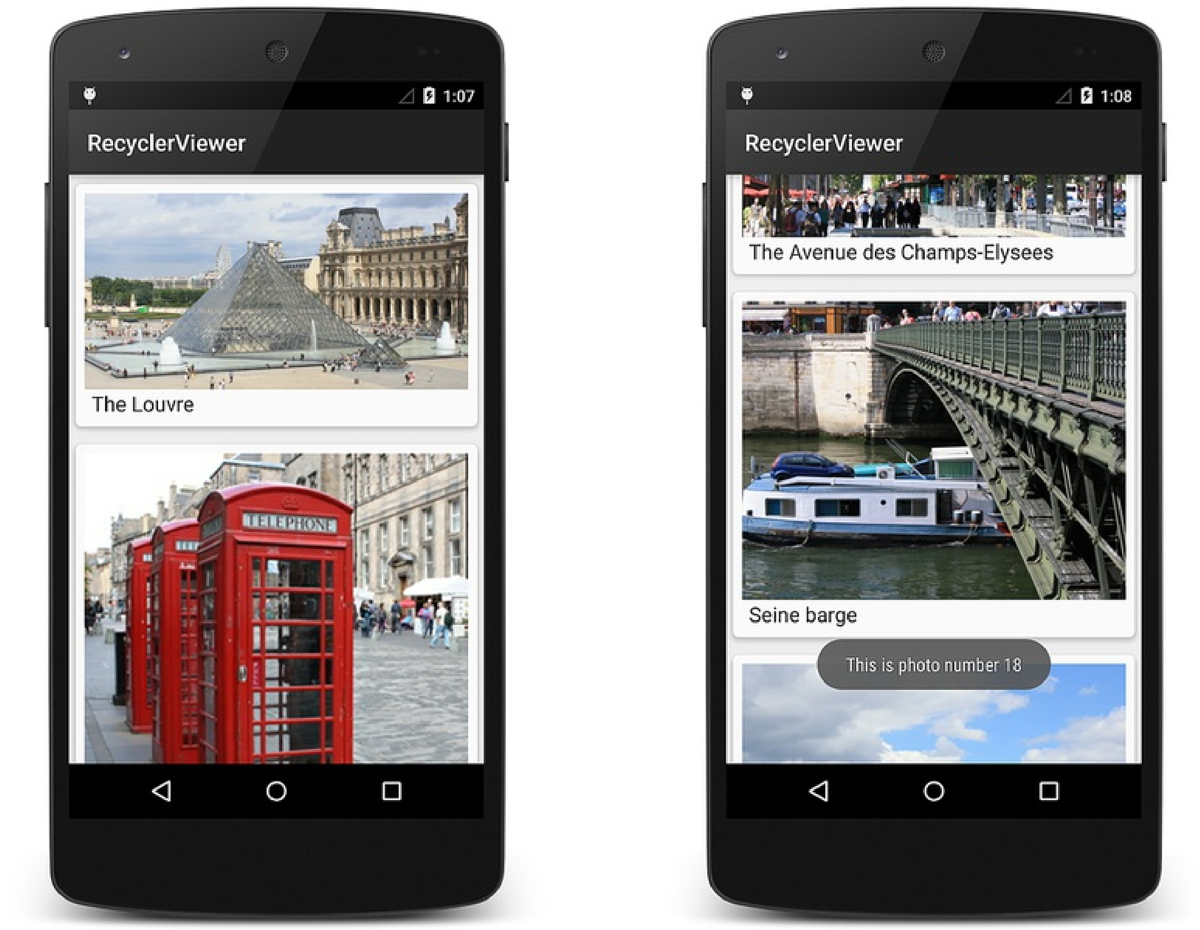
Related Links

- RecyclerView
- A Basic RecyclerView Example
- Extending the RecyclerView Example
- RecyclerView

A Basic RecyclerView Example

7/8/2021 • 10 minutes to read • Edit Online

To understand how `RecyclerView` works in a typical application, this topic explores the RecyclerViewer sample app, a simple code example that uses `RecyclerView` to display a large collection of photos:



RecyclerViewer uses CardView to implement each photograph item in the `RecyclerView` layout. Because of `RecyclerView`'s performance advantages, this sample app is able to quickly scroll through a large collection of photos smoothly and without noticeable delays.

An Example Data Source

In this example app, a "photo album" data source (represented by the `PhotoAlbum` class) supplies `RecyclerView` with item content. `PhotoAlbum` is a collection of photos with captions; when you instantiate it, you get a ready-made collection of 32 photos:

```
PhotoAlbum mPhotoAlbum = new PhotoAlbum ();
```

Each photo instance in `PhotoAlbum` exposes properties that allow you to read its image resource ID, `PhotoID`, and its caption string, `Caption`. The collection of photos is organized such that each photo can be accessed by an indexer. For example, the following lines of code access the image resource ID and caption for the tenth photo in the collection:

```
int imageId = mPhotoAlbum[9].ImageId;
string caption = mPhotoAlbum[9].Caption;
```

`PhotoAlbum` also provides a `RandomSwap` method that you can call to swap the first photo in the collection with a randomly-chosen photo elsewhere in the collection:

```
mPhotoAlbum.RandomSwap ();
```

Because the implementation details of `PhotoAlbum` are not relevant to understanding `RecyclerView`, the `PhotoAlbum` source code is not presented here. The source code to `PhotoAlbum` is available at PhotoAlbum.cs in the RecyclerViewer sample app.

Layout and Initialization

The layout file, **Main.axml**, consists of a single `RecyclerView` within a `LinearLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <android.support.v7.widget.RecyclerView
        android:id="@+id/recyclerView"
        android:scrollbars="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />
</LinearLayout>
```

Note that you must use the fully-qualified name **android.support.v7.widget.RecyclerView** because `RecyclerView` is packaged in a support library. The `OnCreate` method of `MainActivity` initializes this layout, instantiates the adapter, and prepares the underlying data source:

```
public class MainActivity : Activity
{
    RecyclerView mRecyclerView;
    RecyclerView.LayoutManager mLayoutManager;
    PhotoAlbumAdapter mAdapter;
    PhotoAlbum mPhotoAlbum;

    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);

        // Prepare the data source:
        mPhotoAlbum = new PhotoAlbum ();

        // Instantiate the adapter and pass in its data source:
        mAdapter = new PhotoAlbumAdapter (mPhotoAlbum);

        // Set our view from the "main" layout resource:
        SetContentView (Resource.Layout.Main);

        // Get our RecyclerView layout:
        mRecyclerView = FindViewById<RecyclerView> (Resource.Id.recyclerView);

        // Plug the adapter into the RecyclerView:
        mRecyclerView.SetAdapter (mAdapter);
```

This code does the following:

1. Instantiates the `PhotoAlbum` data source.

2. Passes the photo album data source to the constructor of the adapter, `PhotoAlbumAdapter` (which is defined later in this guide). Note that it is considered a best practice to pass the data source as a parameter to the constructor of the adapter.

3. Gets the `RecyclerView` from the layout.

4. Plugs the adapter into the `RecyclerView` instance by calling the `RecyclerView` `SetAdapter` method as shown above.

Layout Manager

Each item in the `RecyclerView` is made up of a `CardView` that contains a photo image and photo caption (details are covered in the View Holder section below). The predefined `LinearLayoutManager` is used to lay out each `CardView` in a vertical scrolling arrangement:

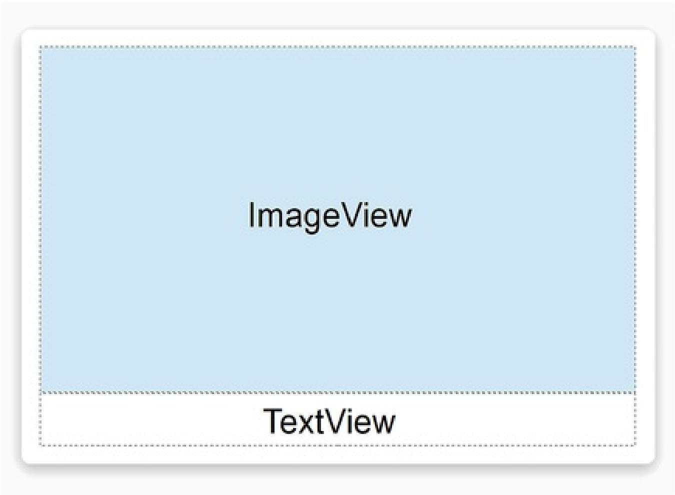
```
mLayoutManager = new LinearLayoutManager (this);
mRecyclerView.SetLayoutManager (mLayoutManager);
```

This code resides in the main activity's `OnCreate` method. The constructor to the layout manager requires a *context*, so the `MainActivity` is passed using `this` as seen above.

Instead of using the predefined `LinearLayoutManager`, you can plug in a custom layout manager that displays two `CardView` items side-by-side, implementing a page-turning animation effect to traverse through the collection of photos. Later in this guide, you will see an example of how to modify the layout by swapping in a different layout manager.

View Holder

The view holder class is called `PhotoViewHolder`. Each `PhotoViewHolder` instance holds references to the `ImageView` and `TextView` of an associated row item, which is laid out in a `CardView` as diagrammed here:



`PhotoViewHolder` derives from `RecyclerView.ViewHolder` and contains properties to store references to the `ImageView` and `TextView` shown in the above layout. `PhotoViewHolder` consists of two properties and one constructor:

```
public class PhotoViewHolder : RecyclerView.ViewHolder
{
    public ImageView Image { get; private set; }
    public TextView Caption { get; private set; }

    public PhotoViewHolder (View itemView) : base (itemView)
    {
        // Locate and cache view references:
        Image = itemView.FindViewById<ImageView> (Resource.Id.imageView);
        Caption = itemView.FindViewById<TextView> (Resource.Id.textView);
    }
}
```

In this code example, the `PhotoViewHolder` constructor is passed a reference to the parent item view (the `CardView`) that `PhotoViewHolder` wraps. Note that you always forward the parent item view to the base constructor. The `PhotoViewHolder` constructor calls `FindViewById` on the parent item view to locate each of its child view references, `ImageView` and `TextView`, storing the results in the `Image` and `Caption` properties, respectively. The adapter later retrieves view references from these properties when it updates this `CardView`'s child views with new data.

For more information about `RecyclerView.ViewHolder`, see the RecyclerView.ViewHolder class reference.

Adapter

The adapter loads each `RecyclerView` row with data for a particular photograph. For a given photograph at row position *P*, for example, the adapter locates the associated data at position *P* within the data source and copies this data to the row item at position *P* in the `RecyclerView` collection. The adapter uses the view holder to lookup the references for the `ImageView` and `TextView` at that position so it doesn't have to repeatedly call `FindViewById` for those views as the user scrolls through the photograph collection and reuses views.

In **RecyclerViewer**, an adapter class is derived from `RecyclerView.Adapter` to create `PhotoAlbumAdapter`:

```
public class PhotoAlbumAdapter : RecyclerView.Adapter
{
    public PhotoAlbum mPhotoAlbum;

    public PhotoAlbumAdapter (PhotoAlbum photoAlbum)
    {
        mPhotoAlbum = photoAlbum;
    }
    ...
}
```

The `mPhotoAlbum` member contains the data source (the photo album) that is passed into the constructor; the constructor copies the photo album into this member variable. The following required `RecyclerView.Adapter` methods are implemented:

- `OnCreateViewHolder` – Instantiates the item layout file and view holder.
- `OnBindViewHolder` – Loads the data at the specified position into the views whose references are stored in the given view holder.
- `ItemCount` – Returns the number of items in the data source.

The layout manager calls these methods while it is positioning items within the `RecyclerView`. The implementation of these methods is examined in the following sections.

OnCreateViewHolder

The layout manager calls `OnCreateViewHolder` when the `RecyclerView` needs a new view holder to represent an item. `OnCreateViewHolder` inflates the item view from the view's layout file and wraps the view in a new `PhotoViewHolder` instance. The `PhotoViewHolder` constructor locates and stores references to child views in the layout as described previously in View Holder.

Each row item is represented by a `CardView` that contains an `ImageView` (for the photo) and a `TextView` (for the caption). This layout resides in the file **PhotoCardView.axml**:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:card_view="http://schemas.android.com/apk/res-auto"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
    <android.support.v7.widget.CardView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        card_view:cardElevation="4dp"
        card_view:cardUseCompatPadding="true"
        card_view:cardCornerRadius="5dp">
        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical"
            android:padding="8dp">
            <ImageView
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:id="@+id/imageView"
                android:scaleType="centerCrop" />
            <TextView
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:textAppearance="?android:attr/textAppearanceMedium"
                android:textColor="#333333"
                android:text="Caption"
                android:id="@+id/textView"
                android:layout_gravity="center_horizontal"
                android:layout_marginLeft="4dp" />
        </LinearLayout>
    </android.support.v7.widget.CardView>
</FrameLayout>
```

This layout represents a single row item in the `RecyclerView`. The `OnBindViewHolder` method (described below) copies data from the data source into the `ImageView` and `TextView` of this layout. `OnCreateViewHolder` inflates this layout for a given photo location in the `RecyclerView` and instantiates a new `PhotoViewHolder` instance (which locates and caches references to the `ImageView` and `TextView` child views in the associated `CardView` layout):

```
public override RecyclerView.ViewHolder
    OnCreateViewHolder (ViewGroup parent, int viewType)
{
    // Inflate the CardView for the photo:
    View itemView = LayoutInflater.From (parent.Context).
                Inflate (Resource.Layout.PhotoCardView, parent, false);

    // Create a ViewHolder to hold view references inside the CardView:
    PhotoViewHolder vh = new PhotoViewHolder (itemView);
    return vh;
}
```

The resulting view holder instance, `vh`, is returned back to the caller (the layout manager).

OnBindViewHolder

When the layout manager is ready to display a particular view in the `RecyclerView`'s visible screen area, it calls the adapter's `OnBindViewHolder` method to fill the item at the specified row position with content from the data source. `OnBindViewHolder` gets the photo information for the specified row position (the photo's image resource and the string for the photo's caption) and copies this data to the associated views. Views are located via references stored in the view holder object (which is passed in through the `holder` parameter):

```
public override void
    OnBindViewHolder (RecyclerView.ViewHolder holder, int position)
{
    PhotoViewHolder vh = holder as PhotoViewHolder;

    // Load the photo image resource from the photo album:
    vh.Image.SetImageResource (mPhotoAlbum[position].PhotoID);

    // Load the photo caption from the photo album:
    vh.Caption.Text = mPhotoAlbum[position].Caption;
}
```

The passed-in view holder object must first be cast into the derived view holder type (in this case, `PhotoViewHolder`) before it is used. The adapter loads the image resource into the view referenced by the view holder's `Image` property, and it copies the caption text into the view referenced by the view holder's `Caption` property. This *binds* the associated view with its data.

Notice that `OnBindViewHolder` is the code that deals directly with the structure of the data. In this case, `OnBindViewHolder` understands how to map the `RecyclerView` item position to its associated data item in the data source. The mapping is straightforward in this case because the position can be used as an array index into the photo album; however, more complex data sources may require extra code to establish such a mapping.

ItemCount

The `ItemCount` method returns the number of items in the data collection. In the example photo viewer app, the item count is the number of photos in the photo album:

```
public override int ItemCount
{
    get { return mPhotoAlbum.NumPhotos; }
}
```

For more information about `RecyclerView.Adapter`, see the RecyclerView.Adapter class reference.

Putting it All Together

The resulting `RecyclerView` implementation for the example photo app consists of `MainActivity` code that creates the data source, layout manager and the adapter. `MainActivity` creates the `mRecyclerView` instance, instantiates the data source and the adapter, and plugs in the layout manager and adapter:

```
public class MainActivity : Activity
{
    RecyclerView mRecyclerView;
    RecyclerView.LayoutManager mLayoutManager;
    PhotoAlbumAdapter mAdapter;
    PhotoAlbum mPhotoAlbum;

    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);
        mPhotoAlbum = new PhotoAlbum();
        SetContentView (Resource.Layout.Main);
        mRecyclerView = FindViewById<RecyclerView> (Resource.Id.recyclerView);

        // Plug in the linear layout manager:
        mLayoutManager = new LinearLayoutManager (this);
        mRecyclerView.SetLayoutManager (mLayoutManager);

        // Plug in my adapter:
        mAdapter = new PhotoAlbumAdapter (mPhotoAlbum);
        mRecyclerView.SetAdapter (mAdapter);
    }
}
```

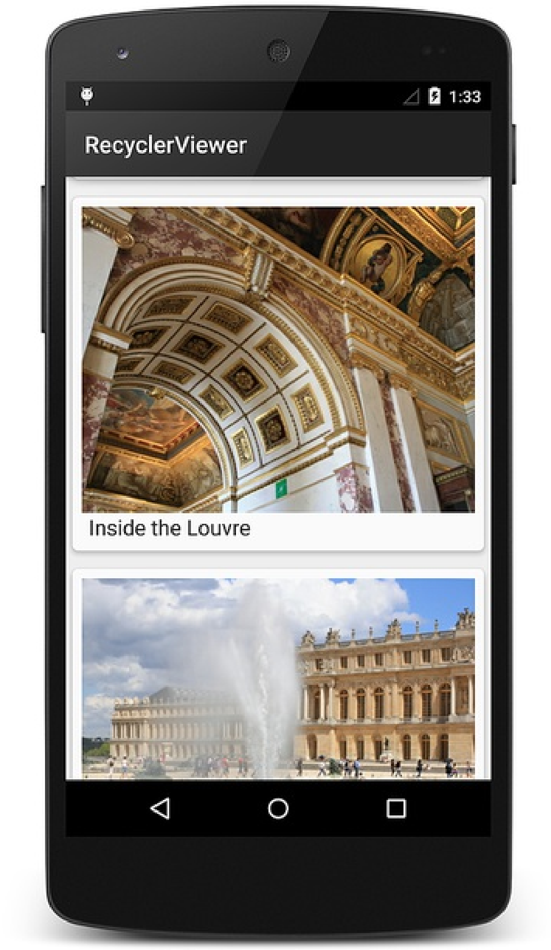
`PhotoViewHolder` locates and caches the view references:

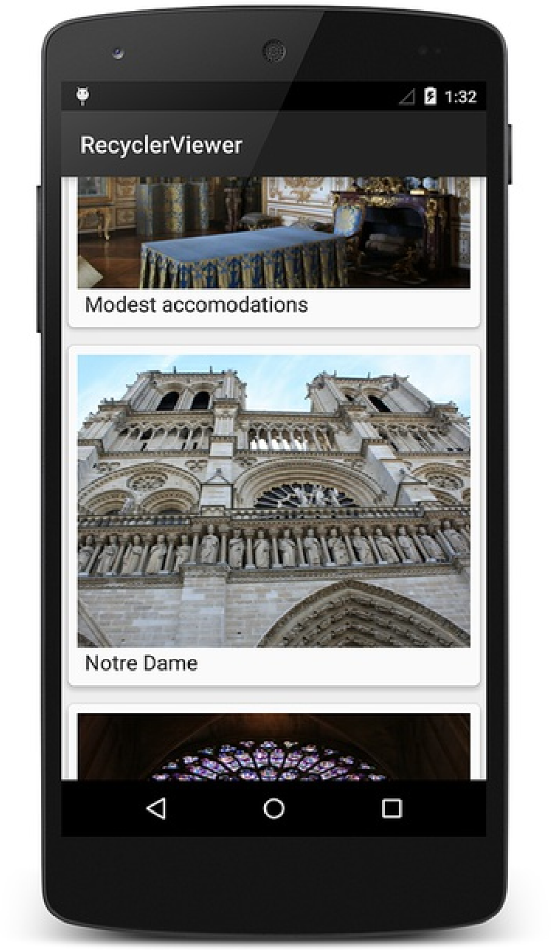
```
public class PhotoViewHolder : RecyclerView.ViewHolder
{
    public ImageView Image { get; private set; }
    public TextView Caption { get; private set; }

    public PhotoViewHolder (View itemView) : base (itemView)
    {
        // Locate and cache view references:
        Image = itemView.FindViewById<ImageView> (Resource.Id.imageView);
        Caption = itemView.FindViewById<TextView> (Resource.Id.textView);
    }
}
```

`PhotoAlbumAdapter` implements the three required method overrides:

```
public class PhotoAlbumAdapter : RecyclerView.Adapter
{
    public PhotoAlbum mPhotoAlbum;
    public PhotoAlbumAdapter (PhotoAlbum photoAlbum)
    {
        mPhotoAlbum = photoAlbum;
    }

    public override RecyclerView.ViewHolder
        OnCreateViewHolder (ViewGroup parent, int viewType)
    {
        View itemView = LayoutInflater.From (parent.Context).
                    Inflate (Resource.Layout.PhotoCardView, parent, false);
        PhotoViewHolder vh = new PhotoViewHolder (itemView);
        return vh;
    }

    public override void
        OnBindViewHolder (RecyclerView.ViewHolder holder, int position)
    {
        PhotoViewHolder vh = holder as PhotoViewHolder;
        vh.Image.SetImageResource (mPhotoAlbum[position].PhotoID);
        vh.Caption.Text = mPhotoAlbum[position].Caption;
    }

    public override int ItemCount
    {
        get { return mPhotoAlbum.NumPhotos; }
    }
}
```

When this code is compiled and run, it creates the basic photo viewing app as shown in the following screenshots:





If shadows are not being drawn (as seen in the above screenshot), edit **Properties/AndroidManifest.xml** and add the following attribute setting to the `<application>` element:

```
android:hardwareAccelerated="true"
```

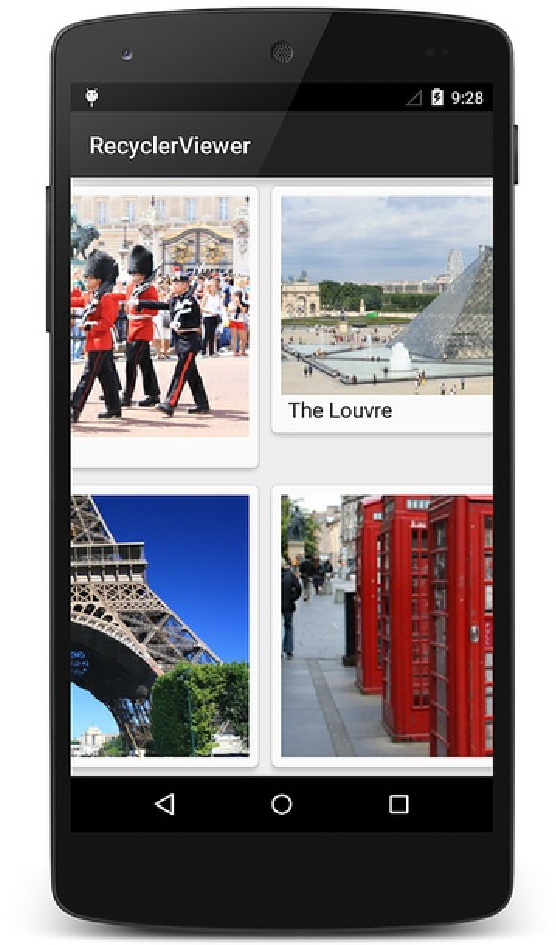
This basic app only supports browsing of the photo album. It does not respond to item-touch events, nor does it handle changes in the underlying data. This functionality is added in Extending the RecyclerView Example.

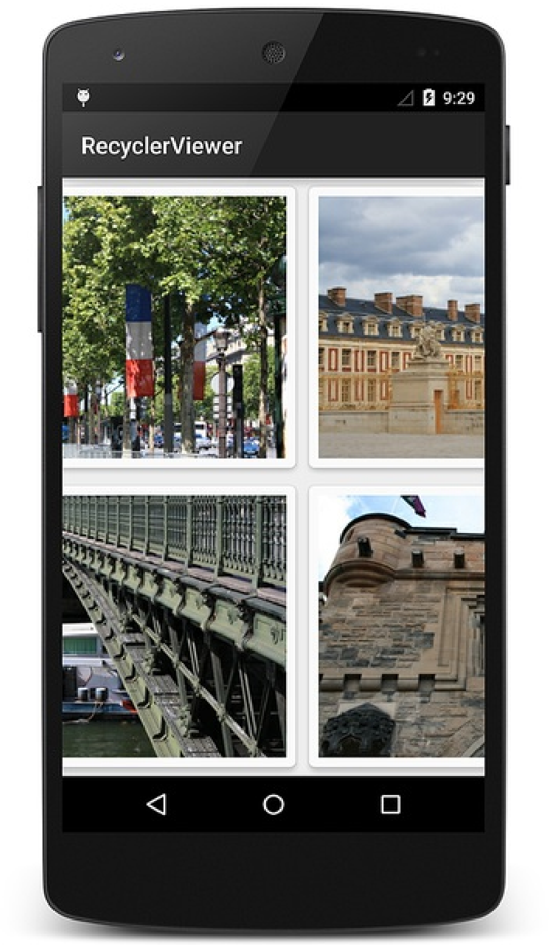
Changing the LayoutManager

Because of `RecyclerView`'s flexibility, it's easy to modify the app to use a different layout manager. In the following example, it is modified to display the photo album with a grid layout that scrolls horizontally rather than with a vertical linear layout. To do this, the layout manager instantiation is modified to use the `GridLayoutManager` as follows:

```
mLayoutManager = new GridLayoutManager(this, 2, GridLayoutManager.Horizontal, false);
```

This code change replaces the vertical `LinearLayoutManager` with a `GridLayoutManager` that presents a grid made up of two rows that scroll in the horizontal direction. When you compile and run the app again, you'll see that the photographs are displayed in a grid and that scrolling is horizontal rather than vertical:





By changing only one line of code, it is possible to modify the photo-viewing app to use a different layout with different behavior. Notice that neither the adapter code nor the layout XML had to be modified to change the layout style.

In the next topic, [Extending the RecyclerView Example], this basic sample app is extended to handle item-click events and update `RecyclerView` when the underlying data source changes.

Related Links

- RecyclerViewer (sample)
- RecyclerView
- RecyclerView Parts and Functionality
- Extending the RecyclerView Example
- RecyclerView

Extending the RecyclerView Example

7/8/2021 • 5 minutes to read • Edit Online

The basic app described in A Basic RecyclerView Example actually doesn't do much – it simply scrolls and displays a fixed list of photograph items to facilitate browsing. In real-world applications, users expect to be able to interact with the app by tapping items in the display. Also, the underlying data source can change (or be changed by the app), and the contents of the display must remain consistent with these changes. In the following sections, you'll learn how to handle item-click events and update `RecyclerView` when the underlying data source changes.

Handling Item-Click Events

When a user touches an item in the `RecyclerView`, an item-click event is generated to notify the app as to which item was touched. This event is not generated by `RecyclerView` – instead, the item view (which is wrapped in the view holder) detects touches and reports these touches as click events.

To illustrate how to handle item-click events, the following steps explain how the basic photo-viewing app is modified to report which photograph had been touched by the user. When an item-click event occurs in the sample app, the following sequence takes place:

1. The photograph's `CardView` detects the item-click event and notifies the adapter.
2. The adapter forwards the event (with item position information) to the activity's item-click handler.
3. The activity's item-click handler responds to the item-click event.

First, an event handler member called `ItemClick` is added to the `PhotoAlbumAdapter` class definition:

```
public event EventHandler<int> ItemClick;
```

Next, an item-click event handler method is added to `MainActivity`. This handler briefly displays a toast that indicates which photograph item was touched:

```
void OnItemClick (object sender, int position)
{
    int photoNum = position + 1;
    Toast.MakeText(this, "This is photo number " + photoNum, ToastLength.Short).Show();
}
```

Next, a line of code is needed to register the `OnItemClick` handler with `PhotoAlbumAdapter`. A good place to do this is immediately after `PhotoAlbumAdapter` is created:

```
mAdapter = new PhotoAlbumAdapter (mPhotoAlbum);
mAdapter.ItemClick += OnItemClick;
```

In this basic example, handler registration takes place in the main activity's `OnCreate` method, but a production app might register the handler in `OnResume` and unregister it in `OnPause` – see Activity Lifecycle for more information.

`PhotoAlbumAdapter` will now call `OnItemClick` when it receives an item-click event. The next step is to create a handler in the adapter that raises this `ItemClick` event. The following method, `OnClick`, is added immediately after the adapter's `ItemCount` method:

```
void OnClick (int position)
{
    if (ItemClick != null)
        ItemClick (this, position);
}
```

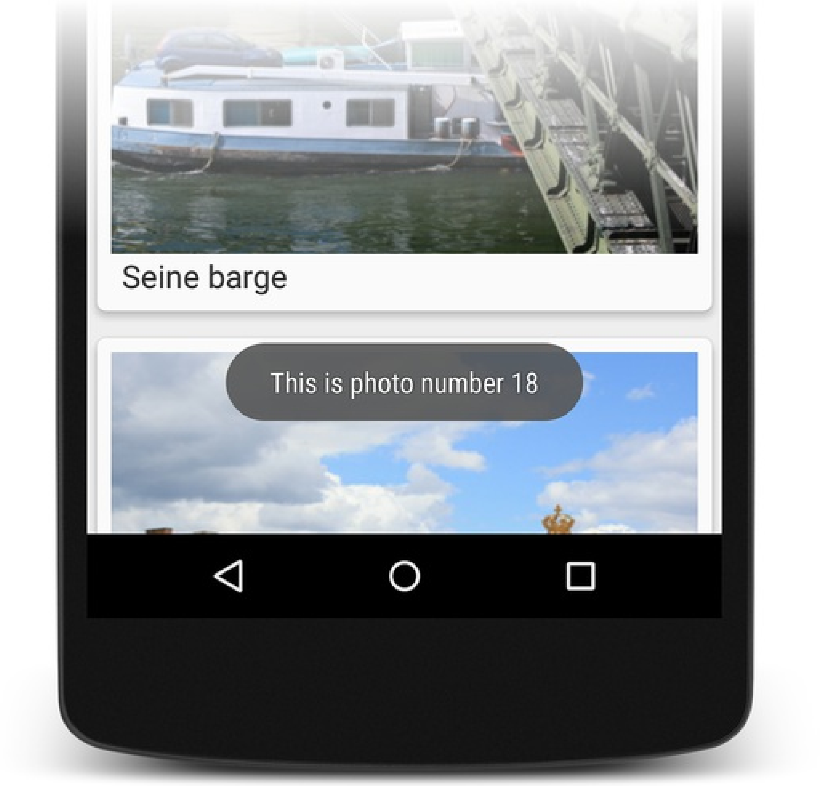
This `OnClick` method is the adapter's *listener* for item-click events from item views. Before this listener can be registered with an item view (via the item view's view holder), the `PhotoViewHolder` constructor must be modified to accept this method as an additional argument, and register `OnClick` with the item view `Click` event. Here's the modified `PhotoViewHolder` constructor:

```
public PhotoViewHolder (View itemView, Action<int> listener)
    : base (itemView)
{
    Image = itemView.FindViewById<ImageView> (Resource.Id.imageView);
    Caption = itemView.FindViewById<TextView> (Resource.Id.textView);

    itemView.Click += (sender, e) => listener (base.LayoutPosition);
}
```

The `itemView` parameter contains a reference to the `CardView` that was touched by the user. Note that the view holder base class knows the layout position of the item (`CardView`) that it represents (via the `LayoutPosition` property), and this position is passed to the adapter's `OnClick` method when an item-click event takes place. The adapter's `OnCreateViewHolder` method is modified to pass the adapter's `OnClick` method to the view-holder's constructor:

```
PhotoViewHolder vh = new PhotoViewHolder (itemView, OnClick);
```

Now when you build and run the sample photo-viewing app, tapping a photo in the display will cause a toast to appear that reports which photograph was touched:



This example demonstrates just one approach for implementing event handlers with `RecyclerView`. Another approach that could be used here is to place events on the view holder and have the adapter subscribe to these events. If the sample photo app provided a photo editing capability, separate events would be required for the `ImageView` and the `TextView` within each `CardView`: touches on the `TextView` would launch an `EditView` dialog that lets the user edit the caption, and touches on the `ImageView` would launch a photo touchup tool that lets the user crop or rotate the photo. Depending on the needs of your app, you must design the best approach for handling and responding to touch events.

To demonstrate how `RecyclerView` can be updated when the data set changes, the sample photo-viewing app can be modified to randomly pick a photo in the data source and swap it with the first photo. First, a **Random Pick** button is added to the example photo app's **Main.axml** layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <Button
        android:id="@+id/randPickButton"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:gravity="center_horizontal"
        android:textAppearance="?android:attr/textAppearanceLarge"
        android:text="Random Pick" />
    <android.support.v7.widget.RecyclerView
        android:id="@+id/recyclerView"
        android:scrollbars="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />
</LinearLayout>
```

Next, code is added at the end of the main activity's `OnCreate` method to locate the `Random Pick` button in the layout and attach a handler to it:

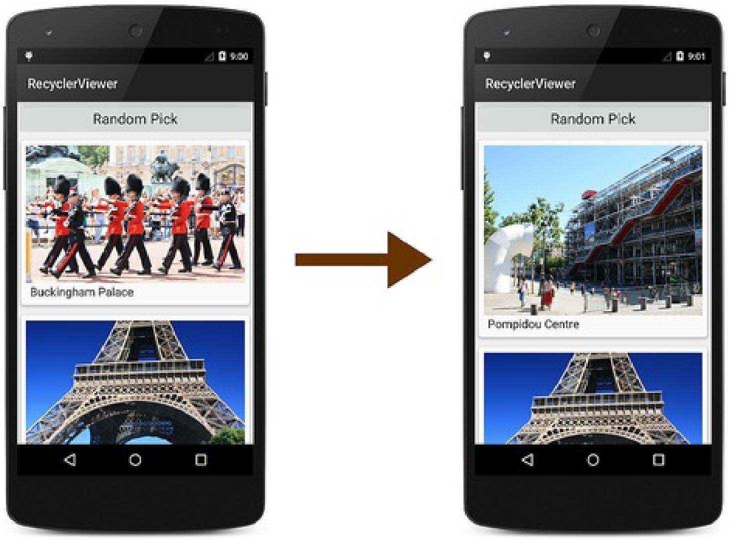
```
Button randomPickBtn = FindViewById<Button>(Resource.Id.randPickButton);

randomPickBtn.Click += delegate
{
    if (mPhotoAlbum != null)
    {
        // Randomly swap a photo with the first photo:
        int idx = mPhotoAlbum.RandomSwap();
    }
};
```

This handler calls the photo album's `RandomSwap` method when the **Random Pick** button is tapped. The `RandomSwap` method randomly swaps a photo with the first photo in the data source, then returns the index of the randomly-swapped photo. When you compile and run the sample app with this code, tapping the **Random Pick** button does not result in a display change because the `RecyclerView` is not aware of the change to the data source.

To keep `RecyclerView` updated after the data source changes, the **Random Pick** click handler must be modified to call the adapter's `NotifyItemChanged` method for each item in the collection that has changed (in this case, two items have changed: the first photo and the swapped photo). This causes `RecyclerView` to update its display so that it is consistent with the new state of the data source:

```
Button randomPickBtn = FindViewById<Button>(Resource.Id.randPickButton);

randomPickBtn.Click += delegate
{
    if (mPhotoAlbum != null)
    {
        int idx = mPhotoAlbum.RandomSwap();

        // First photo has changed:
        mAdapter.NotifyItemChanged(0);

        // Swapped photo has changed:
        mAdapter.NotifyItemChanged(idx);
    }
};
```

Now, when the **Random Pick** button is tapped, `RecyclerView` updates the display to show that a photo further down in the collection has been swapped with the first photo in the collection:



Of course, `NotifyDataSetChanged` could have been called instead of making the two calls to `NotifyItemChanged`, but doing so would force `RecyclerView` to refresh the entire collection even though only two items in the collection had changed. Calling `NotifyItemChanged` is significantly more efficient than calling `NotifyDataSetChanged`.

Related Links

- RecyclerViewer (sample)
- RecyclerView
- RecyclerView Parts and Functionality
- A Basic RecyclerView Example
- RecyclerView

Xamarin.Android ListView

7/8/2021 • 6 minutes to read • Edit Online

ListView is an important UI component of Android applications; it is used everywhere from short lists of menu options to long lists of contacts or internet favorites. It provides a simple way to present a scrolling list of rows that can either be formatted with a built-in style or customized extensively.

Overview

List views and adapters are included in the most fundamental building blocks of Android Applications. The `ListView` class provides a flexible way to present data, whether it is a short menu or a long scrolling list. It provides usability features like fast scrolling, indexes and single or multiple selection to help you build mobile-friendly user interfaces for your applications. A `ListView` instance requires an *Adapter* to feed it with data contained in row views.

This guide explains how to implement `ListView` and the various `Adapter` classes in Xamarin.Android. It also demonstrates how to customize the appearance of a `ListView`, and it discusses the importance of row re-use to reduce memory consumption. There is also some discussion of how the Activity Lifecycle affects `ListView` and `Adapter` use. If you are working on cross-platform applications with Xamarin.iOS, the `ListView` control is structurally similar to the iOS `UITableView` (and the Android `Adapter` is similar to the `UITableViewSource`).

First, a short tutorial introduces the `ListView` with a basic code example. Next, links to more advanced topics are provided to help you use `ListView` in real-world apps.

> **NOTE**
>
> The `RecyclerView` widget is a more advanced and flexible version of `ListView`. Because `RecyclerView` is designed to be the successor to `ListView` (and `GridView`), we recommend that you use `RecyclerView` rather than `ListView` for new app development. For more information, see RecyclerView.

ListView Tutorial

`ListView` is a `ViewGroup` that creates a list of scrollable items. The list items are automatically inserted to the list using a `IListAdapter`.

In this tutorial, you'll create a scrollable list of country names that are read from a string array. When a list item is selected, a toast message will display the position of the item in the list.

Start a new project named **HelloListView**.

Create an XML file named **list_item.xml** and save it inside the **Resources/Layout/** folder. Insert the following:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="10dp"
    android:textSize="16sp">
</TextView>
```

This file defines the layout for each item that will be placed in the `ListView`.

Open `MainActivity.cs` and modify the class to extend `ListActivity` (instead of `Activity`):

```
public class MainActivity : ListActivity
{
```

Insert the following code for the `OnCreate()`) method:

```
protected override void OnCreate (Bundle bundle)
{
    base.OnCreate (bundle);

    ListAdapter = new ArrayAdapter<string> (this, Resource.Layout.list_item, countries);

    ListView.TextFilterEnabled = true;

    ListView.ItemClick += delegate (object sender, AdapterView.ItemClickEventArgs args)
    {
        Toast.MakeText(Application, ((TextView)args.View).Text, ToastLength.Short).Show();
    };
}
```

Notice that this does not load a layout file for the Activity (which you usually do with `SetContentView(int)`)). Instead, setting the `ListAdapter` property automatically adds a `ListView` to fill the entire screen of the `ListActivity`. This method takes an `ArrayAdapter<T>`, which manages the array of list items that will be placed into the `ListView`. The `ArrayAdapter<T>` constructor takes the application `Context`, the layout description for each list item (created in the previous step), and a `T[]` or `Java.Util.IList<T>` array of objects to insert in the `ListView` (defined next).

The `TextFilterEnabled` property turns on text filtering for the `ListView`, so that when the user begins typing, the list will be filtered.

The `ItemClick` event can be used to subscribe handlers for clicks. When an item in the `ListView` is clicked, the handler is called and a `Toast` message is displayed, using the text from the clicked item.

You can use list item designs provided by the platform instead of defining your own layout file for the `ListAdapter`. For example, try using `Android.Resource.Layout.SimpleListItem1` instead of `Resource.Layout.list_item`.

Add the following `using` statement:

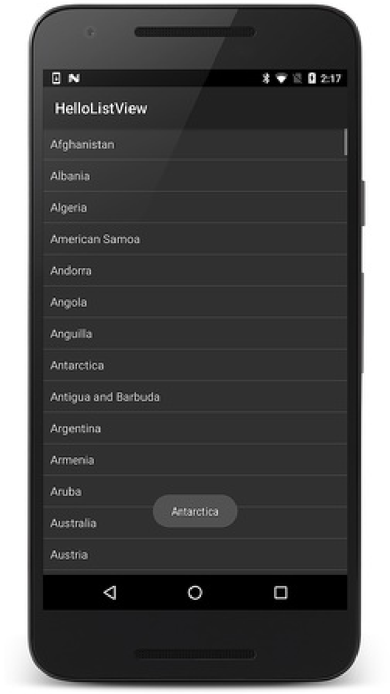
```
using System;
```

Next, add the following string array as a member of `MainActivity`:

```
static readonly string[] countries = new String[] {
    "Afghanistan","Albania","Algeria","American Samoa","Andorra",
    "Angola","Anguilla","Antarctica","Antigua and Barbuda","Argentina",
    "Armenia","Aruba","Australia","Austria","Azerbaijan",
    "Bahrain","Bangladesh","Barbados","Belarus","Belgium",
    "Belize","Benin","Bermuda","Bhutan","Bolivia",
    "Bosnia and Herzegovina","Botswana","Bouvet Island","Brazil","British Indian Ocean Territory",
    "British Virgin Islands","Brunei","Bulgaria","Burkina Faso","Burundi",
    "Cote d'Ivoire","Cambodia","Cameroon","Canada","Cape Verde",
    "Cayman Islands","Central African Republic","Chad","Chile","China",
    "Christmas Island","Cocos (Keeling) Islands","Colombia","Comoros","Congo",
    "Cook Islands","Costa Rica","Croatia","Cuba","Cyprus","Czech Republic",
    "Democratic Republic of the Congo","Denmark","Djibouti","Dominica","Dominican Republic",
    "East Timor","Ecuador","Egypt","El Salvador","Equatorial Guinea","Eritrea",
    "Estonia","Ethiopia","Faeroe Islands","Falkland Islands","Fiji","Finland",
    "Former Yugoslav Republic of Macedonia","France","French Guiana","French Polynesia",
    "French Southern Territories","Gabon","Georgia","Germany","Ghana","Gibraltar",
    "Greece","Greenland","Grenada","Guadeloupe","Guam","Guatemala","Guinea","Guinea-Bissau",
    "Guyana","Haiti","Heard Island and McDonald Islands","Honduras","Hong Kong","Hungary",
    "Iceland","India","Indonesia","Iran","Iraq","Ireland","Israel","Italy","Jamaica",
    "Japan","Jordan","Kazakhstan","Kenya","Kiribati","Kuwait","Kyrgyzstan","Laos",
    "Latvia","Lebanon","Lesotho","Liberia","Libya","Liechtenstein","Lithuania","Luxembourg",
    "Macau","Madagascar","Malawi","Malaysia","Maldives","Mali","Malta","Marshall Islands",
    "Martinique","Mauritania","Mauritius","Mayotte","Mexico","Micronesia","Moldova",
    "Monaco","Mongolia","Montserrat","Morocco","Mozambique","Myanmar","Namibia",
    "Nauru","Nepal","Netherlands","Netherlands Antilles","New Caledonia","New Zealand",
    "Nicaragua","Niger","Nigeria","Niue","Norfolk Island","North Korea","Northern Marianas",
    "Norway","Oman","Pakistan","Palau","Panama","Papua New Guinea","Paraguay","Peru",
    "Philippines","Pitcairn Islands","Poland","Portugal","Puerto Rico","Qatar",
    "Reunion","Romania","Russia","Rwanda","Sqo Tome and Principe","Saint Helena",
    "Saint Kitts and Nevis","Saint Lucia","Saint Pierre and Miquelon",
    "Saint Vincent and the Grenadines","Samoa","San Marino","Saudi Arabia","Senegal",
    "Seychelles","Sierra Leone","Singapore","Slovakia","Slovenia","Solomon Islands",
    "Somalia","South Africa","South Georgia and the South Sandwich Islands","South Korea",
    "Spain","Sri Lanka","Sudan","Suriname","Svalbard and Jan Mayen","Swaziland","Sweden",
    "Switzerland","Syria","Taiwan","Tajikistan","Tanzania","Thailand","The Bahamas",
    "The Gambia","Togo","Tokelau","Tonga","Trinidad and Tobago","Tunisia","Turkey",
    "Turkmenistan","Turks and Caicos Islands","Tuvalu","Virgin Islands","Uganda",
    "Ukraine","United Arab Emirates","United Kingdom",
    "United States","United States Minor Outlying Islands","Uruguay","Uzbekistan",
    "Vanuatu","Vatican City","Venezuela","Vietnam","Wallis and Futuna","Western Sahara",
    "Yemen","Yugoslavia","Zambia","Zimbabwe"
  };
```

This is the array of strings that will be placed into the `ListView`.

Run the application. You can scroll the list, or type to filter it, then click an item to see a message. You should see something like this:



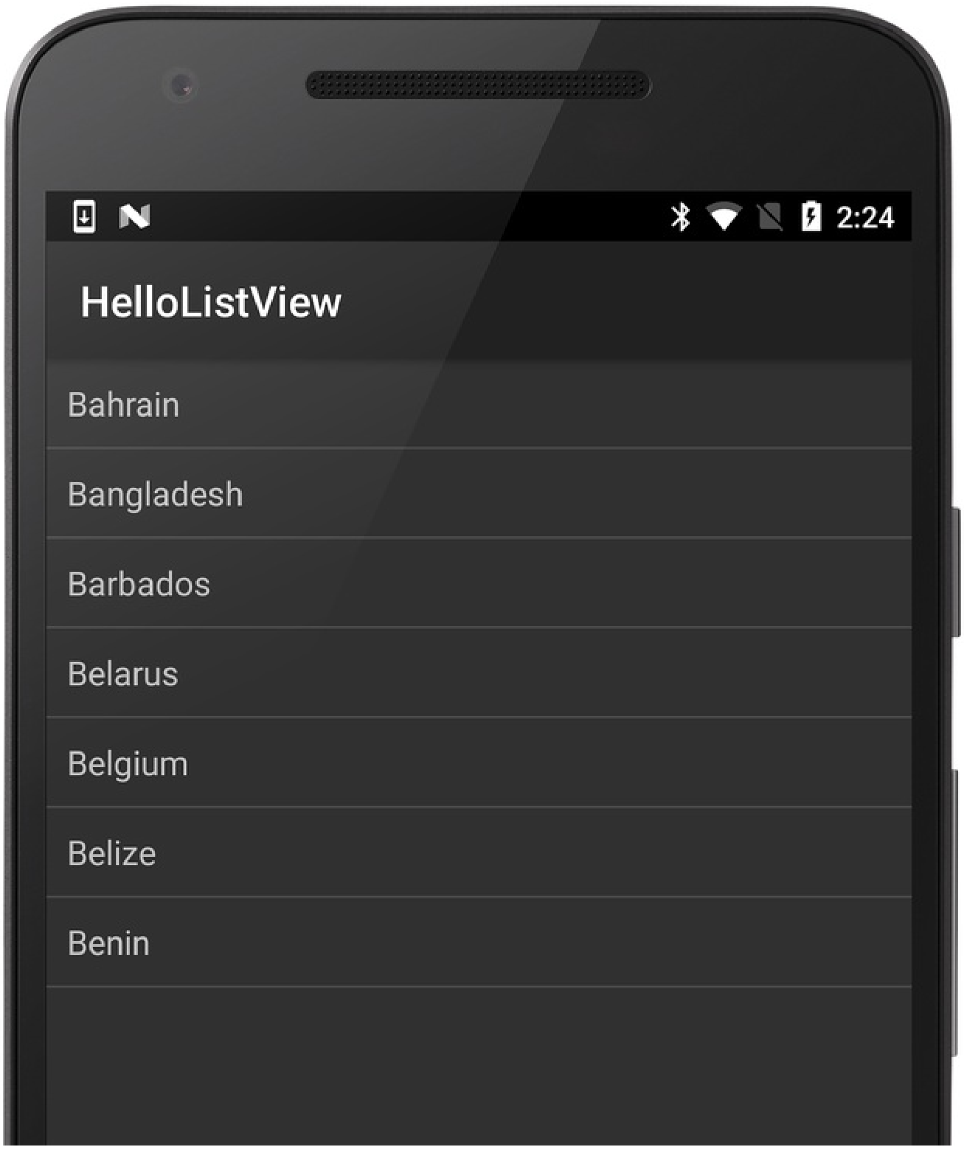
Note that using a hard-coded string array is not the best design practice. One is used in this tutorial for simplicity, to demonstrate the `ListView` widget. The better practice is to reference a string array defined by an external resource, such as with a `string-array` resource in your project **Resources/Values/Strings.xml** file. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">HelloListView</string>
  <string-array name="countries_array">
    <item>Bahrain</item>
    <item>Bangladesh</item>
    <item>Barbados</item>
    <item>Belarus</item>
    <item>Belgium</item>
    <item>Belize</item>
    <item>Benin</item>
  </string-array>
</resources>
```

To use these resource strings for the `ArrayAdapter`, replace the original `ListAdapter` line with the following:

```
string[] countries = Resources.GetStringArray (Resource.Array.countries_array);
ListAdapter = new ArrayAdapter<string> (this, Resource.Layout.list_item, countries);
```

Run the application. You should see something like this:



Going Further with ListView

The remaining topics (linked below) take a comprehensive look at working with the `ListView` class and the different types of Adapter types you can use with it. The structure is as follows:

- **Visual Appearance** – Parts of the `ListView` control and how they work.
- **Classes** – Overview of the classes used to display a `ListView`.
- **Displaying Data in a ListView** – How to display a simple list of data; how to implement `ListView's` usability features; how to use different built-in row layouts; and how Adapters save memory by re-using row views.
- **Custom appearance** – Changing the style of the `ListView` with custom layouts, fonts and colors.
- **Using SQLite** – How to display data from a SQLite database with a `CursorAdapter`.
- **Activity Lifecycle** – Design considerations when implementing `ListView` Activities, including where in the lifecycle you should populate your data and when to release resources.

The discussion (broken into six parts) begins with an overview of the `ListView` class itself before introducing progressively more complex examples of how to use it.

- ListView Parts and Functionality
- Populating a ListView with Data
- Customizing a ListView's Appearance
- Using CursorAdapters
- Using a ContentProvider
- ListView and the Activity Lifecycle

Summary

This set of topics introduced `ListView` and provided some examples of how to use the built-in features of the `ListActivity`. It discussed custom implementations of `ListView` that allowed for colorful layouts and using an SQLite database, and it briefly touched on the relevance of the activity lifecycle on your `ListView` implementation.

Related Links

- AccessoryViews (sample)
- BasicTableAndroid (sample)
- BasicTableAdapter (sample)
- BuiltInViews (sample)
- CustomRowView (sample)
- FastScroll (sample)
- SectionIndex (sample)
- SimpleCursorTableAdapter (sample)
- CursorTableAdapter (sample)
- Activity Lifecycle Tutorial
- Working with Tables and Cells (in Xamarin.iOS)
- ListView Class Reference
- ListActivity Class Reference
- BaseAdapter Class Reference
- ArrayAdapter Class Reference
- CursorAdapter Class Reference

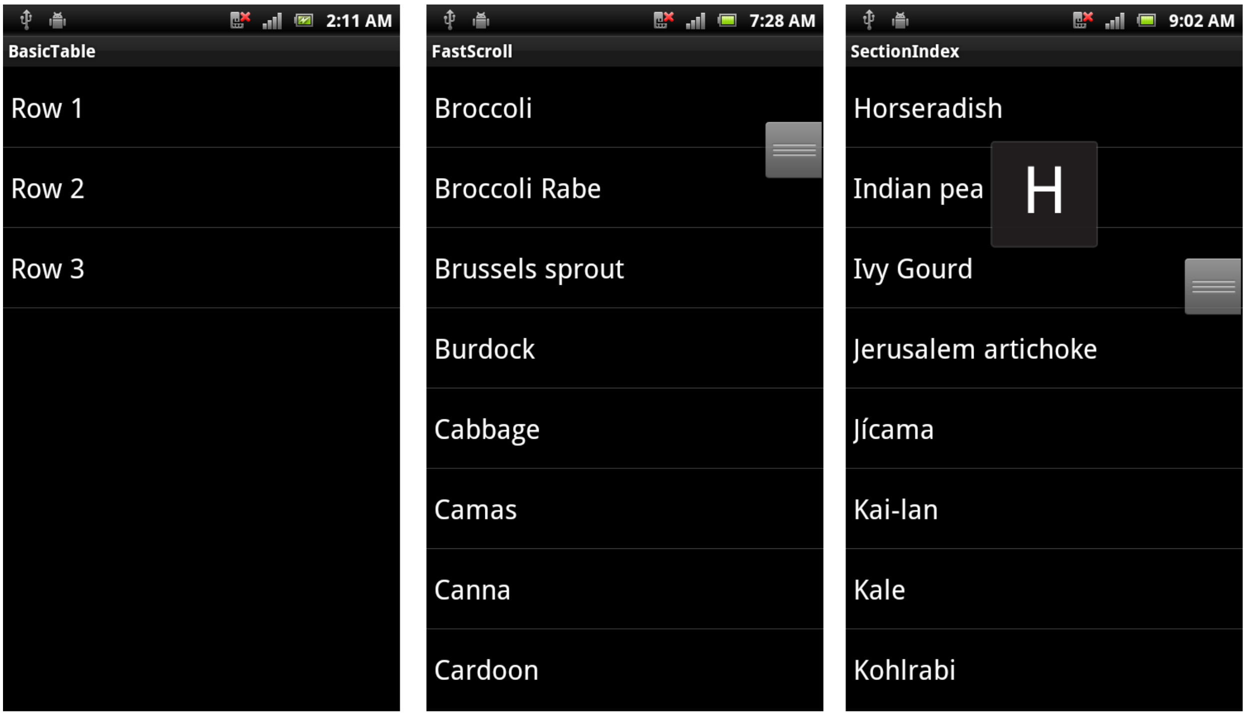
Xamarin.Android ListView Parts and Functionality

7/8/2021 • 2 minutes to read • Edit Online

A `ListView` consists of the following parts:

- **Rows** – The visible representation of the data in the list.
- **Adapter** – A non-visual class that binds the data source to the list view.
- **Fast Scrolling** – A handle that lets the user scroll the length of the list.
- **Section Index** – A user interface element that floats over the scrolling rows to indicate where in the list the current rows are located.

These screenshots use a basic `ListView` control to show how Fast Scrolling and Section Index are rendered:



Plain old rows | Fast Scrolling | Section Index

The elements that make up a `ListView` are described in more detail below:

Rows

Each row has its own `View`. The view can be either one of the built-in views defined in `Android.Resources`, or a custom view. Each row can use the same view layout or they can all be different. There are examples in this document of using built-in layouts and others explaining how to define custom layouts.

Adapter

The `ListView` control requires an `Adapter` to supply the formatted `View` for each row. Android has built-in Adapters and Views that can be used, or custom classes can be created.

Fast Scrolling

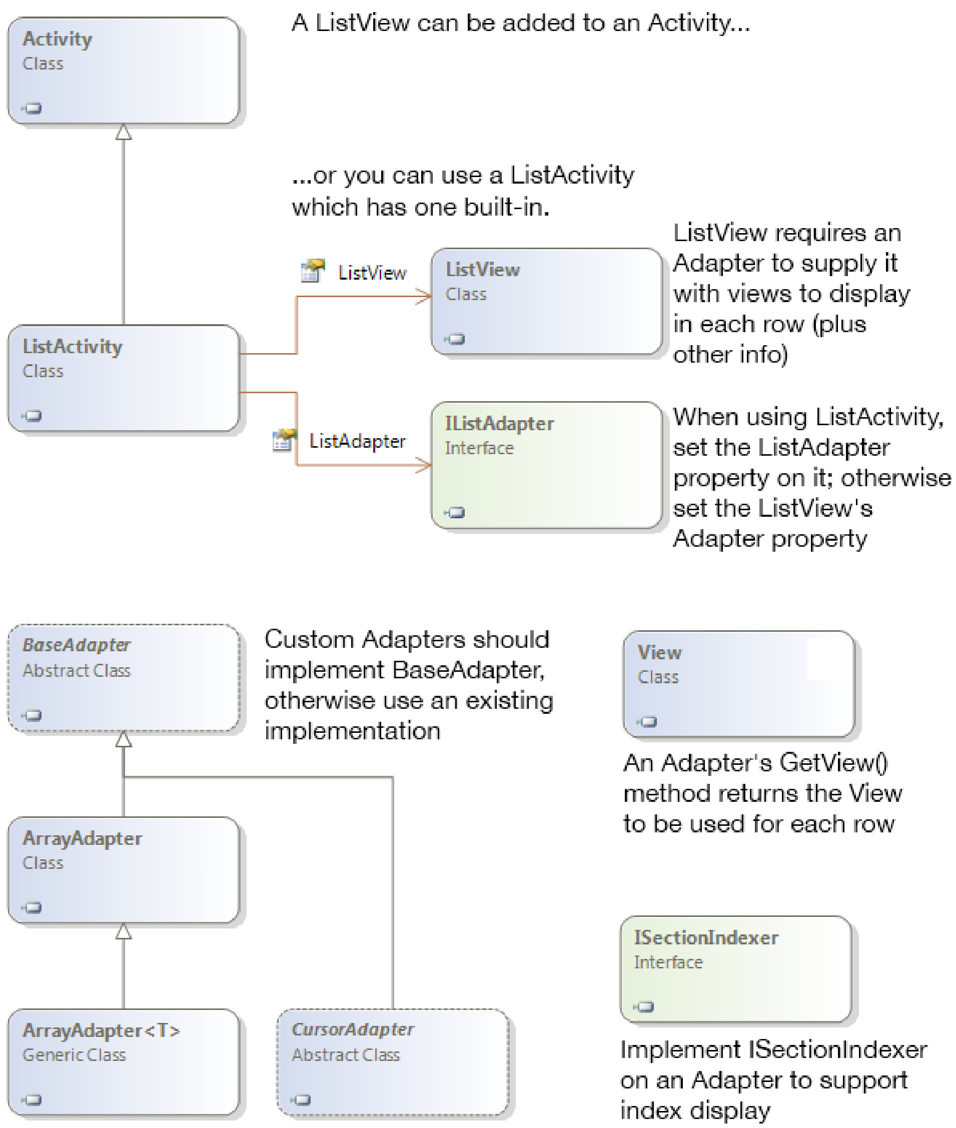
When a `ListView` contains many rows of data fast-scrolling can be enabled to help the user navigate to any part of the list. The fast-scrolling 'scroll bar' can be optionally enabled (and customized in API level 11 and higher).

Section Index

While scrolling through long lists, the optional section index provides the user with feedback on what part of the list they are currently viewing. It is only appropriate on long lists, typically in conjunction with fast scrolling.

Classes Overview

The primary classes used to display `ListViews` are shown here:



The purpose of each class is described below:

- **ListView** – user interface element that displays a scrollable collection of rows. On phones it usually uses up the entire screen (in which case, the `ListActivity` class can be used) or it could be part of a larger layout on phones or tablet devices.
- **View** – a View in Android can be any user interface element, but in the context of a `ListView` it requires a `View` to be supplied for each row.
- **BaseAdapter** – Base class for Adapter implementations to bind a `ListView` to a data source.
- **ArrayAdapter** – Built-in Adapter class that binds an array of strings to a `ListView` for display. The

generic `ArrayAdapter<T>` does the same for other types.

- **CursorAdapter** – Use `CursorAdapter` or `SimpleCursorAdapter` to display data based on an SQLite query.

This document contains simple examples that use an `ArrayAdapter` as well as more complex examples that require custom implementations of `BaseAdapter` or `CursorAdapter`.

Populating a Xamarin.Android ListView with data

7/8/2021 • 6 minutes to read • Edit Online

To add rows to a `ListView` you need to add it to your layout and implement an `IListAdapter` with methods that the `ListView` calls to populate itself. Android includes built-in `ListActivity` and `ArrayAdapter` classes that you can use without defining any custom layout XML or code. The `ListActivity` class automatically creates a `ListView` and exposes a `ListAdapter` property to supply the row views to display via an adapter.

The built-in adapters take a view resource ID as a parameter that gets used for each row. You can use built-in resources such as those in `Android.Resource.Layout` so you don't need to write your own.

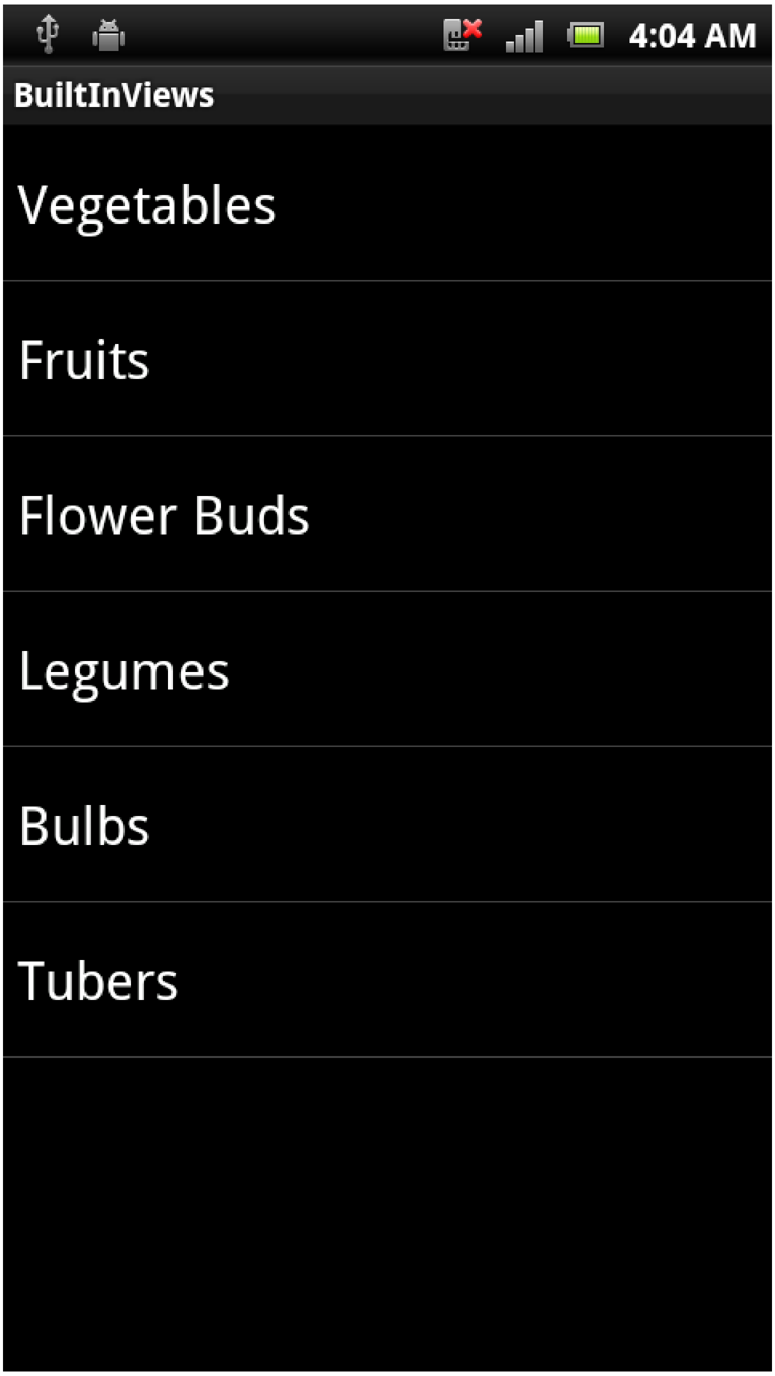
Using ListActivity and ArrayAdapter<String>

The example **BasicTable/HomeScreen.cs** demonstrates how to use these classes to display a `ListView` in only a few lines of code:

```
[Activity(Label = "BasicTable", MainLauncher = true, Icon = "@drawable/icon")]
public class HomeScreen : ListActivity {
   string[] items;
   protected override void OnCreate(Bundle bundle)
   {
       base.OnCreate(bundle);
       items = new string[] { "Vegetables","Fruits","Flower Buds","Legumes","Bulbs","Tubers" };
       ListAdapter = new ArrayAdapter<String>(this, Android.Resource.Layout.SimpleListItem1, items);
   }
}
```

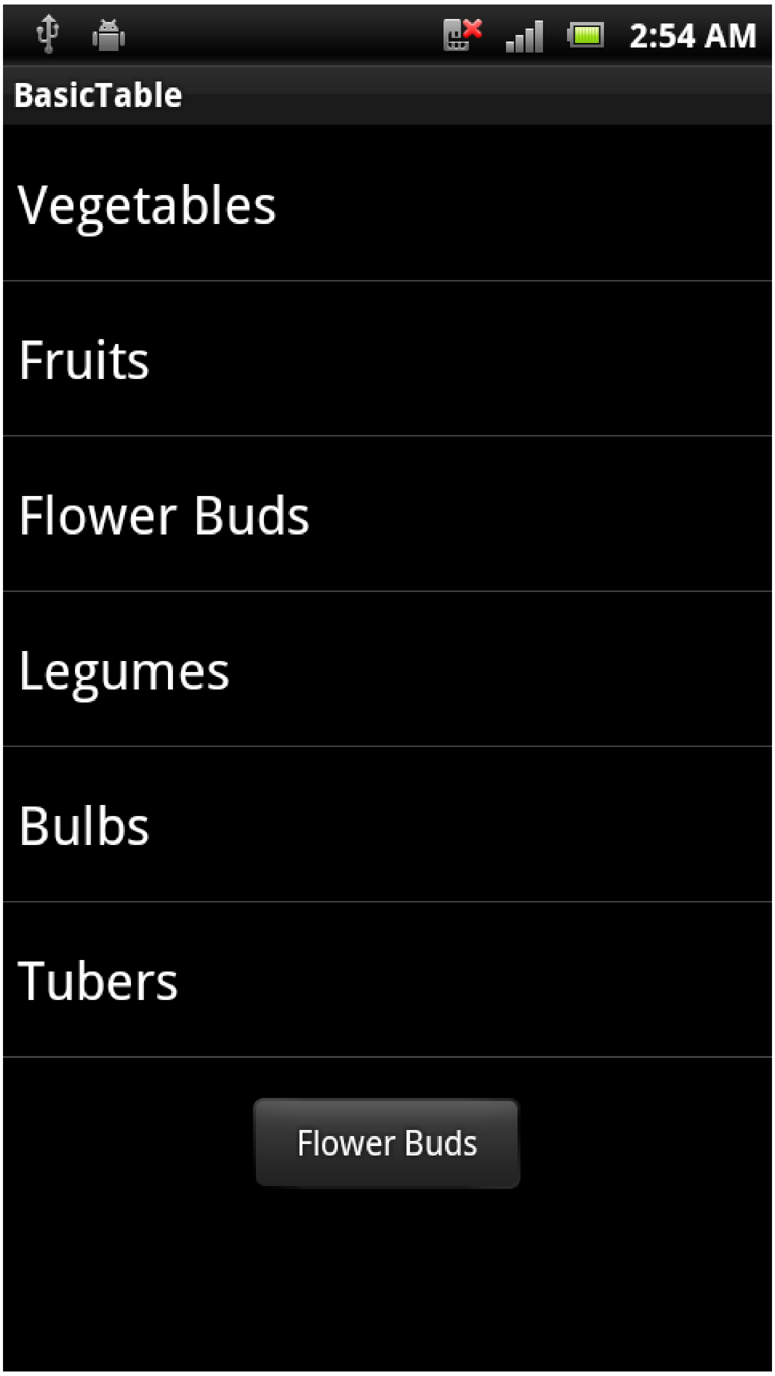
Handling row clicks

Usually a `ListView` will also allow the user to touch a row to perform some action (such as playing a song, or calling a contact, or showing another screen). To respond to user touches there needs to be one more method implemented in the `ListActivity` – `OnListItemClick` – like this:




```
protected override void OnListItemClick(ListView l, View v, int position, long id)
{
   var t = items[position];
   Android.Widget.Toast.MakeText(this, t, Android.Widget.ToastLength.Short).Show();
}
```

Now the user can touch a row and a `Toast` alert will appear:



Implementing a ListAdapter

`ArrayAdapter<string>` is great because of its simplicity, but it's extremely limited. However, often times you have a collection of business entities, rather than just strings that you want to bind. For example, if your data consists of a collection of Employee classes, then you might want the list to just display the names of each employee. To customize the behavior of a `ListView` to control what data is displayed you must implement a subclass of `BaseAdapter` overriding the following four items:

- **Count** – To tell the control how many rows are in the data.
- **GetView** – To return a View for each row, populated with data. This method has a parameter for the `ListView` to pass in an existing, unused row for re-use.
- **GetItemId** – Return a row identifier (typically the row number, although it can be any long value that you like).
- **this[int]** indexer – To return the data associated with a particular row number.

The example code in **BasicTableAdapter/HomeScreenAdapter.cs** demonstrates how to subclass

`BaseAdapter`:

```
public class HomeScreenAdapter : BaseAdapter<string> {
   string[] items;
   Activity context;
   public HomeScreenAdapter(Activity context, string[] items) : base() {
       this.context = context;
       this.items = items;
   }
   public override long GetItemId(int position)
  {
       return position;
   }
   public override string this[int position] {
       get { return items[position]; }
   }
   public override int Count {
       get { return items.Length; }
   }
   public override View GetView(int position, View convertView, ViewGroup parent)
   {
       View view = convertView; // re-use an existing view, if one is available
      if (view == null) // otherwise create a new one
           view = context.LayoutInflater.Inflate(Android.Resource.Layout.SimpleListItem1, null);
       view.FindViewById<TextView>(Android.Resource.Id.Text1).Text = items[position];
       return view;
   }
}
```

Using a custom adapter

Using the custom adapter is similar to the built-in `ArrayAdapter`, passing in a `context` and the `string[]` of values to display:

```
ListAdapter = new HomeScreenAdapter(this, items);
```

Because this example uses the same row layout (`SimpleListItem1`) the resulting application will look identical to the previous example.

Row view re-Use

In this example there are only six items. Since the screen can fit eight, no row re-use required. When displaying hundreds or thousands of rows, however, it would be a waste of memory to create hundreds or thousands of `View` objects when only eight fit on the screen at a time. To avoid this situation, when a row disappears from the screen its view is placed in a queue for re-use. As the user scrolls, the `ListView` calls `GetView` to request new views to display – if available it passes an unused view in the `convertView` parameter. If this value is null then your code should create a new view instance, otherwise you can re-set the properties of that object and re-use it.

The `GetView` method should follow this pattern to re-use row views:

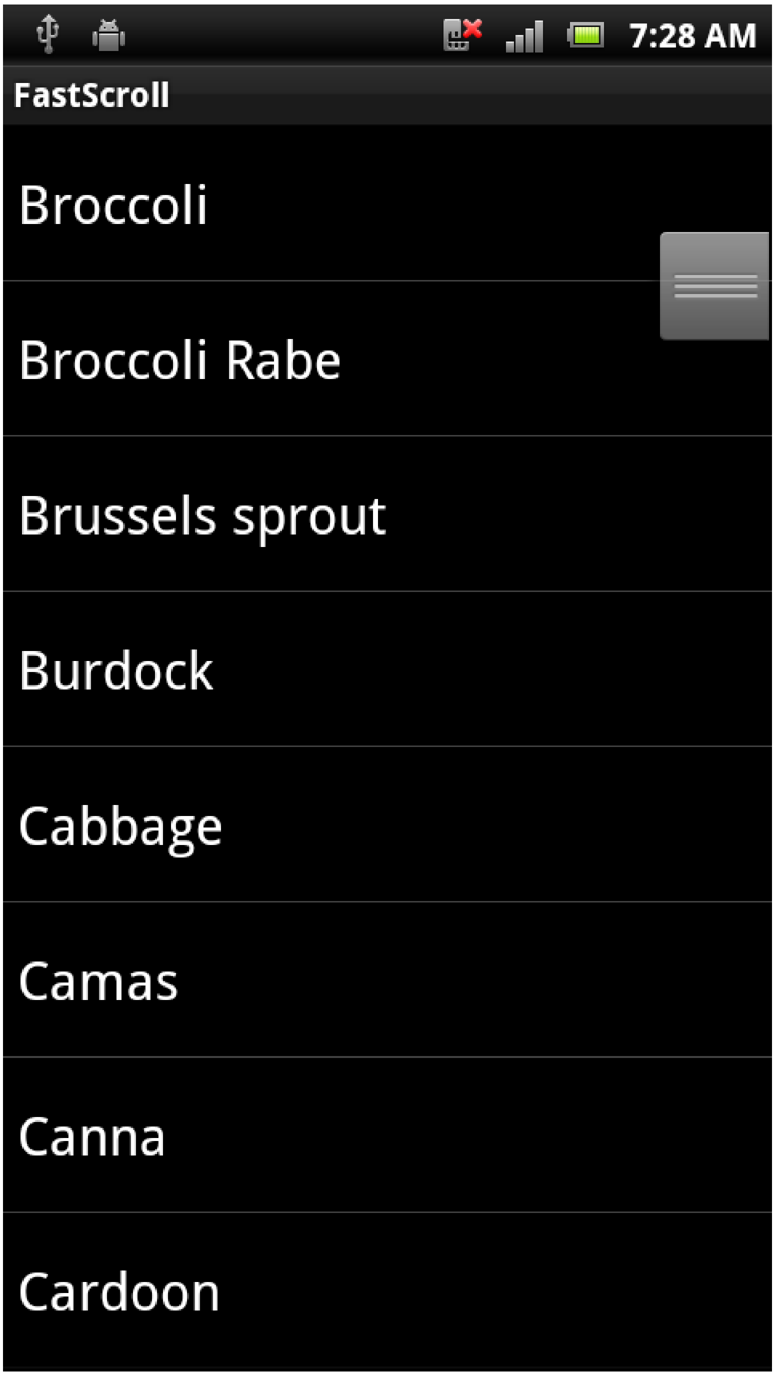
```
public override View GetView(int position, View convertView, ViewGroup parent)
{
   View view = convertView; // re-use an existing view, if one is supplied
   if (view == null) // otherwise create a new one
       view = context.LayoutInflater.Inflate(Android.Resource.Layout.SimpleListItem1, null);
   // set view properties to reflect data for the given row
   view.FindViewById<TextView>(Android.Resource.Id.Text1).Text = items[position];
   // return the view, populated with data, for display
   return view;
}
```

Custom adapter implementations should *always* re-use the `convertView` object before creating new views to ensure they do not run out of memory when displaying long lists.

Some adapter implementations (such as the `CursorAdapter`) don't have a `GetView` method, rather they require two different methods `NewView` and `BindView` which enforce row re-use by separating the responsibilities of `GetView` into two methods. There is a `CursorAdapter` example later in the document.

Enabling fast scrolling

Fast Scrolling helps the user to scroll through long lists by providing an additional 'handle' that acts as a scroll bar to directly access a part of the list. This screenshot shows the fast scroll handle:



Causing the fast scrolling handle to appear is as simple as setting the `FastScrollEnabled` property to `true`:

```
ListView.FastScrollEnabled = true;
```

Adding a section index

A section index provides additional feedback for users when they are fast-scrolling through a long list – it shows which 'section' they have scrolled to. To cause the section index to appear the Adapter subclass must implement the `ISectionIndexer` interface to supply the index text depending on the rows being displayed:

To implement `ISectionIndexer` you need to add three methods to an adapter:

- **GetSections** – Provides the complete list of section index titles that could be displayed. This method requires an array of Java Objects so the code needs to create a `Java.Lang.Object[]` from a .NET collection. In our example it returns a list of the initial characters in the list as `Java.Lang.String` .
- **GetPositionForSection** – Returns the first row position for a given section index.
- **GetSectionForPosition** – Returns the section index to be displayed for a given row.

The example `SectionIndex/HomeScreenAdapter.cs` file implements those methods, and some additional code in the constructor. The constructor builds the section index by looping through every row and extracting the first character of the title (the items must already be sorted for this to work).

```
alphaIndex = new Dictionary<string, int>();
for (int i = 0; i < items.Length; i++) { // loop through items
   var key = items[i][0].ToString();
   if (!alphaIndex.ContainsKey(key))
       alphaIndex.Add(key, i); // add each 'new' letter to the index
}
sections = new string[alphaIndex.Keys.Count];
alphaIndex.Keys.CopyTo(sections, 0); // convert letters list to string[]

// Interface requires a Java.Lang.Object[], so we create one here
sectionsObjects = new Java.Lang.Object[sections.Length];
for (int i = 0; i < sections.Length; i++) {
   sectionsObjects[i] = new Java.Lang.String(sections[i]);
}
```

With the data structures created, the `ISectionIndexer` methods are very simple:

```
public Java.Lang.Object[] GetSections()
{
   return sectionsObjects;
}
public int GetPositionForSection(int section)
{
   return alphaIndexer[sections[section]];
}
public int GetSectionForPosition(int position)
{   // this method isn't called in this example, but code is provided for completeness
    int prevSection = 0;
    for (int i = 0; i < sections.Length; i++)
    {
        if (GetPositionForSection(i) > position)
        {
            break;
        }
        prevSection = i;
    }
    return prevSection;
}
```

Your section index titles don't need to map 1:1 to your actual sections. This is why the `GetPositionForSection` method exists. `GetPositionForSection` gives you an opportunity to map whatever indices are in your index list to whatever sections are in your list view. For example, you may have a "z" in your index, but you may not have a table section for every letter, so instead of "z" mapping to 26, it may map to 25 or 24, or whatever section index "z" should map to.

Related links

- BasicTableAndroid (sample)
- BasicTableAdapter (sample)
- FastScroll (sample)

Customizing a ListView's Appearance with Xamarin.Android

7/8/2021 • 9 minutes to read • Edit Online

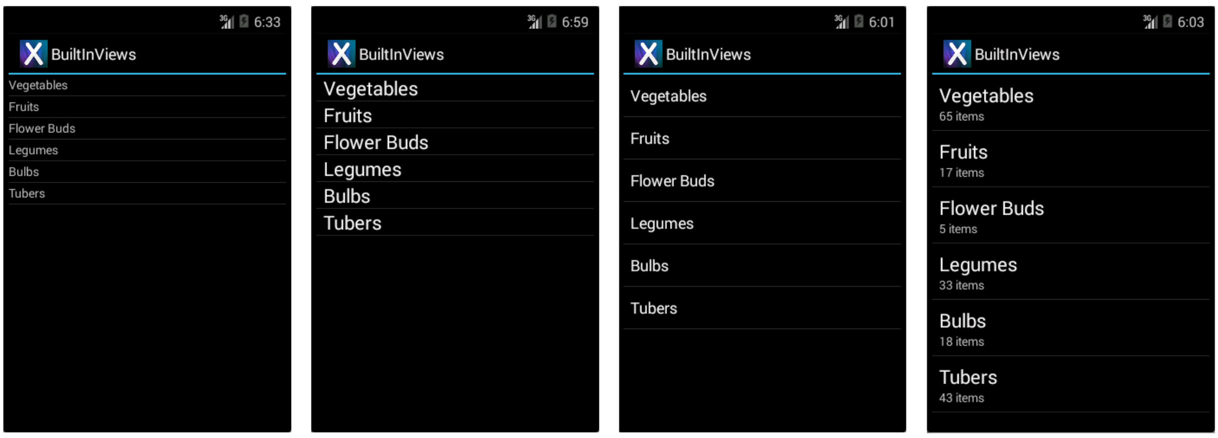
The appearance of a ListView is dictated by the layout of the rows being displayed. To change the appearance of a `ListView`, use a different row layout.

Built-in Row Views

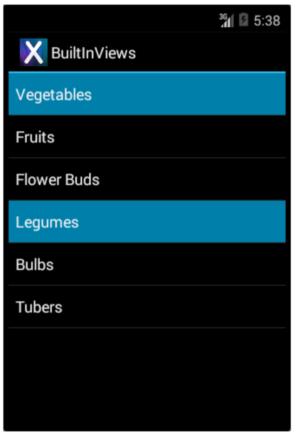
There are twelve built-in Views that can be referenced using **Android.Resource.Layout**:

- **TestListItem** – Single line of text with minimal formatting.
- **SimpleListItem1** – Single line of text.
- **SimpleListItem2** – Two lines of text.
- **SimpleSelectableListItem** – Single line of text that supports single or multiple item selection (added in API level 11).
- **SimpleListItemActivated1** – Similar to SimpleListItem1, but the background color indicates when a row is selected (added in API level 11).
- **SimpleListItemActivated2** – Similar to SimpleListItem2, but the background color indicates when a row is selected (added in API level 11).
- **SimpleListItemChecked** – Displays check marks to indicate selection.
- **SimpleListItemMultipleChoice** – Displays check boxes to indicate multiple-choice selection.
- **SimpleListItemSingleChoice** – Displays radio buttons to indicate mutually-exclusive selection.
- **TwoLineListItem** – Two lines of text.
- **ActivityListItem** – Single line of text with an image.
- **SimpleExpandableListItem** – Groups rows by categories, and each group can be expanded or collapsed.

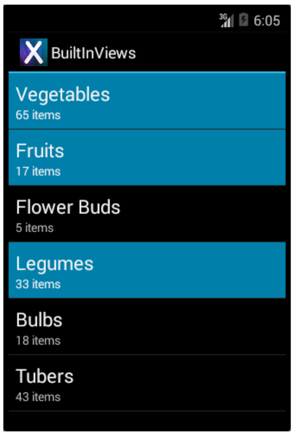
Each built-in row view has a built in style associated with it. These screenshots show how each view appears:



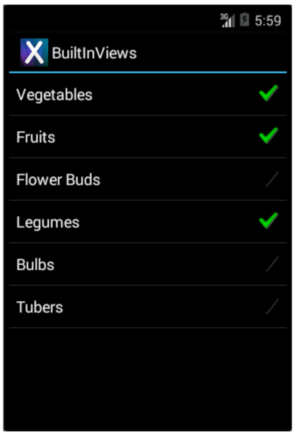
TestListItem SimpleSelectableListItem SimpleListItem1 SimpleListItem2



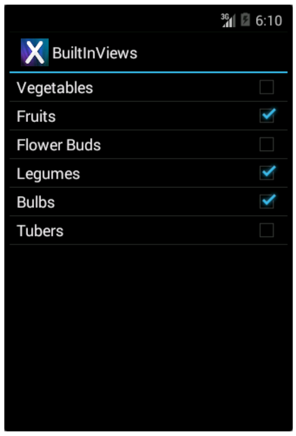
SimpleListItemActivated1



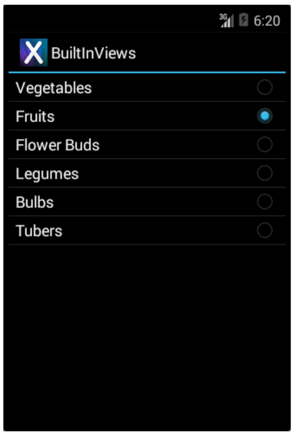
SimpleListItemActivated2



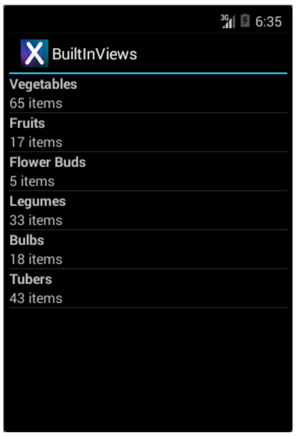
SimpleListItemChecked



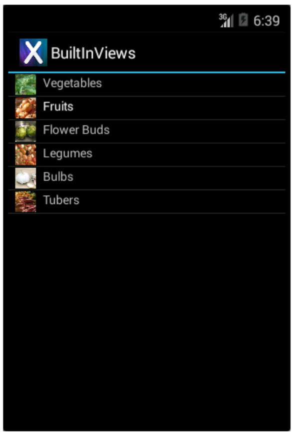
SimpleListItemMultipleChoice



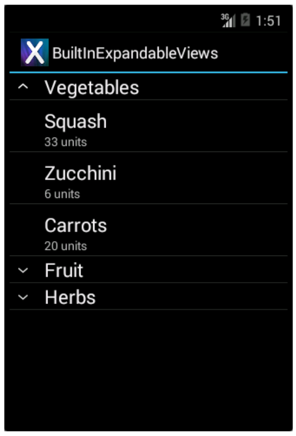
SimpleListItemSingleChoice



TwoLineListItem



ActivityListItem



SimpleExpandableListItem

The **BuiltInViews/HomeScreenAdapter.cs** sample file (in the **BuiltInViews** solution) contains the code to produce the non-expandable list item screens. The view is set in the `GetView` method like this:

```
view = context.LayoutInflater.Inflate(Android.Resource.Layout.SimpleListItem1, null);
```

The view's properties can then be set by referencing the standard control identifiers `Text1`, `Text2` and `Icon` under `Android.Resource.Id` (do not set properties that the view does not contain or an exception will be thrown):

```
view.FindViewById<TextView>(Android.Resource.Id.Text1).Text = item.Heading;
view.FindViewById<TextView>(Android.Resource.Id.Text2).Text = item.SubHeading;
view.FindViewById<ImageView>(Android.Resource.Id.Icon).SetImageResource(item.ImageResourceId); // only use
with ActivityListItem
```

The **BuiltInExpandableViews/ExpandableScreenAdapter.cs** sample file (in the **BuiltInViews** solution) contains the code to produce the SimpleExpandableListItem screen. The group view is set in the `GetGroupView` method like this:

```
view = context.LayoutInflater.Inflate(Android.Resource.Layout.SimpleExpandableListItem1, null);
```

The child view is set in the `GetChildView` method like this:

```
view = context.LayoutInflater.Inflate(Android.Resource.Layout.SimpleExpandableListItem2, null);
```

The properties for the group view and the child view can then be set by referencing the standard `Text1` and

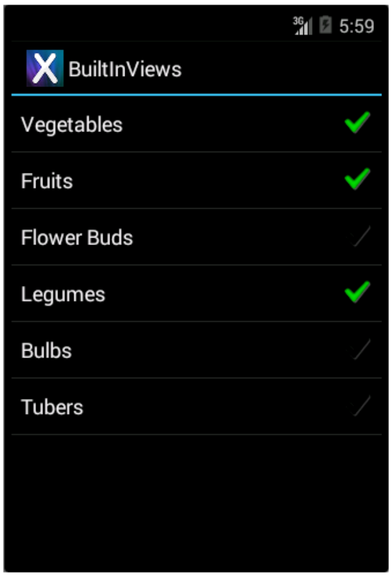
`Text2` control identifiers as shown above. The SimpleExpandableListItem screenshot (shown above) provides an example of a one-line group view (SimpleExpandableListItem1) and a two-line child view (SimpleExpandableListItem2). Alternately, the group view can be configured for two lines (SimpleExpandableListItem2) and the child view can be configured for one line (SimpleExpandableListItem1), or both group view and child view can have the same number of lines.

Accessories

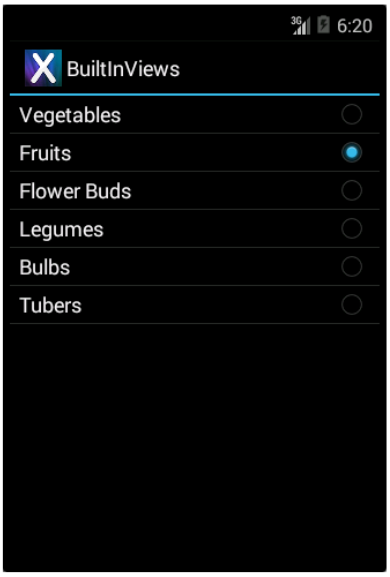
Rows can have accessories added to the right of the view to indicate selection state:

- **SimpleListItemChecked** – Creates a single-selection list with a check as the indicator.
- **SimpleListItemSingleChoice** – Creates radio-button-type lists where only one choice is possible.
- **SimpleListItemMultipleChoice** – Creates checkbox-type lists where multiple choices are possible.

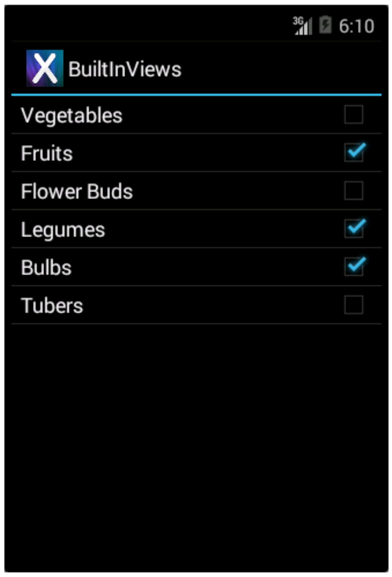
The aforementioned accessories are illustrated in the following screens, in their respective order:



SimpleListItemChecked



SimpleListItemSingleChoice



SimpleListItemMultipleChoice

To display one of these accessories pass the required layout resource ID to the adapter then manually set the selection state for the required rows. This line of code shows how to create and assign an `Adapter` using one of these layouts:

```
ListAdapter = new ArrayAdapter<String>(this, Android.Resource.Layout.SimpleListItemChecked, items);
```

The `ListView` itself supports different selection modes, regardless of the accessory being displayed. To avoid confusion, use `Single` selection mode with `SingleChoice` accessories and the `Checked` or `Multiple` mode with the `MultipleChoice` style. The selection mode is controlled by the `ChoiceMode` property of the `ListView`.

Handling API Level

Earlier versions of Xamarin.Android implemented enumerations as integer properties. The latest version has introduced proper .NET enumeration types which makes it much easier to discover the potential options.

Depending on which API level you are targeting, `ChoiceMode` is either an integer or an enumeration. The sample file **AccessoryViews/HomeScreen.cs** has a block commented out if you wish to target the Gingerbread API:

```
// For targeting Gingerbread the ChoiceMode is an int, otherwise it is an
// enumeration.

lv.ChoiceMode = Android.Widget.ChoiceMode.Single; // 1
//lv.ChoiceMode = Android.Widget.ChoiceMode.Multiple; // 2
//lv.ChoiceMode = Android.Widget.ChoiceMode.None; // 0

// Use this block if targeting Gingerbread or lower
/*
lv.ChoiceMode = 1; // Single
//lv.ChoiceMode = 0; // none
//lv.ChoiceMode = 2; // Multiple
//lv.ChoiceMode = 3; // MultipleModal
*/
```

Selecting Items Programmatically

Manually setting which items are 'selected' is done with the `SetItemChecked` method (it can be called multiple times for multiple selection):

```
// Set the initially checked row ("Fruits")
lv.SetItemChecked(1, true);
```

The code also needs to detect single selections differently from multiple selections. To determine which row has been selected in `Single` mode use the `CheckedItemPosition` integer property:

```
FindViewById<ListView>(Android.Resource.Id.List).CheckedItemPosition
```

To determine which rows have been selected in `Multiple` mode you need to loop through the `CheckedItemPositions` `SparseBooleanArray`. A sparse array is like a dictionary that only contains entries where the value has been changed, so you must traverse the entire array looking for `true` values to know what has been selected in the list as illustrated in the following code snippet:

```
var sparseArray = FindViewById<ListView>(Android.Resource.Id.List).CheckedItemPositions;
for (var i = 0; i < sparseArray.Size(); i++ )
{
   Console.Write(sparseArray.KeyAt(i) + "=" + sparseArray.ValueAt(i) + ",");
}
Console.WriteLine();
```

Creating Custom Row Layouts

The four built-in row views are very simple. To display more complex layouts (such as a list of emails, or tweets, or contact info) a custom view is required. Custom views are generally declared as AXML files in the **Resources/Layout** directory and then loaded using their resource Id by a custom adapter. The view can contain any number of display classes (such as TextViews, ImageViews and other controls) with custom colors, fonts and layout.

This example differs from the previous examples in a number of ways:

- Inherits from `Activity` , not `ListActivity` . You can customize rows for any `ListView` , however other controls can also be included in an `Activity` layout (such as a heading, buttons or other user interface elements). This example adds a heading above the `ListView` to illustrate.
- Requires an AXML layout file for the screen; in the previous examples the `ListActivity` does not require a layout file. This AXML contains a `ListView` control declaration.

- Requires an AXML layout file to render each row. This AXML file contains the text and image controls with custom font and color settings.
- Uses an optional custom selector XML file to set the appearance of the row when it is selected.
- The `Adapter` implementation returns a custom layout from the `GetView` override.
- `ItemClick` must be declared differently (an event handler is attached to `ListView.ItemClick` rather than an overriding `OnListItemClick` in `ListActivity`).

These changes are detailed below, starting with creating the activity's view and the custom row view and then covering the modifications to the Adapter and Activity to render them.

Adding a ListView to an Activity Layout

Because `HomeScreen` no longer inherits from `ListActivity` it doesn't have a default view, so a layout AXML file must be created for the HomeScreen's view. For this example, the view will have a heading (using a `TextView`) and a `ListView` to display data. The layout is defined in the **Resources/Layout/HomeScreen.axml** file which is shown here:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent">
    <TextView android:id="@+id/Heading"
        android:text="Vegetable Groups"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:background="#00000000"
        android:textSize="30dp"
        android:textColor="#FF267F00"
        android:textStyle="bold"
        android:padding="5dp"
    />
    <ListView android:id="@+id/List"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:cacheColorHint="#FFDAFF7F"
    />
</LinearLayout>
```

The benefit of using an `Activity` with a custom layout (instead of a `ListActivity`) lies in being able to add additional controls to the screen, such as the heading `TextView` in this example.

Creating a Custom Row Layout

Another AXML layout file is required to contain the custom layout for each row that will appear in the list view. In this example the row will have a green background, brown text and right-aligned image. The Android XML markup to declare this layout is described in **Resources/Layout/CustomView.axml**:

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout  xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   android:background="#FFDAFF7F"
   android:padding="8dp">
    <LinearLayout android:id="@+id/Text"
       android:orientation="vertical"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:paddingLeft="10dip">
        <TextView
         android:id="@+id/Text1"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:textColor="#FF7F3300"
         android:textSize="20dip"
         android:textStyle="italic"
         />
        <TextView
         android:id="@+id/Text2"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:textSize="14dip"
         android:textColor="#FF267F00"
         android:paddingLeft="100dip"
         />
    </LinearLayout>
    <ImageView
        android:id="@+id/Image"
        android:layout_width="48dp"
        android:layout_height="48dp"
        android:padding="5dp"
        android:src="@drawable/icon"
        android:layout_alignParentRight="true" />
</RelativeLayout >
```

While a custom row layout can contain many different controls, scrolling performance can be affected by complex designs and using images (especially if they have to be loaded over the network). See Google's article for more information on addressing scrolling performance issues.

Referencing a Custom Row View

The implementation of the custom adapter example is in `HomeScreenAdapter.cs`. The key method is `GetView` where it loads the custom AXML using the resource ID `Resource.Layout.CustomView`, and then sets properties on each of the controls in the view before returning it. The complete adapter class is shown:

```
public class HomeScreenAdapter : BaseAdapter<TableItem> {
   List<TableItem> items;
   Activity context;
   public HomeScreenAdapter(Activity context, List<TableItem> items)
       : base()
   {
       this.context = context;
       this.items = items;
   }
   public override long GetItemId(int position)
   {
       return position;
   }
   public override TableItem this[int position]
   {
       get { return items[position]; }
   }
   public override int Count
   {
       get { return items.Count; }
   }
   public override View GetView(int position, View convertView, ViewGroup parent)
   {
       var item = items[position];
       View view = convertView;
       if (view == null) // no view to re-use, create new
           view = context.LayoutInflater.Inflate(Resource.Layout.CustomView, null);
       view.FindViewById<TextView>(Resource.Id.Text1).Text = item.Heading;
       view.FindViewById<TextView>(Resource.Id.Text2).Text = item.SubHeading;
       view.FindViewById<ImageView>(Resource.Id.Image).SetImageResource(item.ImageResourceId);
       return view;
   }
}
```

Referencing the Custom ListView in the Activity

Because the `HomeScreen` class now inherits from `Activity`, a `ListView` field is declared in the class to hold a reference to the control declared in the AXML:

```
ListView listView;
```

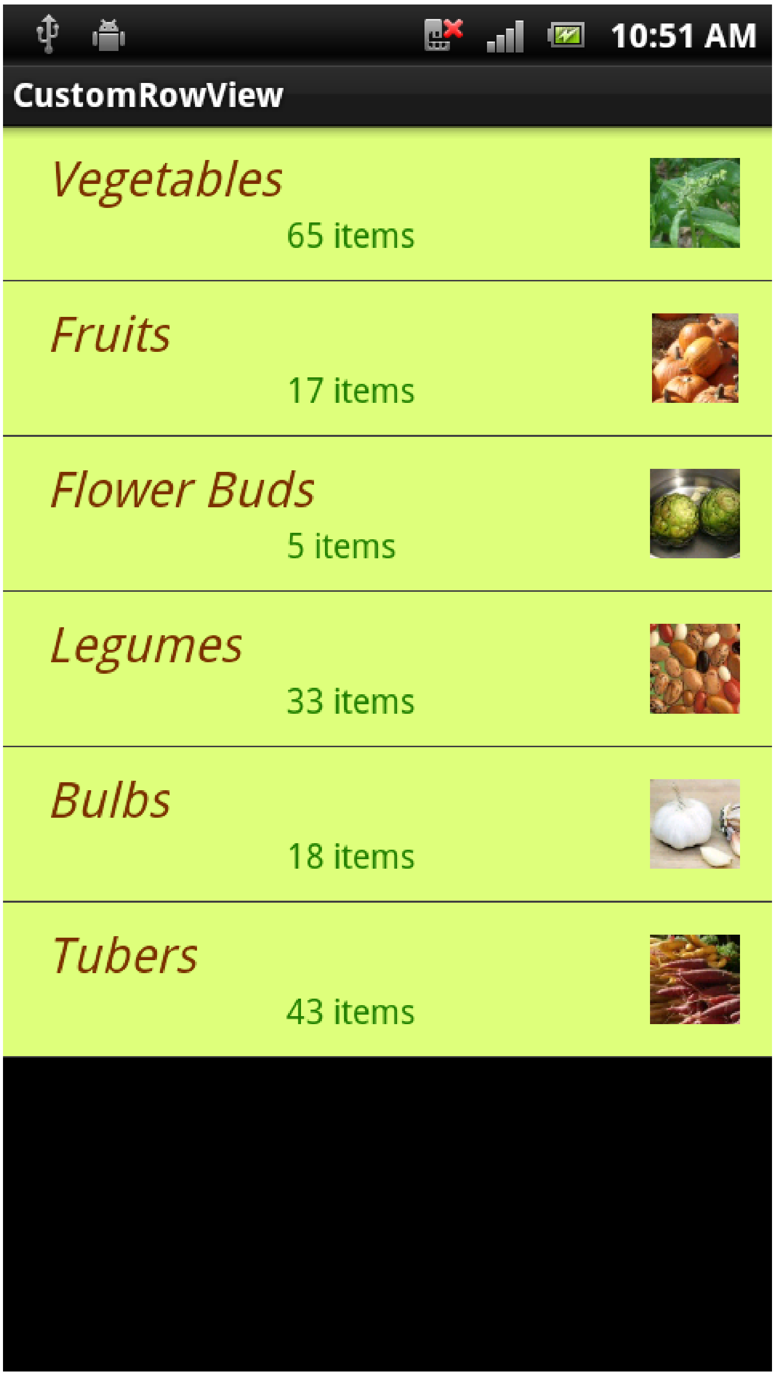
The class must then load the Activity's custom layout AXML using the `SetContentView` method. It can then find the `ListView` control in the layout then creates and assigns the adapter and assigns the click handler. The code for the OnCreate method is shown here:

```
SetContentView(Resource.Layout.HomeScreen); // loads the HomeScreen.axml as this activity's view
listView = FindViewById<ListView>(Resource.Id.List); // get reference to the ListView in the layout

// populate the listview with data
listView.Adapter = new HomeScreenAdapter(this, tableItems);
listView.ItemClick += OnListItemClick;  // to be defined
```

Finally the `ItemClick` handler must be defined; in this case it just displays a `Toast` message:

```
void OnListItemClick(object sender, AdapterView.ItemClickEventArgs e)
{
   var listView = sender as ListView;
   var t = tableItems[e.Position];
   Android.Widget.Toast.MakeText(this, t.Heading, Android.Widget.ToastLength.Short).Show();
}
```

The resulting screen looks like this:



Customizing the Row Selector Color

When a row is touched it should be highlighted for user feedback. When a custom view specifies as background color as **CustomView.axml** does, it also overrides the selection highlight. This line of code in **CustomView.axml** sets the background to light green, but it also means there is no visual indicator when the row is touched:

```
android:background="#FFDAFF7F"
```

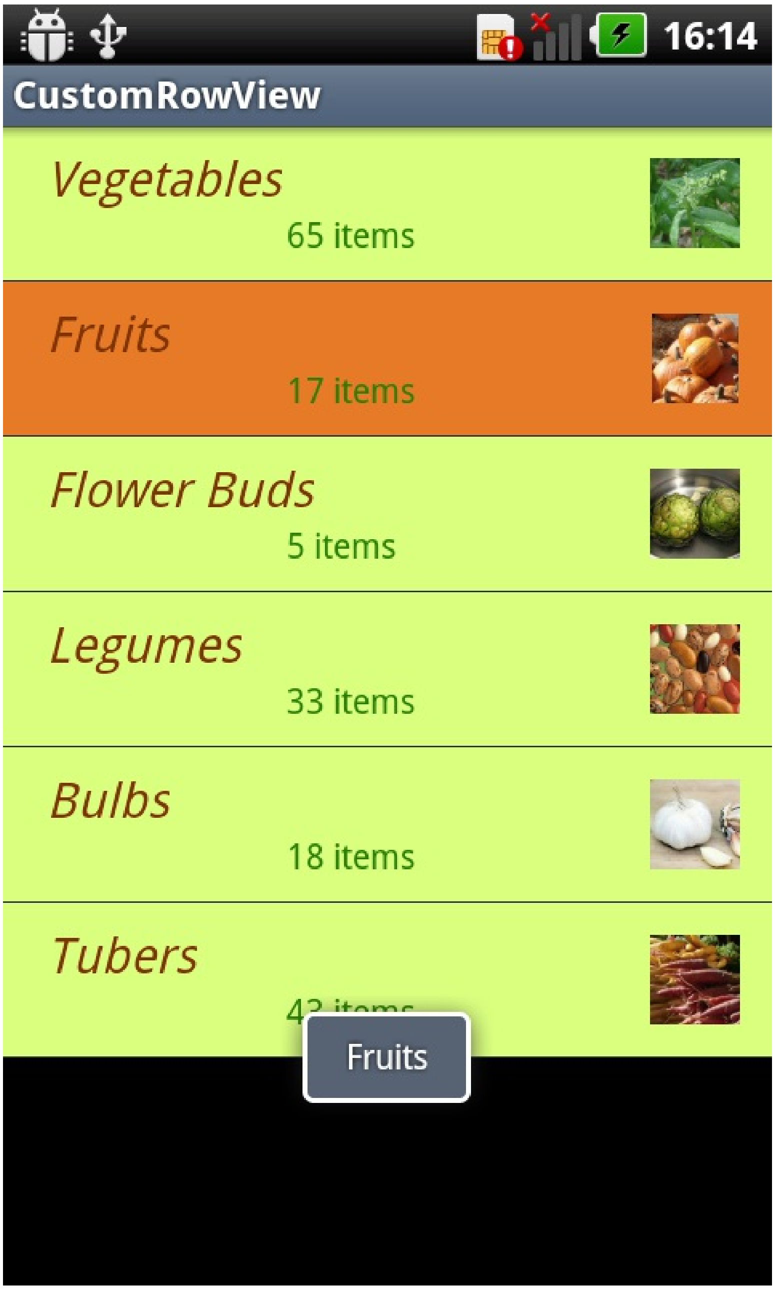
To re-enable the highlight behavior, and also to customize the color that is used, set the background attribute to a custom selector instead. The selector will declare both the default background color as well as the highlight color. The file **Resources/Drawable/CustomSelector.xml** contains the following declaration:

```xml
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
<item android:state_pressed="false"
  android:state_selected="false"
  android:drawable="@color/cellback" />
<item android:state_pressed="true" >
  <shape>
     <gradient
      android:startColor="#E77A26"
        android:endColor="#E77A26"
        android:angle="270" />
  </shape>
</item>
<item android:state_selected="true"
  android:state_pressed="false"
  android:drawable="@color/cellback" />
</selector>
```

To reference the custom selector, change the background attribute in **CustomView.axml** to:

```
android:background="@drawable/CustomSelector"
```

A selected row and the corresponding `Toast` message now looks like this:



Preventing Flickering on Custom Layouts

Android attempts to improve the performance of `ListView` scrolling by caching layout information. If you have long scrolling lists of data you should also set the `android:cacheColorHint` property on the `ListView` declaration in the Activity's AXML definition (to the same color value as your custom row layout's background). Failure to include this hint could result in a 'flicker' as the user scrolls through a list with custom row background colors.

Related Links

- BuiltInViews (sample)
- AccessoryViews (sample)
- CustomRowView (sample)

Using CursorAdapters with Xamarin.Android

11/2/2020 • 5 minutes to read • Edit Online

Android provides adapter classes specifically to display data from an SQLite database query:

SimpleCursorAdapter – Similar to an `ArrayAdapter` because it can be used without subclassing. Simply provide the required parameters (such as a cursor and layout information) in the constructor and then assign to a `ListView`.

CursorAdapter – A base class that you can inherit from when you need more control over the binding of data values to layout controls (for example, hiding/showing controls or changing their properties).

Cursor adapters provide a high-performance way to scroll through long lists of data that are stored in SQLite. The consuming code must define an SQL query in a `Cursor` object and then describe how to create and populate the views for each row.

Creating an SQLite Database

To demonstrate cursor adapters requires a simple SQLite database implementation. The code in **SimpleCursorTableAdapter/VegetableDatabase.cs** contains the code and SQL to create a table and populate it with some data. The complete `VegetableDatabase` class is shown here:

```
class VegetableDatabase  : SQLiteOpenHelper {
   public static readonly string create_table_sql =
       "CREATE TABLE [vegetables] ([_id] INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE, [name] TEXT NOT
NULL UNIQUE)";
   public static readonly string DatabaseName = "vegetables.db";
   public static readonly int DatabaseVersion = 1;
   public VegetableDatabase(Context context) : base(context, DatabaseName, null, DatabaseVersion) { }
   public override void OnCreate(SQLiteDatabase db)
   {
       db.ExecSQL(create_table_sql);
       // seed with data
       db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Vegetables')");
       db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Fruits')");
       db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Flower Buds')");
       db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Legumes')");
       db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Bulbs')");
       db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Tubers')");
   }
   public override void OnUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
   {   // not required until second version :)
       throw new NotImplementedException();
   }
}
```

The `VegetableDatabase` class will be instantiated in the `OnCreate` method of the `HomeScreen` activity. The `SQLiteOpenHelper` base class manages the setup of the database file and ensures that the SQL in its `OnCreate` method is only run once. This class is used in the following two examples for `SimpleCursorAdapter` and `CursorAdapter`.

The cursor query *must* have an integer column `_id` for the `CursorAdapter` to work. If the underlying table does not have an integer column named `_id` then use a column alias for another unique integer in the `RawQuery` that makes up the cursor. Refer to the Android docs for further information.

Creating the Cursor

The examples use a `RawQuery` to turn an SQL query into a `Cursor` object. The column list that is returned from the cursor defines the data columns that are available for display in the cursor adapter. The code that creates the database in the **SimpleCursorTableAdapter/HomeScreen.cs** `OnCreate` method is shown here:

```
vdb = new VegetableDatabase(this);
cursor = vdb.ReadableDatabase.RawQuery("SELECT * FROM vegetables", null); // cursor query
StartManagingCursor(cursor);
// use either SimpleCursorAdapter or CursorAdapter subclass here!
```

Any code that calls `StartManagingCursor` should also call `StopManagingCursor`. The examples use `OnCreate` to start, and `OnDestroy` to close the cursor. The `OnDestroy` method contains this code:

```
StopManagingCursor(cursor);
cursor.Close();
```

Once an application has a SQLite database available and has created a cursor object as shown, it can utilize either a `SimpleCursorAdapter` or a subclass of `CusorAdapter` to display rows in a `ListView`.

Using SimpleCursorAdapter

`SimpleCursorAdapter` is like the `ArrayAdapter`, but specialized for use with SQLite. It does not require subclassing – just set some simple parameters when creating the object and then assign it to a `ListView`'s `Adapter` property.

The parameters for the SimpleCursorAdapter constructor are:

Context – A reference to the containing Activity.

Layout – The resource ID of the row view to use.

ICursor – A cursor containing the SQLite query for the data to display.

From string array – An array of strings corresponding to the names of columns in the cursor.

To integer array – An array of layout IDs that correspond to the controls in the row layout. The value of the column specified in the `from` array will be bound to the ControlID specified in this array at the same index.

The `from` and `to` arrays must have the same number of entries because they form a mapping from the data source to the layout controls in the view.

The **SimpleCursorTableAdapter/HomeScreen.cs** sample code wires up a `SimpleCursorAdapter` like this:

```
// which columns map to which layout controls
string[] fromColumns = new string[] {"name"};
int[] toControlIDs = new int[] {Android.Resource.Id.Text1};
// use a SimpleCursorAdapter
listView.Adapter = new SimpleCursorAdapter (this, Android.Resource.Layout.SimpleListItem1, cursor,
        fromColumns,
        toControlIDs);
```

`SimpleCursorAdapter` is a fast and simple way to display SQLite data in a `ListView`. The main limitation is that it can only bind column values to display controls, it does not allow you to change other aspects of the row layout (for example, showing/hiding controls or changing properties).

Subclassing CursorAdapter

A `CursorAdapter` subclass has the same performance benefits as the `SimpleCursorAdapter` for displaying data from SQLite, but it also gives you complete control over the creation and layout of each row view. The `CursorAdapter` implementation is very different from subclassing `BaseAdapter` because it does not override `GetView`, `GetItemId`, `Count` or `this[]` indexer.

Given a working SQLite database, you only need to override two methods to create a `CursorAdapter` subclass:

- **BindView** – Given a view, update it to display the data in the provided cursor.
- **NewView** – Called when the `ListView` requires a new view to display. The `CursorAdapter` will take care of recycling views (unlike the `GetView` method on regular Adapters).

The adapter subclasses in earlier examples have methods to return the number of rows and to retrieve the current item – the `CursorAdapter` does not require these methods because that information can be gleaned from the cursor itself. By splitting the creation and population of each view into these two methods, the `CursorAdapter` enforces view re-use. This is in contrast to a regular adapter where it's possible to ignore the `convertView` parameter of the `BaseAdapter.GetView` method.

Implementing the CursorAdapter

The code in **CursorTableAdapter/HomeScreenCursorAdapter.cs** contains a `CursorAdapter` subclass. It stores a context reference passed into the constructor so that it can access a `LayoutInflater` in the `NewView` method. The complete class looks like this:

```
public class HomeScreenCursorAdapter : CursorAdapter {
   Activity context;
   public HomeScreenCursorAdapter(Activity context, ICursor c)
       : base(context, c)
   {
       this.context = context;
   }
   public override void BindView(View view, Context context, ICursor cursor)
   {
       var textView = view.FindViewById<TextView>(Android.Resource.Id.Text1);
       textView.Text = cursor.GetString(1); // 'name' is column 1 in the cursor query
   }
   public override View NewView(Context context, ICursor cursor, ViewGroup parent)
   {
       return this.context.LayoutInflater.Inflate(Android.Resource.Layout.SimpleListItem1, parent, false);
   }
}
```

Assigning the CursorAdapter

In the `Activity` that will display the `ListView`, create the cursor and `CursorAdapter` then assign it to the list view.

The code that performs this action in the **CursorTableAdapter/HomeScreen.cs** `OnCreate` method is shown here:

```
// create the cursor
vdb = new VegetableDatabase(this);
cursor = vdb.ReadableDatabase.RawQuery("SELECT * FROM vegetables", null);
StartManagingCursor(cursor);

// create the CursorAdapter
listView.Adapter = (IListAdapter)new HomeScreenCursorAdapter(this, cursor, false);
```

The `OnDestroy` method contains the `StopManagingCursor` method call described previously.

Related Links

- SimpleCursorTableAdapter (sample)
- CursorTableAdapter (sample)

Using a ContentProvider with Xamarin.Android

10/28/2019 • 2 minutes to read • Edit Online

CursorAdapters can also be used to display data from a ContentProvider. ContentProviders allow you to access data exposed by other applications (including Android system data like contacts, media and calendar information).

The preferred way to access a ContentProvider is with a CursorLoader using the LoaderManager. LoaderManager was introduced in Android 3.0 (API Level 11, Honeycomb) to move blocking tasks off the main thread, and using a CursorLoader allows the data to be loaded in a thread before being bound to a ListView for display.

Refer to Intro to ContentProviders for more information.

Xamarin.Android ListView and the Activity Lifecycle

10/28/2019 • 2 minutes to read • Edit Online

Activities go through certain states as your application runs, such as starting up, running, being paused and being stopped. For more information, and specific guidelines on handling state transitions, see the Activity Lifecycle Tutorial. It is important to understand the activity lifecycle and place your `ListView` code in the correct locations.

All of the examples in this document perform 'setup tasks' in the Activity's `OnCreate` method and (when required) perform 'teardown' in `OnDestroy`. The examples generally use small data sets that do not change, so re-loading the data more frequently is unnecessary.

However, if your data is frequently changing or uses a lot of memory it might be appropriate to use different lifecycle methods to populate and refresh your `ListView`. For example, if the underlying data is constantly changing (or may be affected by updates on other activities) then creating the adapter in `OnStart` or `OnResume` will ensure the latest data is displayed each time the Activity is shown.

If the Adapter uses resources like memory, or a managed cursor, remember to release those resources in the complementary method to where they were instantiated (eg. objects created in `OnStart` can be disposed of in `OnStop`).

Configuration Changes

It's important to remember that configuration changes – especially screen rotation and keyboard visibility – can cause the current activity to be destroyed and re-created (unless you specify otherwise using the `ConfigurationChanges` attribute). This means that under normal conditions, rotating a device will cause a `ListView` and `Adapter` to be re-created and (unless you have written code in `OnPause` and `OnResume`) the scroll position and row selection states will be lost.

The following attribute would prevent an activity from being destroyed and recreated as a result of configuration changes:

```
[Activity(ConfigurationChanges="keyboardHidden|orientation")]
```

The Activity should then override `OnConfigurationChanged` to respond to those changes appropriately. For more details on how to handle configuration changes see the documentation.

Xamarin.Android GridView

7/8/2021 • 3 minutes to read • Edit Online

`GridView` is a `ViewGroup` that displays items in a two-dimensional, scrollable grid. The grid items are automatically inserted to the layout using a `ListAdapter`.

In this tutorial, you'll create a grid of image thumbnails. When an item is selected, a toast message will display the position of the image.

Start a new project named **HelloGridView**.

Find some photos you'd like to use, or download these sample images. Add the image files to the project's **Resources/Drawable** directory. In the **Properties** window, set the Build Action for each to **AndroidResource**.

Open the **Resources/Layout/Main.axml** file and insert the following:

```
<?xml version="1.0" encoding="utf-8"?>
<GridView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/gridview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:columnWidth="90dp"
    android:numColumns="auto_fit"
    android:verticalSpacing="10dp"
    android:horizontalSpacing="10dp"
    android:stretchMode="columnWidth"
    android:gravity="center"
/>
```

This `GridView` will fill the entire screen. The attributes are rather self explanatory. For more information about valid attributes, see the `GridView` reference.

Open `HelloGridView.cs` and insert the following code for the `OnCreate()` method:

```
protected override void OnCreate (Bundle bundle)
{
    base.OnCreate (bundle);

    SetContentView (Resource.Layout.Main);

    var gridview = FindViewById<GridView> (Resource.Id.gridview);
    gridview.Adapter = new ImageAdapter (this);

    gridview.ItemClick += delegate (object sender, AdapterView.ItemClickEventArgs args) {
        Toast.MakeText (this, args.Position.ToString (), ToastLength.Short).Show ();
    };
}
```

After the **Main.axml** layout is set for the content view, the `GridView` is captured from the layout with `FindViewById`. The `Adapter` property is then used to set a custom adapter (`ImageAdapter`) as the source for all items to be displayed in the grid. The `ImageAdapter` is created in the next step.

To do something when an item in the grid is clicked, an anonymous delegate is subscribed to the `ItemClick` event. It shows a `Toast` that displays the index position (zero-based) of the selected item (in a real world scenario, the position could be used to get the full sized image for some other task). Note that Java-style listener classes can be used instead of .NET events.

Create a new class called `ImageAdapter` that subclasses `BaseAdapter`:

```
public class ImageAdapter : BaseAdapter
{
    Context context;

    public ImageAdapter (Context c)
    {
        context = c;
    }

    public override int Count {
        get { return thumbIds.Length; }
    }

    public override Java.Lang.Object GetItem (int position)
    {
        return null;
    }

    public override long GetItemId (int position)
    {
        return 0;
    }

    // create a new ImageView for each item referenced by the Adapter
    public override View GetView (int position, View convertView, ViewGroup parent)
    {
        ImageView imageView;

        if (convertView == null) {  // if it's not recycled, initialize some attributes
            imageView = new ImageView (context);
            imageView.LayoutParameters = new GridView.LayoutParams (85, 85);
            imageView.SetScaleType (ImageView.ScaleType.CenterCrop);
            imageView.SetPadding (8, 8, 8, 8);
        } else {
            imageView = (ImageView)convertView;
        }

        imageView.SetImageResource (thumbIds[position]);
        return imageView;
    }

    // references to our images
    int[] thumbIds = {
        Resource.Drawable.sample_2, Resource.Drawable.sample_3,
        Resource.Drawable.sample_4, Resource.Drawable.sample_5,
        Resource.Drawable.sample_6, Resource.Drawable.sample_7,
        Resource.Drawable.sample_0, Resource.Drawable.sample_1,
        Resource.Drawable.sample_2, Resource.Drawable.sample_3,
        Resource.Drawable.sample_4, Resource.Drawable.sample_5,
        Resource.Drawable.sample_6, Resource.Drawable.sample_7,
        Resource.Drawable.sample_0, Resource.Drawable.sample_1,
        Resource.Drawable.sample_2, Resource.Drawable.sample_3,
        Resource.Drawable.sample_4, Resource.Drawable.sample_5,
        Resource.Drawable.sample_6, Resource.Drawable.sample_7
    };
}
```

First, this implements some required methods inherited from `BaseAdapter`. The constructor and the `Count` property are self-explanatory. Normally, `GetItem(int)` should return the actual object at the specified position in the adapter, but it's ignored for this example. Likewise, `GetItemId(int)` should return the row id of the item, but it's not needed here.

The first method necessary is `GetView()`. This method creates a new `View` for each image added to the `ImageAdapter`. When this is called, a `View` is passed in, which is normally a recycled object (at least after this has been called once), so there's a check to see if the object is null. If it *is* null, an `ImageView` is instantiated and configured with desired properties for the image presentation:

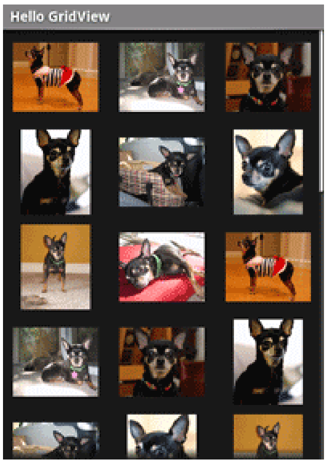
- `LayoutParams` sets the height and width for the View—this ensures that, no matter the size of the drawable, each image is resized and cropped to fit in these dimensions, as appropriate.
- `SetScaleType()` declares that images should be cropped toward the center (if necessary).
- `SetPadding(int, int, int, int)` defines the padding for all sides. (Note that, if the images have different aspect-ratios, then less padding will cause for more cropping of the image if it does not match the dimensions given to the ImageView.)

If the `View` passed to `GetView()` is *not* null, then the local `ImageView` is initialized with the recycled `View` object.

At the end of the `GetView()` method, the `position` integer passed into the method is used to select an image from the `thumbIds` array, which is set as the image resource for the `ImageView`.

All that's left is to define the `thumbIds` array of drawable resources.

Run the application. Your grid layout should look something like this:



Try experimenting with the behaviors of the `GridView` and `ImageView` elements by adjusting their properties. For example, instead of using `LayoutParams` try using `SetAdjustViewBounds()`.

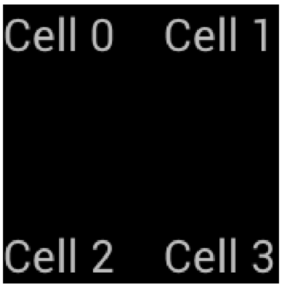
References

- `GridView`
- `ImageView`
- `BaseAdapter`

Xamarin.Android GridLayout

7/8/2021 • 3 minutes to read • Edit Online

The `GridLayout` is a new `ViewGroup` subclass that supports laying out views in a 2D grid, similar to an HTML table, as shown below:



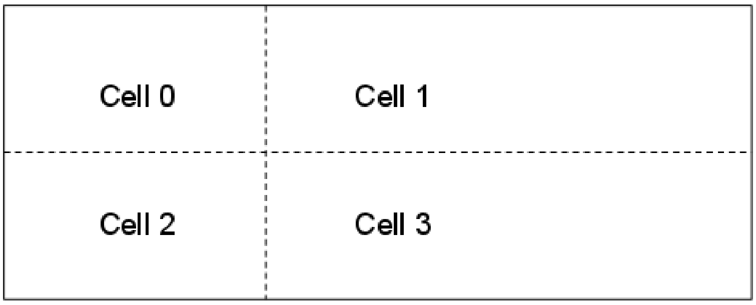
`GridLayout` works with a flat-view hierarchy, where child views set their locations in the grid by specifying the rows and columns they should be in. This way, the *GridLayout* is able to position views in the grid without requiring that any intermediate views provide a table structure, such as seen in the table rows used in the TableLayout. By maintaining a flat hierarchy, *GridLayout* is able to more swiftly layout its child views. Let's take a look at an example to illustrate what this concept actually means in code.

Creating a Grid Layout

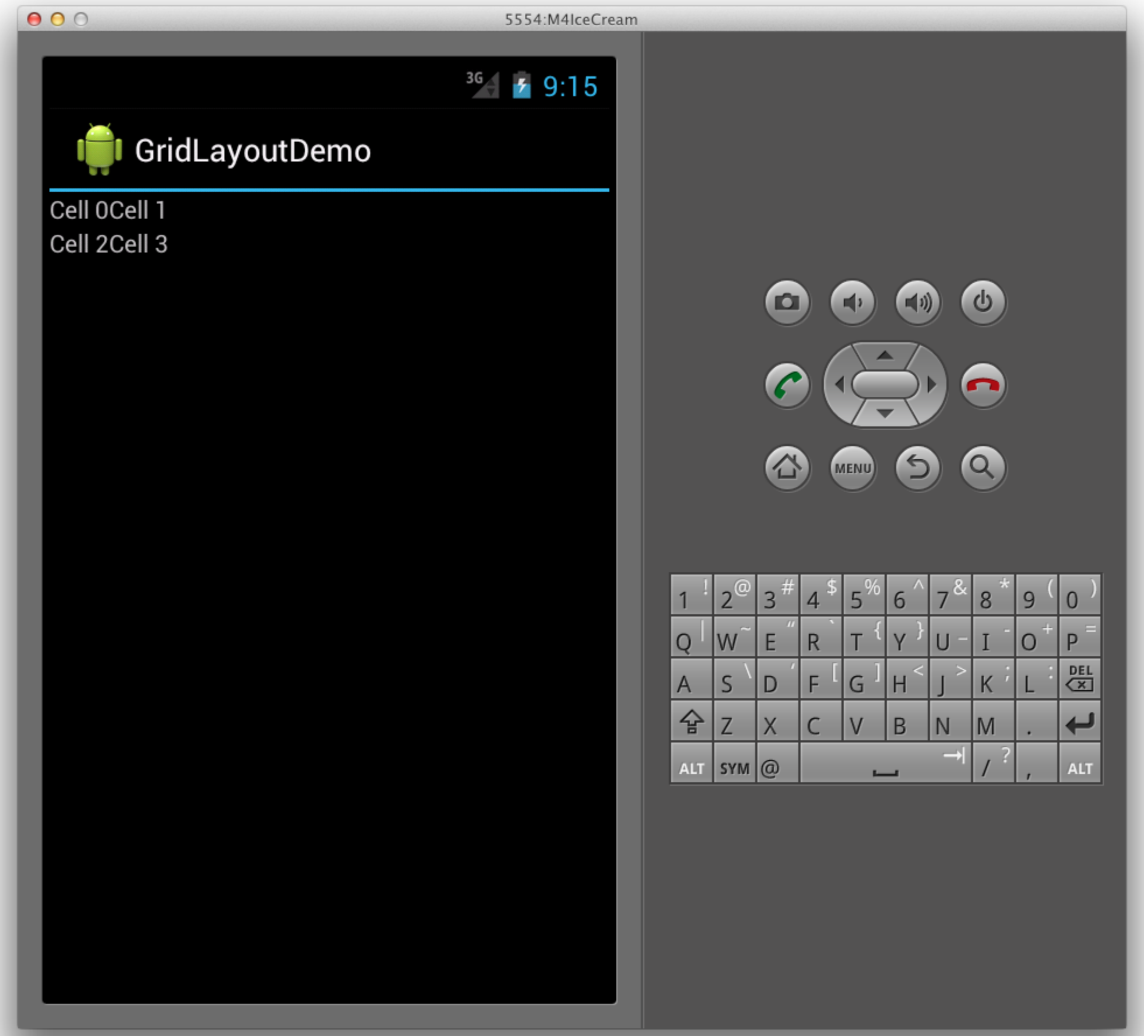
The following XML adds several `TextView` controls to a *GridLayout*.

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:rowCount="2"
        android:columnCount="2">
     <TextView
            android:text="Cell 0"
            android:textSize="14dip" />
     <TextView
            android:text="Cell 1"
            android:textSize="14dip" />
     <TextView
            android:text="Cell 2"
            android:textSize="14dip" />
     <TextView
            android:text="Cell 3"
            android:textSize="14dip" />
</GridLayout>
```

The layout will adjust the row and column sizes so that the cells can fit their content, as illustrated by the following diagram:



This results in the following user interface when run in an application:

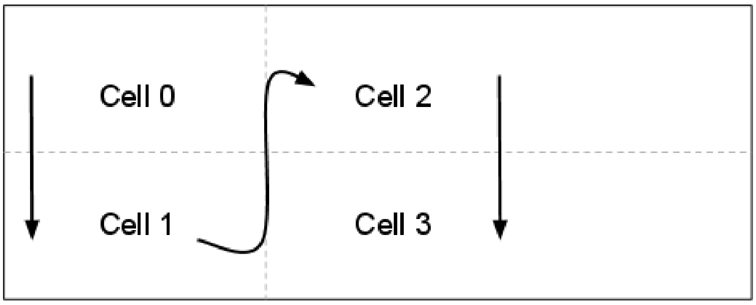


Specifying Orientation

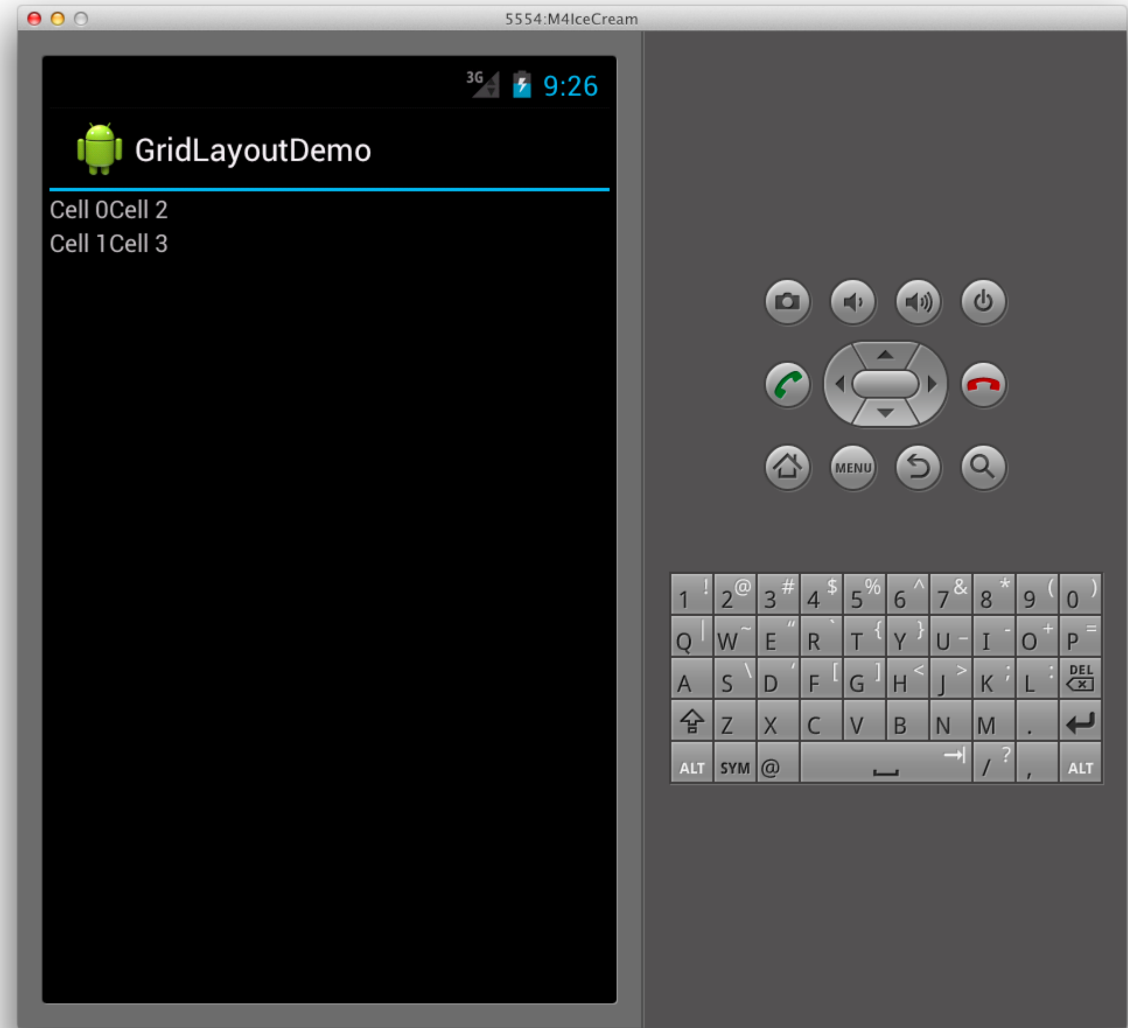
Notice in the XML above, each `TextView` does not specify a row or column. When these are not specified, the `GridLayout` assigns each child view in order, based upon the orientation. For example, let's change the GridLayout's orientation from the default, which is horizontal, to vertical like this:

```
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:rowCount="2"
        android:columnCount="2"
        android:orientation="vertical">
</GridLayout>
```

Now, the `GridLayout` will position the cells from top to bottom in each column, instead of left to right, as shown below:



This results in the following user interface at runtime:



Specifying Explicit Position

If we want to explicitly control the positions of the child views in the `GridLayout`, we can set their `layout_row` and `layout_column` attributes. For example, the following XML will result in the layout shown in the first screenshot (shown above), regardless of the orientation.

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:rowCount="2"
        android:columnCount="2">
     <TextView
            android:text="Cell 0"
            android:textSize="14dip"
            android:layout_row="0"
            android:layout_column="0" />
     <TextView
            android:text="Cell 1"
            android:textSize="14dip"
            android:layout_row="0"
            android:layout_column="1" />
     <TextView
            android:text="Cell 2"
            android:textSize="14dip"
            android:layout_row="1"
            android:layout_column="0" />
     <TextView
            android:text="Cell 3"
            android:textSize="14dip"
            android:layout_row="1"
            android:layout_column="1"  />
</GridLayout>
```

Specifying spacing

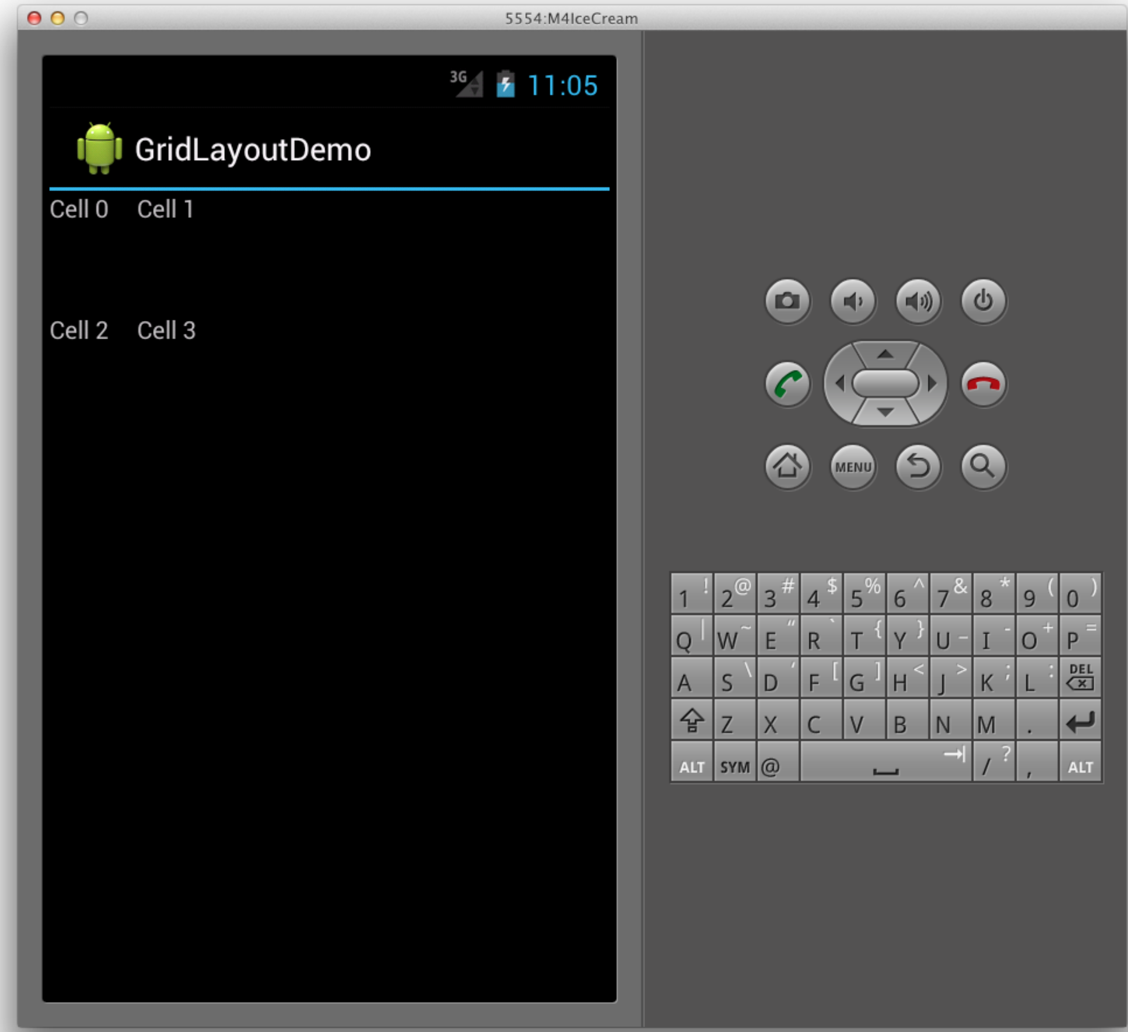
We have a couple of options that will provide spacing between the child views of the `GridLayout`. We can use the `layout_margin` attribute to set the margin on each child view directly, as shown below

```
<TextView
            android:text="Cell 0"
            android:textSize="14dip"
            android:layout_row="0"
            android:layout_column="0"
            android:layout_margin="10dp" />
```

Additionally, in Android 4, a new general-purpose spacing view called `Space` is now available. To use it, simply add it as a child view. For example, the XML below adds an additional row to the `GridLayout` by setting its `rowcount` to 3, and adds a `Space` view that provides spacing between the `TextViews`.

```xml
<?xml version="1.0" encoding="utf-8"?>
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:rowCount="3"
        android:columnCount="2"
        android:orientation="vertical">
     <TextView
            android:text="Cell 0"
            android:textSize="14dip"
            android:layout_row="0"
            android:layout_column="0" />
     <TextView
            android:text="Cell 1"
            android:textSize="14dip"
            android:layout_row="0"
            android:layout_column="1" />
     <Space
            android:layout_row="1"
            android:layout_column="0"
            android:layout_width="50dp"
            android:layout_height="50dp" />
     <TextView
            android:text="Cell 2"
            android:textSize="14dip"
            android:layout_row="2"
            android:layout_column="0" />
     <TextView
            android:text="Cell 3"
            android:textSize="14dip"
            android:layout_row="2"
            android:layout_column="1" />
</GridLayout>
```

This XML creates spacing in the `GridLayout` as shown below:



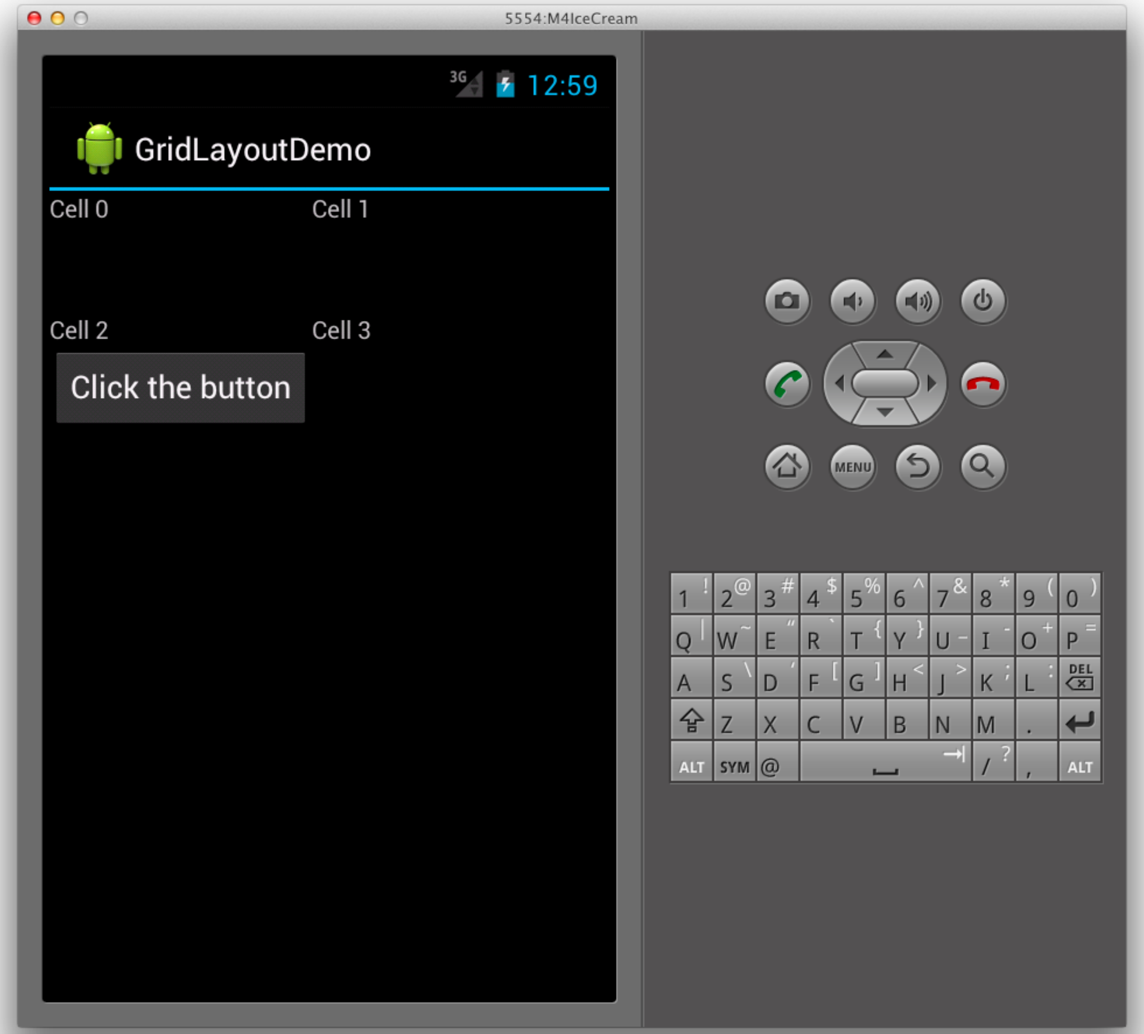
The benefit of using the new `Space` view is that it allows for spacing and doesn't require us to set attributes on every child view.

Spanning Columns and Rows

The `GridLayout` also supports cells that span multiple columns and rows. For example, say we add another row containing a button to the `GridLayout` as shown below:

```xml
<?xml version="1.0" encoding="utf-8"?>
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:rowCount="4"
        android:columnCount="2"
        android:orientation="vertical">
     <TextView
            android:text="Cell 0"
            android:textSize="14dip"
            android:layout_row="0"
            android:layout_column="0" />
     <TextView
            android:text="Cell 1"
            android:textSize="14dip"
            android:layout_row="0"
            android:layout_column="1" />
     <Space
            android:layout_row="1"
            android:layout_column="0"
            android:layout_width="50dp"
            android:layout_height="50dp" />
     <TextView
            android:text="Cell 2"
            android:textSize="14dip"
            android:layout_row="2"
            android:layout_column="0" />
     <TextView
            android:text="Cell 3"
            android:textSize="14dip"
            android:layout_row="2"
            android:layout_column="1" />
     <Button
            android:id="@+id/myButton"
            android:text="@string/hello"
            android:layout_row="3"
            android:layout_column="0" />
</GridLayout>
```

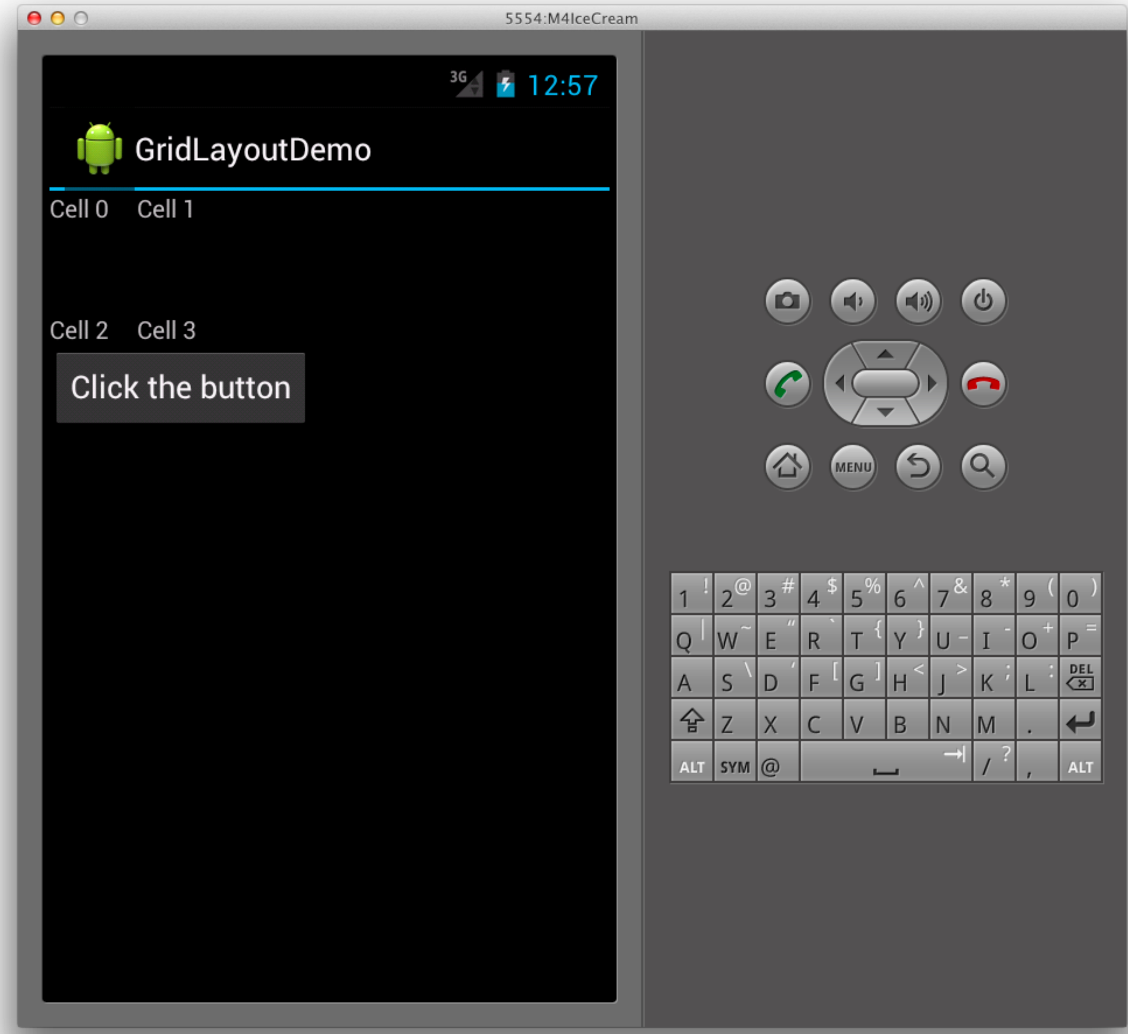
This will result in the first column of the `GridLayout` being stretched to accommodate the size of the button, as we see here:



To keep the first column from stretching, we can set the button to span two columns by setting its columnspan like this:

```
<Button
    android:id="@+id/myButton"
    android:text="@string/hello"
    android:layout_row="3"
    android:layout_column="0"
    android:layout_columnSpan="2" />
```

Doing this results in a layout for the `TextViews` that is similar to the layout we had earlier, with the button added to the bottom of the `GridLayout` as shown below:



Related Links

- GridLayoutDemo (sample)

Tabbed Layouts

10/28/2019 • 2 minutes to read • Edit Online

Overview

Tabs are a popular user interface pattern in mobile applications because of their simplicity and usability. They provide a consistent, easy way to navigate between various screens in an application. Android has several API's for tabbed interfaces:

- **ActionBar** – This is part of a new set of API's that was introduced in Android 3.0 (API level 11) with goal of providing a consistent navigation and view-switching interface. It has been back ported to Android 2.2 (API level 8) with the Android Support Library v7.
- **PagerTabStrip** – Indicates the current, next, and previous pages of a `ViewPager`. `ViewPager` is available only via Android Support Library v4. For more information about `PagerTabStrip`, see ViewPager.
- **Toolbar** – `Toolbar` is a newer and more flexible action bar component that replaces `ActionBar`. `Toolbar` is available in Android 5.0 Lollipop or later, and it is also available for older versions of Android via the Android Support Library v7 NuGet package. `Toolbar` is currently the recommended action bar component to use in Android apps. For more information, see Toolbar.

Related Links

- Material Design - Tabs- ActionBar
- Android Support Library v7 AppCompat NuGet Package
- v7 appcompat library

Tabbed Layouts with the ActionBar

7/8/2021 • 5 minutes to read • Edit Online

This guide introduces and explains how to use the ActionBar APIs to create a tabbed user interface in a Xamarin.Android application.

Overview

The action bar is an Android UI pattern that is used to provide a consistent user interface for key features such as tabs, application identity, menus, and search. In Android 3.0 (API level 11), Google introduced the ActionBar APIs to the Android platform. The ActionBar APIs introduce UI themes to provide a consistent look and feel and classes that allow for tabbed user interfaces. This guide discusses how to add Action Bar tabs to a Xamarin.Android application. It also discusses how to use the Android Support Library v7 to backport ActionBar tabs to Xamarin.Android applications targeting Android 2.1 to Android 2.3.

Note that `Toolbar` is a newer and more generalized action bar component that you should use instead of `ActionBar` (`Toolbar` was designed to replace `ActionBar`). For more information, see Toolbar.

Requirements

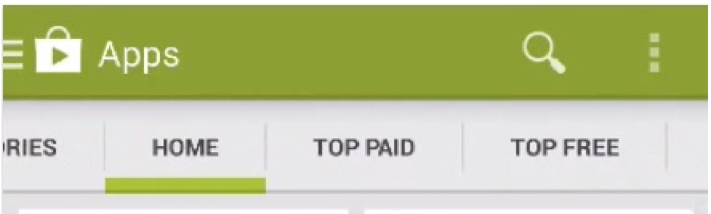
Any Xamarin.Android application that targets API level 11 (Android 3.0) or higher has access to the ActionBar APIs as a part of the native Android APIs.

Some of the ActionBar APIs have been back ported to API level 7 (Android 2.1) and are available via the V7 AppCompat Library, which is made available to Xamarin.Android apps via the Xamarin Android Support Library - V7 package.

Introducing Tabs in the ActionBar

The action bar tries to display all of its tabs concurrently and make all the tabs equal in size based on the width of the widest tab label. This is illustrated by the following screenshot:

When the ActionBar can't display all of the tabs, it will set up the tabs in a horizontally scrollable view. The user may swipe left or right to see the remaining tabs. This screenshot from Google Play shows an example of this:



Each tab in the action bar should be associated with a *fragment*. When the user selects a tab, the application will display the fragment that is associated with the tab. The ActionBar is not responsible for displaying the appropriate fragment to the user. Instead, the ActionBar will notify an application about state changes in a tab through a class that implements the ActionBar.ITabListener interface. This interface provides three callback methods that Android will invoke when the state of the tab changes:

- **OnTabSelected** - This method is called when the user selects the tab. It should display the fragment.
- **OnTabReselected** - This method is called when the tab is already selected but is selected again by the user. This callback is typically used to refresh/update the displayed fragment.
- **OnTabUnselected** - This method is called when the user selects another tab. This callback is used to save the state in the displayed fragment before it disappears.

Xamarin.Android wraps the `ActionBar.ITabListener` with events on the `ActionBar.Tab` class. Applications may assign event handlers to one or more of these events. There are three events (one for each method in `ActionBar.ITabListener`) that an action bar tab will raise:

- TabSelected
- TabReselected
- TabUnselected

Adding Tabs to the ActionBar

The ActionBar is native to Android 3.0 (API level 11) and higher and is available to any Xamarin.Android application that targets this API as a minimum.

The following steps illustrate how to add ActionBar tabs to an Android Activity:

1. In the `OnCreate` method of an Activity – *before initializing any UI widgets* – an application must set the `NavigationMode` on the `ActionBar` to `ActionBar.NavigationModeTabs` as shown in this code snippet:

   ```
   ActionBar.NavigationMode = ActionBarNavigationMode.Tabs;
   SetContentView(Resource.Layout.Main);
   ```

2. Create a new tab using `ActionBar.NewTab()`.
3. Assign event handlers or provide a custom `ActionBar.ITabListener` implementation that will respond to the events that are raised when the user interacts with the ActionBar tabs.
4. Add the tab that was created in the previous step to the `ActionBar`.

The following code is one example of using these steps to add tabs to an application that uses event handlers to respond to state changes:

```
protected override void OnCreate(Bundle bundle)
{
    ActionBar.NavigationMode = ActionBarNavigationMode.Tabs;
    SetContentView(Resource.Layout.Main);

    ActionBar.Tab tab = ActionBar.NewTab();
    tab.SetText(Resources.GetString(Resource.String.tab1_text));
    tab.SetIcon(Resource.Drawable.tab1_icon);
    tab.TabSelected += (sender, args) => {
                           // Do something when tab is selected
                       };
    ActionBar.AddTab(tab);

    tab = ActionBar.NewTab();
    tab.SetText(Resources.GetString(Resource.String.tab2_text));
    tab.SetIcon(Resource.Drawable.tab2_icon);
    tab.TabSelected += (sender, args) => {
                           // Do something when tab is selected
                       };
    ActionBar.AddTab(tab);
}
```

Event Handlers vs ActionBar.ITabListener

Applications should use event handlers and `ActionBar.ITabListener` for different scenarios. Event handlers do offer a certain amount of syntactic convenience; they save you from having to create a class and implement `ActionBar.ITabListener`. This convenience does come at a cost – Xamarin.Android performs this transformation for you, creating one class and implementing `ActionBar.ITabListener` for you. This is fine when an application has a limited number of tabs.

When dealing with many tabs, or sharing common functionality between ActionBar tabs, it can be more efficient in terms of memory and performance to create a custom class that implements `ActionBar.ITabListener`, and sharing a single instance of the class. This will reduce the number of GREF's that a Xamarin.Android application is using.

Backwards Compatibility for Older Devices

The Android Support Library v7 AppCompat back ports ActionBar tabs to Android 2.1 (API level 7). Tabs are accessible in a Xamarin.Android application once this component has been added to the project.

To use the ActionBar, an activity must subclass `ActionBarActivity` and use the AppCompat theme as shown in the following code snippet:

```
[Activity(Label = "@string/app_name", Theme = "@style/Theme.AppCompat", MainLauncher = true, Icon =
"@drawable/ic_launcher")]
public class MainActivity: ActionBarActivity
```

An Activity may obtain a reference to its ActionBar from the `ActionBarActivity.SupportingActionBar` property. The following code snippet illustrates an example of setting up the ActionBar in an Activity:

```
[Activity(Label = "@string/app_name", Theme = "@style/Theme.AppCompat", MainLauncher = true, Icon =
"@drawable/ic_launcher")]
public class MainActivity : ActionBarActivity, ActionBar.ITabListener
{
    static readonly string Tag = "ActionBarTabsSupport";

    public void OnTabReselected(ActionBar.Tab tab, FragmentTransaction ft)
    {
        // Optionally refresh/update the displayed tab.
        Log.Debug(Tag, "The tab {0} was re-selected.", tab.Text);
    }

    public void OnTabSelected(ActionBar.Tab tab, FragmentTransaction ft)
    {
        // Display the fragment the user should see
        Log.Debug(Tag, "The tab {0} has been selected.", tab.Text);
    }

    public void OnTabUnselected(ActionBar.Tab tab, FragmentTransaction ft)
    {
        // Save any state in the displayed fragment.
        Log.Debug(Tag, "The tab {0} as been unselected.", tab.Text);
    }

    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);
        SupportActionBar.NavigationMode = ActionBar.NavigationModeTabs;
        SetContentView(Resource.Layout.Main);
    }

    void AddTabToActionBar(int labelResourceId, int iconResourceId)
    {
        ActionBar.Tab tab = SupportActionBar.NewTab()
                                            .SetText(labelResourceId)
                                            .SetIcon(iconResourceId)
                                            .SetTabListener(this);
        SupportActionBar.AddTab(tab);
    }
}
```

Summary

In this guide we discussed how to create a tabbed user interface in a Xamarin.Android using the ActionBar. We covered how to add tabs to the ActionBar and how an Activity can interact with tab events via the `ActionBar.ITabListener` interface. We also saw how the Android Support Library v7 AppCompat package backports the ActionBar tabs to older versions of Android.

Related Links

- ActionBarTabs (sample)
- Toolbar
- Fragments
- ActionBar
- ActionBarActivity
- Action Bar Pattern
- Android v7 AppCompat
- Xamarin.Android Support Library v7 AppCompat NuGet Package

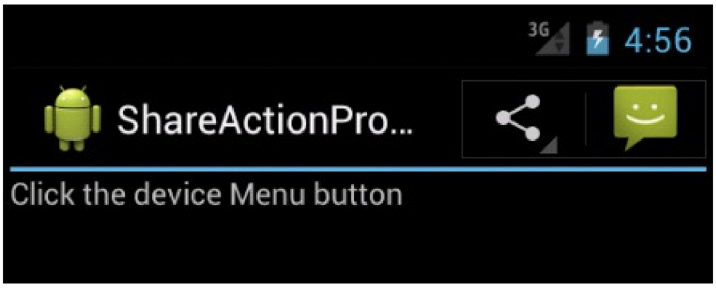
Xamarin.Android Controls (Widgets)

7/8/2021 • 2 minutes to read • Edit Online

Xamarin.Android exposes all of the native user interface controls (widgets) provided by Android. These controls can be easily added to Xamarin.Android apps using the Android Designer or programatically via XML layout files. Regardless of which method you choose, Xamarin.Android exposes all of the user interface object properties and methods in C#. The following sections introduce the most common Android user interface controls and explain how to incorporate them into Xamarin.Android apps.

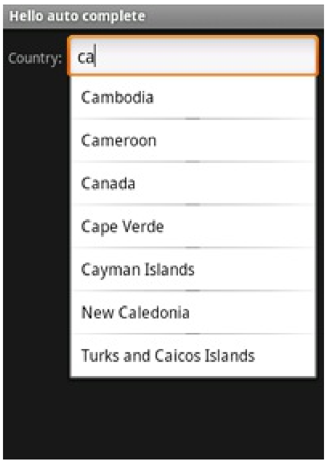
Action Bar

`ActionBar` is a toolbar that displays the activity title, navigation interfaces, and other interactive items. Typically, the action bar appears at the top of an activity's window.



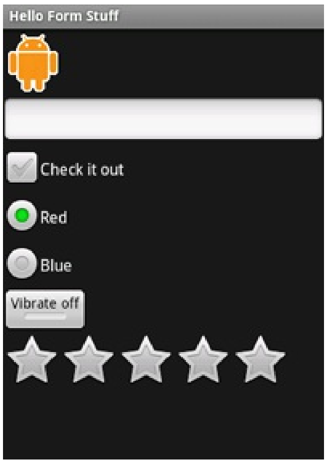
Auto Complete

`AutoCompleteTextView` is an editable text view element that shows completion suggestions automatically while the user is typing. The list of suggestions is displayed in a drop down menu from which the user can choose an item to replace the content of the edit box with.



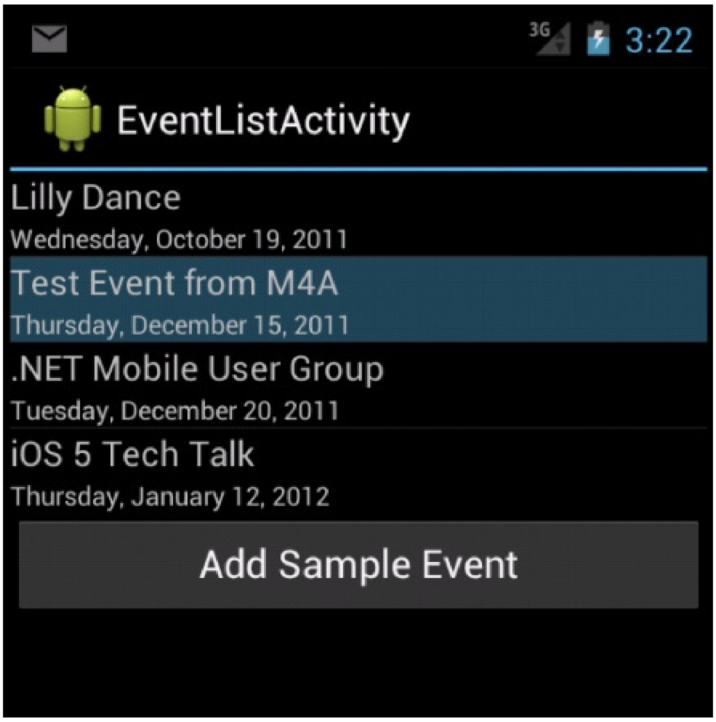
Buttons

Buttons are UI elements that the user taps to perform an action.



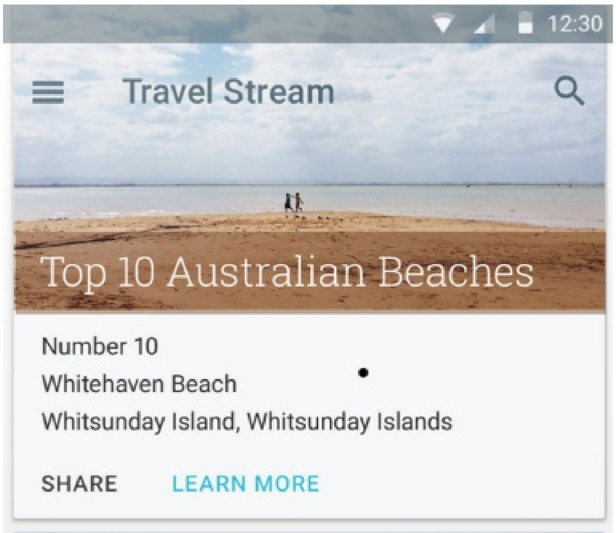
Calendar

The `Calendar` class is used for converting a specific instance in time (a millisecond value that is offset from the epoch) to values such as year, month, hour, day of the month, and the date of the next week. `Calendar` supports a wealth of interaction options with calendar data, including the ability to read and write events, attendees, and reminders. By using the calendar provider in your application, data you add through the API will appear in the built-in calendar app that comes with Android.



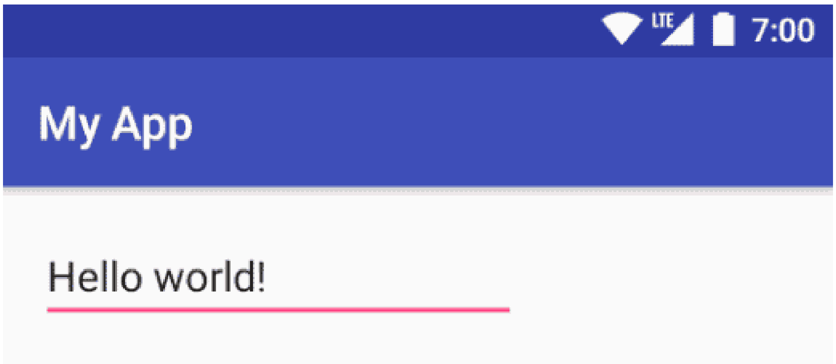
CardView

`CardView` is a UI component that presents text and image content in views that resemble cards. `CardView` is implemented as a `FrameLayout` widget with rounded corners and a shadow. Typically, a `CardView` is used to present a single row item in a `ListView` or `GridView` view group.



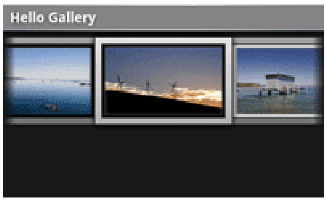
Edit Text

`EditText` is a UI element that is used for entering and modifying text.



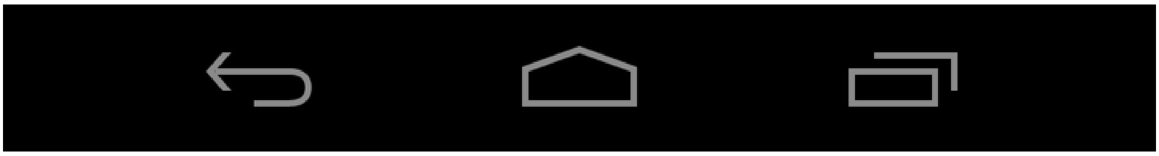
Gallery

`Gallery` is a layout widget that is used to display items in a horizontally scrolling list; it positions the current selection at the center of the view.



Navigation Bar

The *Navigation Bar* provides navigation controls on devices that do not include hardware buttons for **Home**, **Back**, and **Menu**.

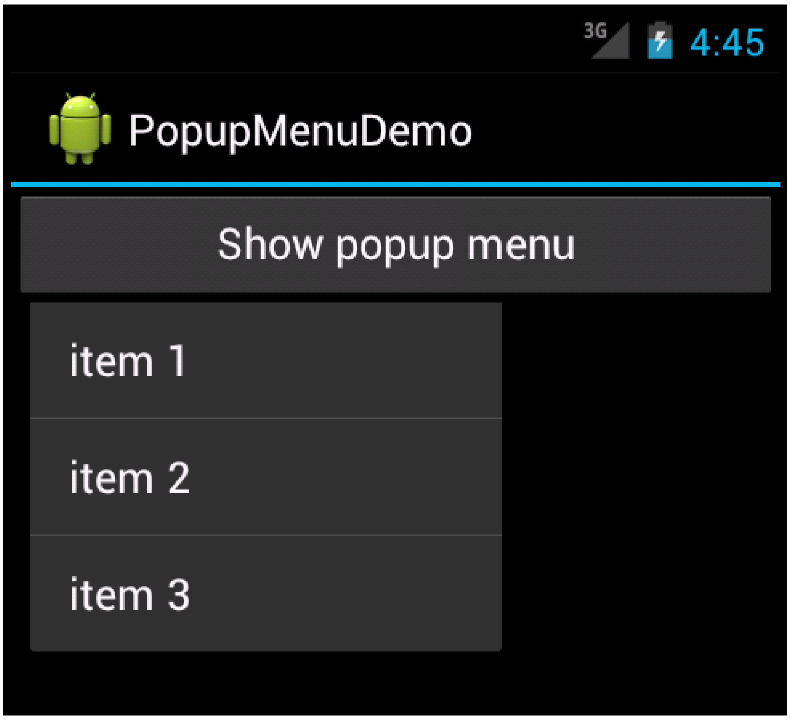


Pickers

Pickers are UI elements that allow the user to pick a date or a time by using dialogs that are provided by Android.

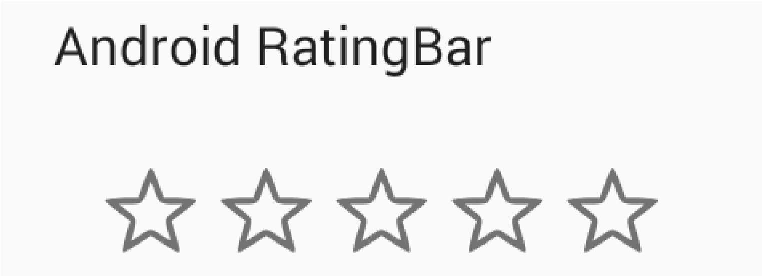
Popup Menu

`PopupMenu` is used for displaying popup menus that are attached to a particular view.



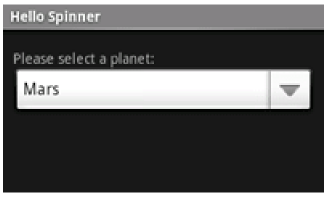
RatingBar

A `RatingBar` is a UI element that displays a rating in stars.



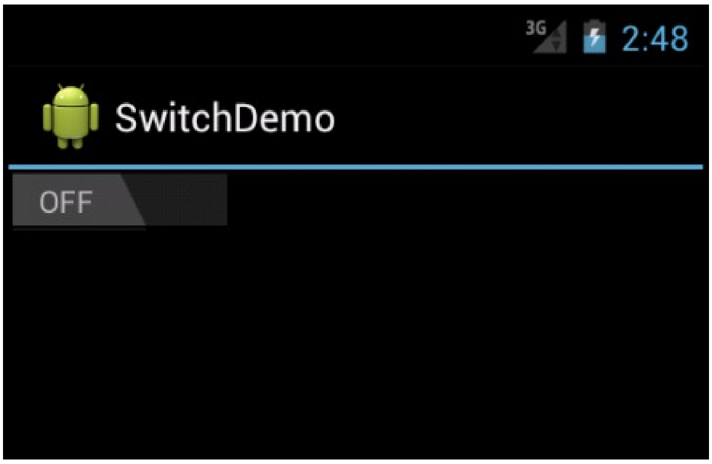
Spinner

`Spinner` is a UI element that provides a quick way to select one value from a set. It is similar to a drop-down list.



Switch

`Switch` is a UI element that allows a user to toggle between two states, such as ON or OFF. The `Switch` default value is OFF.

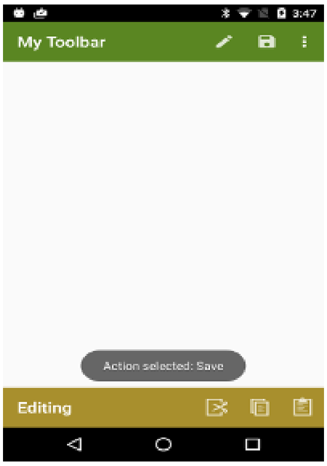


TextureView

`TextureView` is a view that uses hardware-accelerated 2D rendering to enable a video or OpenGL content stream to be displayed.

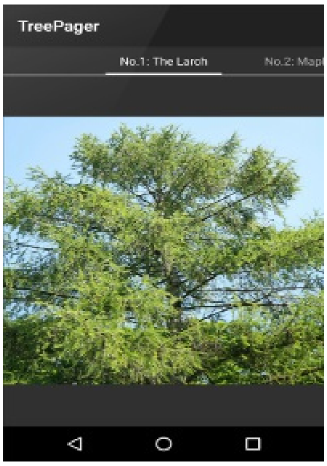
ToolBar

The `Toolbar` widget (introduced in Android 5.0 Lollipop) can be thought of as a generalization of the action bar interface – it is intended to replace the action bar. The `Toolbar` can be used anywhere in an app layout, and it is much more customizable than an action bar.



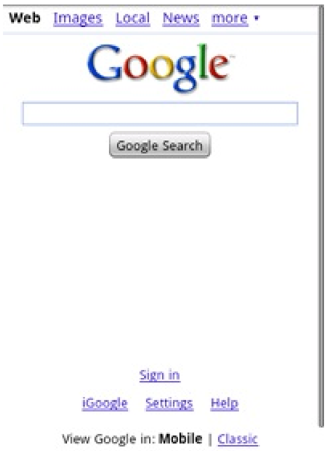
ViewPager

The `ViewPager` is a layout manager that allows the user to flip left and right through pages of data.



WebView

`WebView` is a UI element that allows you to create your own window for viewing web pages (or even develop a complete browser).



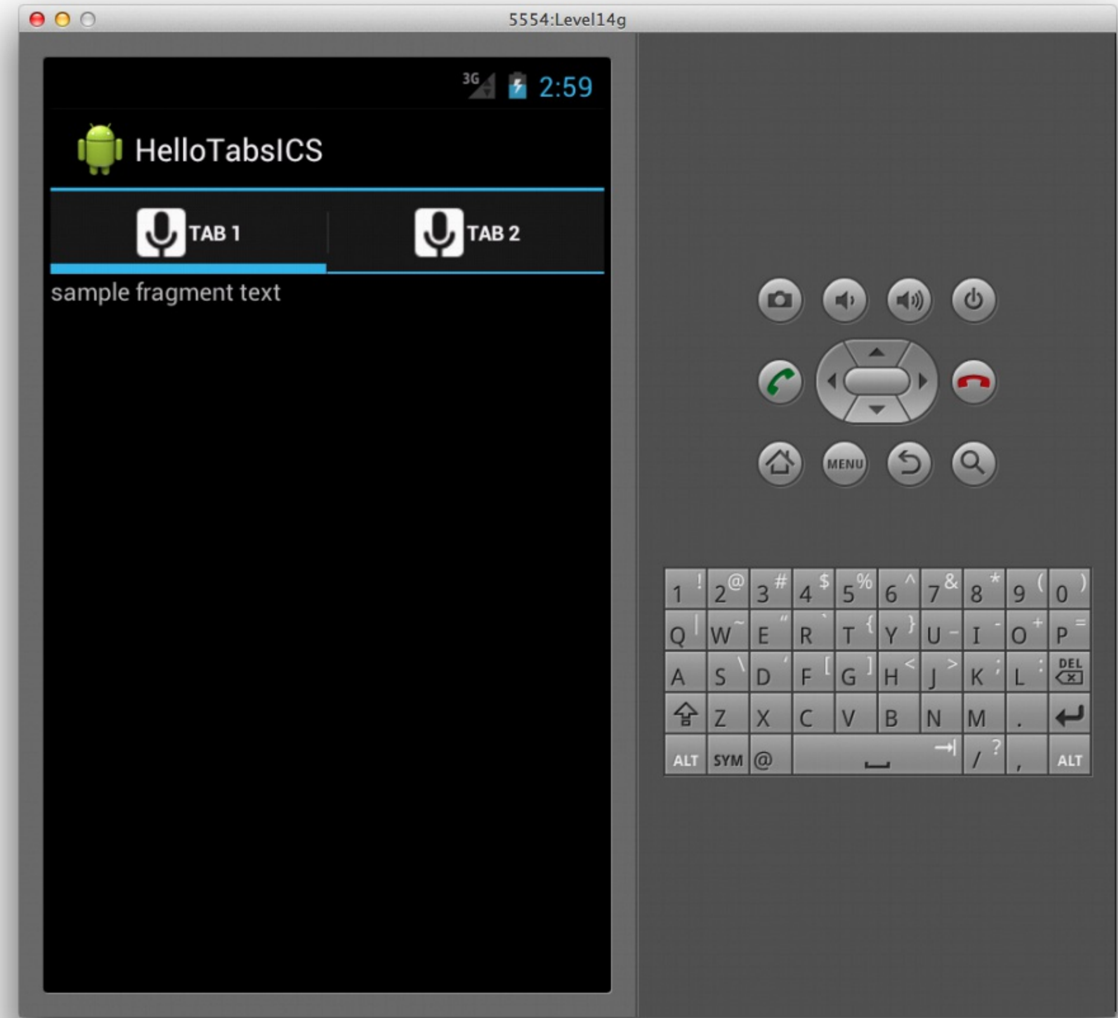
ActionBar for Xamarin.Android

7/8/2021 • 3 minutes to read • Edit Online

When using `TabActivity`, the code to create the tab icons has no effect when run against the Android 4.0 framework. Although functionally it works as it did in versions of Android prior to 2.3, the `TabActivity` class itself has been deprecated in 4.0. A new way to create a tabbed interface has been introduced that uses the Action Bar, which we'll discuss next.

Action Bar Tabs

The Action Bar includes support for adding tabbed interfaces in Android 4.0. The following screenshot shows an example of such an interface.



To create tabs in the Action Bar, we first need to set its `NavigationMode` property to support tabs. In Android 4, an `ActionBar` property is available on the Activity class, which we can use to set the `NavigationMode` like this:

```
this.ActionBar.NavigationMode = ActionBarNavigationMode.Tabs;
```

Once this is done, we can create a tab by calling the `NewTab` method on the Action Bar. With this tab instance, we can call the `SetText` and `SetIcon` methods to set the tab's label text and icon; these calls are made in order in the code shown below:

```
var tab = this.ActionBar.NewTab ();
tab.SetText (tabText);
tab.SetIcon (Resource.Drawable.ic_tab_white);
```

Before we can add the tab however, we need to handle the `TabSelected` event. In this handler, we can create the content for the tab. Action Bar tabs are designed to work with *Fragments*, which are classes that represent a portion of the user interface in an Activity. For this example, the Fragment's view contains a single `TextView`, which we inflate in our `Fragment` subclass like this:

```
class SampleTabFragment: Fragment
{
    public override View OnCreateView (LayoutInflater inflater,
        ViewGroup container, Bundle savedInstanceState)
    {
        base.OnCreateView (inflater, container, savedInstanceState);

        var view = inflater.Inflate (
            Resource.Layout.Tab, container, false);

        var sampleTextView =
            view.FindViewById<TextView> (Resource.Id.sampleTextView);
        sampleTextView.Text = "sample fragment text";

        return view;
    }
}
```

The event argument passed in the `TabSelected` event is of type `TabEventArgs`, which includes a `FragmentTransaction` property that we can use to add the fragment as shown below:

```
tab.TabSelected += delegate(object sender, ActionBar.TabEventArgs e) {
    e.FragmentTransaction.Add (Resource.Id.fragmentContainer,
        new SampleTabFragment ());
};
```

Finally, we can add the tab to the Action Bar by calling the `AddTab` method as shown in this code:

```
this.ActionBar.AddTab (tab);
```

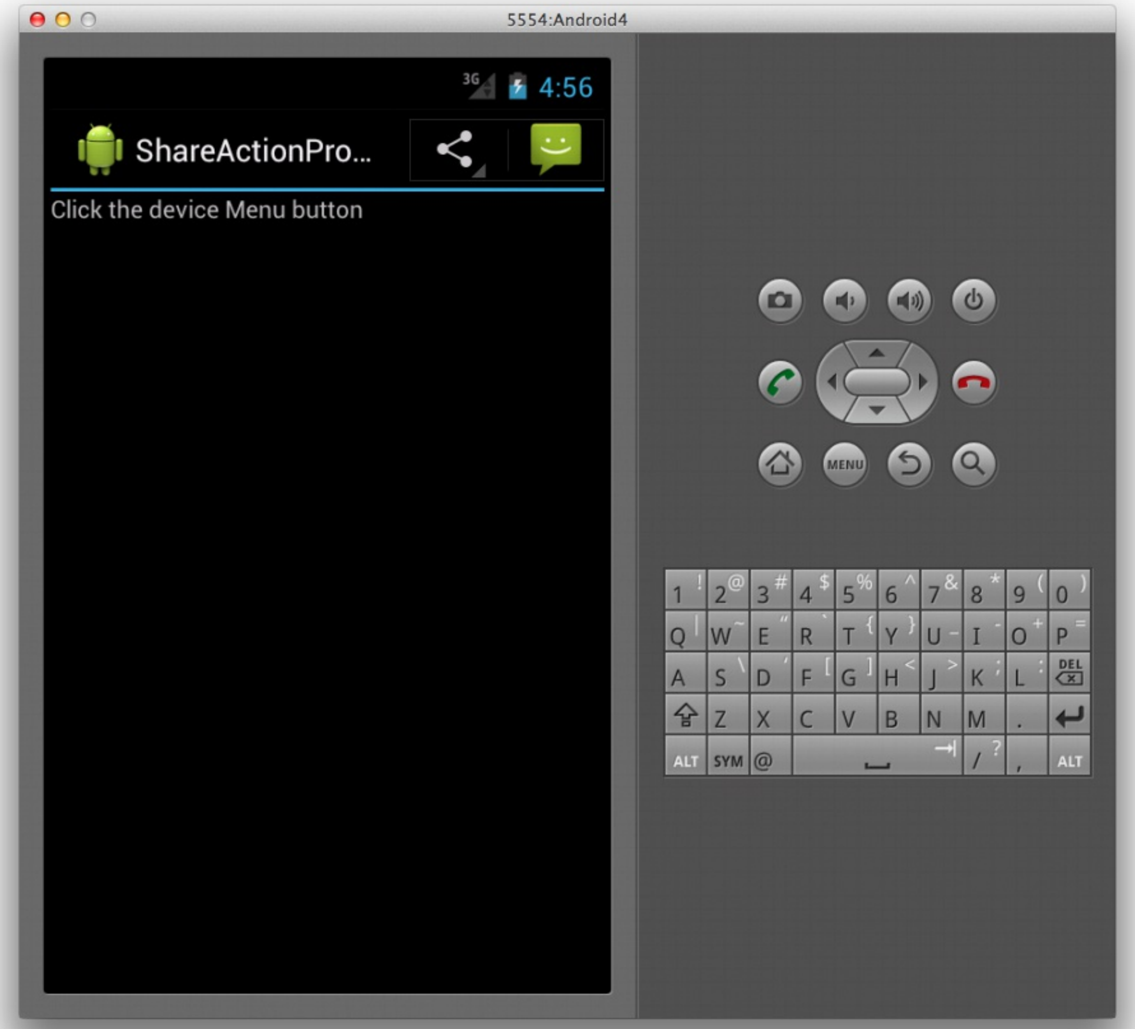
For the complete example, see the *HelloTabsICS* project in the sample code for this document.

ShareActionProvider

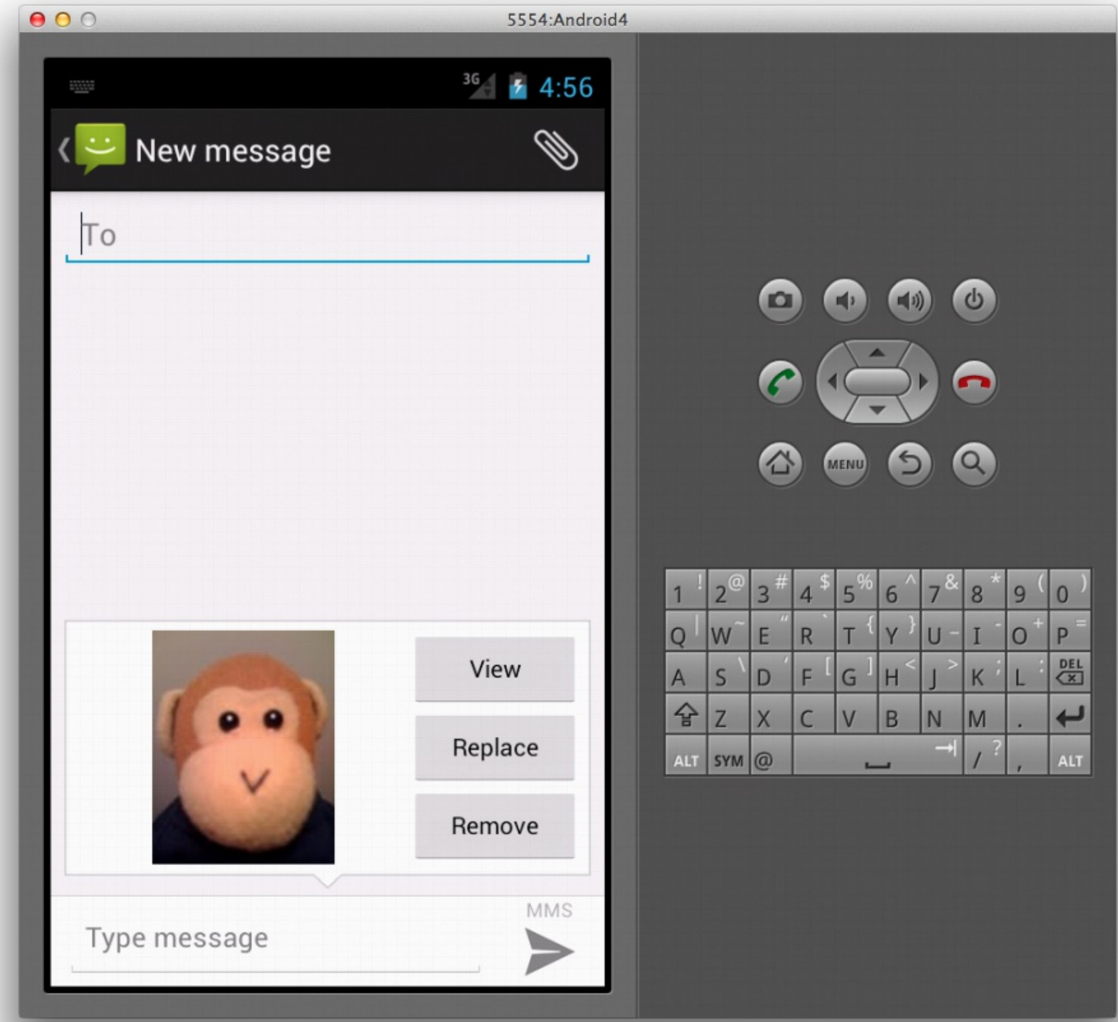
The `ShareActionProvider` class enables a sharing action to take place from an Action Bar. It takes care of creating an action view with a list of apps that can handle a sharing Intent and keeps a history of the previously used applications for easy access to them later from the Action Bar. This allows applications to share data via a user experience that's consistent throughout Android.

Image Sharing Example

For example, below is a screenshot of an Action Bar with a menu item to share an image (taken from the ShareActionProvider sample). When the user taps the menu item on the Action Bar, the ShareActionProvider loads the application to handle an Intent that is associated with the `ShareActionProvider`. In this example, the messaging application has been previously used, so it is presented on the Action Bar.



When the user clicks on the item in the Action Bar, the messaging app that contains the shared image is launched, as shown below:



Specifying the action Provider Class

To use the `ShareActionProvider`, set the `android:actionProviderClass` attribute on a menu item in the XML for the Action Bar's menu as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/shareMenuItem"
      android:showAsAction="always"
      android:title="@string/sharePicture"
      android:actionProviderClass="android.widget.ShareActionProvider" />
</menu>
```

Inflating the Menu

To inflate the menu, we override `OnCreateOptionsMenu` in the Activity subclass. Once we have a reference to the menu, we can get the `ShareActionProvider` from the `ActionProvider` property of the menu item and then use the SetShareIntent method to set the `ShareActionProvider`'s Intent, as shown below:

```
public override bool OnCreateOptionsMenu (IMenu menu)
{
    MenuInflater.Inflate (Resource.Menu.ActionBarMenu, menu);

    var shareMenuItem = menu.FindItem (Resource.Id.shareMenuItem);
    var shareActionProvider =
       (ShareActionProvider)shareMenuItem.ActionProvider;
    shareActionProvider.SetShareIntent (CreateIntent ());
}
```

Creating the Intent

The `ShareActionProvider` will use the Intent, passed to the `SetShareIntent` method in the above code, to launch the appropriate Activity. In this case we create an Intent to send an image by using the following code:

```
Intent CreateIntent ()
{
    var sendPictureIntent = new Intent (Intent.ActionSend);
    sendPictureIntent.SetType ("image/*");
    var uri = Android.Net.Uri.FromFile (GetFileStreamPath ("monkey.png"));
    sendPictureIntent.PutExtra (Intent.ExtraStream, uri);
    return sendPictureIntent;
}
```

The image in the code example above is included as an asset with the application and copied to a publicly accessible location when the Activity is created, so it will be accessible to other applications, such as the messaging app. The sample code that accompanies this article contains the full source of this example, illustrating its use.

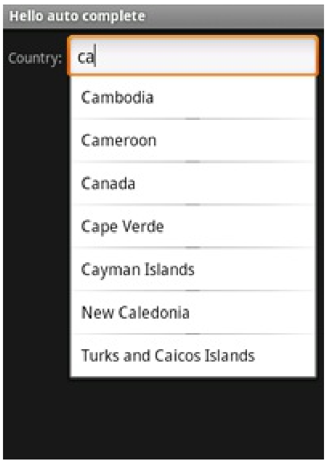
Related Links

- Hello Tabs ICS (sample)
- ShareActionProvider Demo (sample)

Auto Complete for Xamarin.Android

7/8/2021 • 4 minutes to read • Edit Online

`AutoCompleteTextView` is an editable text view element that shows completion suggestions automatically while the user is typing. The list of suggestions is displayed in a drop down menu from which the user can choose an item to replace the content of the edit box with.



Overview

To create a text entry widget that provides auto-complete suggestions, use the `AutoCompleteTextView` widget. Suggestions are received from a collection of strings associated with the widget through an `ArrayAdapter`.

In this tutorial, you will create a `AutoCompleteTextView` widget that provides suggestions for a country name.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:padding="5dp">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Country" />
    <AutoCompleteTextView android:id="@+id/autocomplete_country"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="5dp"/>
</LinearLayout>
```

The `TextView` is a label that introduces the `AutoCompleteTextView` widget.

Tutorial

Start a new project named *HelloAutoComplete*.

Create an XML file named `list_item.xml` and save it inside the **Resources/Layout** folder. Set the Build Action of this file to `AndroidResource`. Edit the file to look like this:

```xml
<?xml version="1.0" encoding="utf-8"?>

<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="10dp"
    android:textSize="16sp"
    android:textColor="#000">
</TextView>
```

This file defines a simple `TextView` that will be used for each item that appears in the list of suggestions.

Open **Resources/Layout/Main.axml** and insert the following:

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:padding="5dp">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Country" />
    <AutoCompleteTextView android:id="@+id/autocomplete_country"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="5dp"/>
</LinearLayout>
```

Open **MainActivity.cs** and insert the following code for the `OnCreate()` method:

```csharp
protected override void OnCreate (Bundle bundle)
{
    base.OnCreate (bundle);

    // Set our view from the "Main" layout resource
    SetContentView (Resource.Layout.Main);

    AutoCompleteTextView textView = FindViewById<AutoCompleteTextView> (Resource.Id.autocomplete_country);
    var adapter = new ArrayAdapter<String> (this, Resource.Layout.list_item, COUNTRIES);

    textView.Adapter = adapter;
}
```

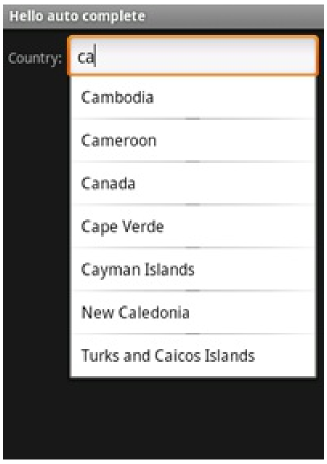
After the content view is set to the `main.xml` layout, the `AutoCompleteTextView` widget is captured from the layout with `FindViewById`. A new `ArrayAdapter` is then initialized to bind the `list_item.xml` layout to each list item in the `COUNTRIES` string array (defined in the next step). Finally, `SetAdapter()` is called to associate the `ArrayAdapter` with the `AutoCompleteTextView` widget so that the string array will populate the list of suggestions.

Inside the `MainActivity` class, add the string array:

```
static string[] COUNTRIES = new string[] {
  "Afghanistan", "Albania", "Algeria", "American Samoa", "Andorra",
  "Angola", "Anguilla", "Antarctica", "Antigua and Barbuda", "Argentina",
  "Armenia", "Aruba", "Australia", "Austria", "Azerbaijan",
  "Bahrain", "Bangladesh", "Barbados", "Belarus", "Belgium",
  "Belize", "Benin", "Bermuda", "Bhutan", "Bolivia",
  "Bosnia and Herzegovina", "Botswana", "Bouvet Island", "Brazil", "British Indian Ocean Territory",
  "British Virgin Islands", "Brunei", "Bulgaria", "Burkina Faso", "Burundi",
  "Cote d'Ivoire", "Cambodia", "Cameroon", "Canada", "Cape Verde",
  "Cayman Islands", "Central African Republic", "Chad", "Chile", "China",
  "Christmas Island", "Cocos (Keeling) Islands", "Colombia", "Comoros", "Congo",
  "Cook Islands", "Costa Rica", "Croatia", "Cuba", "Cyprus", "Czech Republic",
  "Democratic Republic of the Congo", "Denmark", "Djibouti", "Dominica", "Dominican Republic",
  "East Timor", "Ecuador", "Egypt", "El Salvador", "Equatorial Guinea", "Eritrea",
  "Estonia", "Ethiopia", "Faeroe Islands", "Falkland Islands", "Fiji", "Finland",
  "Former Yugoslav Republic of Macedonia", "France", "French Guiana", "French Polynesia",
  "French Southern Territories", "Gabon", "Georgia", "Germany", "Ghana", "Gibraltar",
  "Greece", "Greenland", "Grenada", "Guadeloupe", "Guam", "Guatemala", "Guinea", "Guinea-Bissau",
  "Guyana", "Haiti", "Heard Island and McDonald Islands", "Honduras", "Hong Kong", "Hungary",
  "Iceland", "India", "Indonesia", "Iran", "Iraq", "Ireland", "Israel", "Italy", "Jamaica",
  "Japan", "Jordan", "Kazakhstan", "Kenya", "Kiribati", "Kuwait", "Kyrgyzstan", "Laos",
  "Latvia", "Lebanon", "Lesotho", "Liberia", "Libya", "Liechtenstein", "Lithuania", "Luxembourg",
  "Macau", "Madagascar", "Malawi", "Malaysia", "Maldives", "Mali", "Malta", "Marshall Islands",
  "Martinique", "Mauritania", "Mauritius", "Mayotte", "Mexico", "Micronesia", "Moldova",
  "Monaco", "Mongolia", "Montserrat", "Morocco", "Mozambique", "Myanmar", "Namibia",
  "Nauru", "Nepal", "Netherlands", "Netherlands Antilles", "New Caledonia", "New Zealand",
  "Nicaragua", "Niger", "Nigeria", "Niue", "Norfolk Island", "North Korea", "Northern Marianas",
  "Norway", "Oman", "Pakistan", "Palau", "Panama", "Papua New Guinea", "Paraguay", "Peru",
  "Philippines", "Pitcairn Islands", "Poland", "Portugal", "Puerto Rico", "Qatar",
  "Reunion", "Romania", "Russia", "Rwanda", "Sqo Tome and Principe", "Saint Helena",
  "Saint Kitts and Nevis", "Saint Lucia", "Saint Pierre and Miquelon",
  "Saint Vincent and the Grenadines", "Samoa", "San Marino", "Saudi Arabia", "Senegal",
  "Seychelles", "Sierra Leone", "Singapore", "Slovakia", "Slovenia", "Solomon Islands",
  "Somalia", "South Africa", "South Georgia and the South Sandwich Islands", "South Korea",
  "Spain", "Sri Lanka", "Sudan", "Suriname", "Svalbard and Jan Mayen", "Swaziland", "Sweden",
  "Switzerland", "Syria", "Taiwan", "Tajikistan", "Tanzania", "Thailand", "The Bahamas",
  "The Gambia", "Togo", "Tokelau", "Tonga", "Trinidad and Tobago", "Tunisia", "Turkey",
  "Turkmenistan", "Turks and Caicos Islands", "Tuvalu", "Virgin Islands", "Uganda",
  "Ukraine", "United Arab Emirates", "United Kingdom",
  "United States", "United States Minor Outlying Islands", "Uruguay", "Uzbekistan",
  "Vanuatu", "Vatican City", "Venezuela", "Vietnam", "Wallis and Futuna", "Western Sahara",
  "Yemen", "Yugoslavia", "Zambia", "Zimbabwe"
};
```

This is the list of suggestions that will be provided in a drop-down list when the user types into the `AutoCompleteTextView` widget.

Run the application. As you type, you should see something like this:



More Information

Note that using a hard-coded string array is not a recommended design practice because your application code should focus on behavior, not content. Application content such as strings should be externalized from the code to make modifications to the content easier and facilitate localization of the content. The hard-coded strings are used in this tutorial only to make it simple and focus on the `AutoCompleteTextView` widget. Instead, your application should declare such string arrays in an XML file. This can be done with a `<string-array>` resource in your project `res/values/strings.xml` file. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="countries_array">
        <item>Bahrain</item>
        <item>Bangladesh</item>
        <item>Barbados</item>
        <item>Belarus</item>
        <item>Belgium</item>
        <item>Belize</item>
        <item>Benin</item>
    </string-array>
</resources>
```

To use these resource strings for the `ArrayAdapter`, replace the original `ArrayAdapter` constructor line with the following:

```
string[] countries = Resources.GetStringArray (Resource.array.countries_array);
var adapter = new ArrayAdapter<String> (this, Resource.layout.list_item, countries);
```

References

- AutoCompleteTextView Recipe – Xamarin.Android sample project for the `AutoCompleteTextView`
- `ArrayAdapter`
- `AutoCompleteTextView`

Buttons in Xamarin.Android

10/28/2019 • 2 minutes to read • Edit Online

The `Button` class is used to represent various different styles of button in Android screens. This section introduces the different options for working with buttons in Xamarin.Android:

- RadioButton allows the user to select one option from a set.
- ToggleButton allow the user to flip (toggle) a setting between two states.
- CheckBox is a special type of button that can be either checked or unchecked to indicate one of two possible states.
- You can also create a custom button that uses an image instead of text.

RadioButton

10/29/2019 • 2 minutes to read • Edit Online

In this section, you will create two mutually-exclusive radio buttons (enabling one disables the other), using the `RadioGroup` and `RadioButton` widgets. When either radio button is pressed, a toast message will be displayed.

Open the **Resources/layout/Main.axml** file and add two `RadioButton`s, nested in a `RadioGroup` (inside the `LinearLayout`):

```
<RadioGroup
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:orientation="vertical">
  <RadioButton android:id="@+id/radio_red"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Red" />
  <RadioButton android:id="@+id/radio_blue"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Blue" />
</RadioGroup>
```

It's important that the `RadioButton`s are grouped together by the `RadioGroup` element so that no more than one can be selected at a time. This logic is automatically handled by the Android system. When one `RadioButton` within a group is selected, all others are automatically deselected.

To do something when each `RadioButton` is selected, we need to write an event handler:

```
private void RadioButtonClick (object sender, EventArgs e)
{
    RadioButton rb = (RadioButton)sender;
    Toast.MakeText (this, rb.Text, ToastLength.Short).Show ();
}
```

First, the sender that is passed in is cast into a RadioButton. Then a `Toast` message displays the selected radio button's text.

Now, at the bottom of the `OnCreate()` method, add the following:

```
RadioButton radio_red = FindViewById<RadioButton>(Resource.Id.radio_red);
RadioButton radio_blue = FindViewById<RadioButton>(Resource.Id.radio_blue);

radio_red.Click += RadioButtonClick;
radio_blue.Click += RadioButtonClick;
```

This captures each of the `RadioButton`s from the layout and adds the newly-created event handlerto each.

Run the application.

TIP

If you need to change the state yourself (such as when loading a saved `CheckBoxPreference`), use the `Checked` property setter or `Toggle()` method.

ToggleButton

7/8/2021 • 2 minutes to read • Edit Online

In this section, you'll create a button used specifically for toggling between two states, using the `ToggleButton` widget. This widget is an excellent alternative to radio buttons if you have two simple states that are mutually exclusive ("on" and "off", for example). Android 4.0 (API level 14) introduced an alternative to the toggle button known as a `Switch`.

An example of a **ToggleButton** can be seen in the left hand pair of images, while the right hand pair of images presents an example of a **Switch**:

Which control an application uses is a matter of style. Both widgets are functionally equivalent.

Open the **Resources/layout/Main.axml** file and add the `ToggleButton` element (inside the `LinearLayout`):

To do something when the state is changed, add the following code to the end of the `OnCreate()` method:

```
ToggleButton togglebutton = FindViewById<ToggleButton>(Resource.Id.togglebutton);

togglebutton.Click += (o, e) => {
    // Perform action on clicks
    if (togglebutton.Checked)
        Toast.MakeText(this, "Checked", ToastLength.Short).Show ();
    else
        Toast.MakeText(this, "Not checked", ToastLength.Short).Show ();
};
```

This captures the `ToggleButton` element from the layout, and handles the Click event, which defines the action to perform when the button is clicked. In this example, the method checks the new state of the button, then shows a `Toast` message that indicates the current state.

Notice that the `ToggleButton` handles its own state change between checked and unchecked, so you just ask which it is.

Run the application.

> **TIP**
>
> If you need to change the state yourself (such as when loading a saved `CheckBoxPreference`), use the `Checked` property setter or `Toggle()` method.

Related Links

- ToggleButton
- Switch

CheckBox

10/29/2019 • 2 minutes to read • Edit Online

In this section, you will create a checkbox for selecting items, using the `CheckBox` widget. When the checkbox is pressed, a toast message will indicate the current state of the checkbox.

Open the **Resources/layout/Main.axml** file and add the `CheckBox` element (inside the `LinearLayout`):

```
<CheckBox android:id="@+id/checkbox"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="check it out" />
```

To do something when the state is changed, add the following code to the end of the `OnCreate()` method:

```
CheckBox checkbox = FindViewById<CheckBox>(Resource.Id.checkbox);

checkbox.Click += (o, e) => {
    if (checkbox.Checked)
        Toast.MakeText (this, "Selected", ToastLength.Short).Show ();
    else
        Toast.MakeText (this, "Not selected", ToastLength.Short).Show ();
};
```

This captures the `CheckBox` element from the layout, then handles the Click event, which defines the action to be made when the checkbox is clicked. When clicked, the `Checked` property is called to check the new state of the check box. If it has been checked, then a `Toast` displays the message "Selected", otherwise it displays "Not selected". The `CheckBox` handles its own state changes, so you only need to query the current state.

Run it.

TIP

If you need to change the state yourself (such as when loading a saved `CheckBoxPreference`, use the `Checked` property setter or `Toggle()` method.

Custom Button

7/8/2021 • 2 minutes to read • Edit Online

In this section, you will create a button with a custom image instead of text, using the `Button` widget and an XML file that defines three different images to use for the different button states. When the button is pressed, a short message will be displayed.

Right-click and download the three images below, then copy them to the **Resources/drawable** directory of your project. These will be used for the different button states.

Create a new file in the **Resources/drawable** directory named **android_button.xml**. Insert the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@drawable/android_pressed"
          android:state_pressed="true" />
    <item android:drawable="@drawable/android_focused"
          android:state_focused="true" />
    <item android:drawable="@drawable/android_normal" />
</selector>
```

This defines a single drawable resource, which will change its image based on the current state of the button. The first `<item>` defines **android_pressed.png** as the image when the button is pressed (it's been activated); the second `<item>` defines **android_focused.png** as the image when the button is focused (when the button is highlighted using the trackball or directional pad); and the third `<item>` defines **android_normal.png** as the image for the normal state (when neither pressed nor focused). This XML file now represents a single drawable resource and when referenced by a `Button` for its background, the image displayed will change based on these three states.

> **NOTE**
>
> The order of the `<item>` elements is important. When this drawable is referenced, the `<item>`s are traversed in-order to determine which one is appropriate for the current button state. Because the "normal" image is last, it is only applied when the conditions `android:state_pressed` and `android:state_focused` have both evaluated false.

Open the **Resources/layout/Main.axml** file and add the `Button` element:

```
<Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="10dp"
        android:background="@drawable/android_button" />
```

The `android:background` attribute specifies the drawable resource to use for the button background (which, when saved at **Resources/drawable/android.xml**, is referenced as `@drawable/android`). This replaces the normal background image used for buttons throughout the system. In order for the drawable to change its image based on the button state, the image must be applied to the background.

To make the button do something when pressed, add the following code at the end of the `OnCreate()` method:

```
Button button = FindViewById<Button>(Resource.Id.button);

button.Click += (o, e) => {
    Toast.MakeText (this, "Beep Boop", ToastLength.Short).Show ();
};
```

This captures the `Button` from the layout, then adds a `Toast` message to be displayed when the `Button` is clicked.

Now run the application.

Xamarin.Android Calendar

7/8/2021 • 7 minutes to read • Edit Online

Calendar API

A new set of calendar APIs introduced in Android 4 supports applications that are designed to read or write data to the calendar provider. These APIs support a wealth of interaction options with calendar data, including the ability to read and write events, attendees, and reminders. By using the calendar provider in your application, data you add through the API will appear in the built-in calendar app that comes with Android 4.

Adding Permissions

When working with the new calendar APIs in your application, the first thing you need to do is add the appropriate permissions to the Android manifest. The permissions you need to add are `android.permisson.READ_CALENDAR` and `android.permission.WRITE_CALENDAR`, depending on whether you are reading and/or writing calendar data.

Using the Calendar Contract

Once you set the permissions, you can interact with calendar data by using the `CalendarContract` class. This class provides a data model that applications can use when they interact with the calendar provider. The `CalendarContract` allows applications to resolve the Uris to calendar entities, such as calendars and events. It also provides a way to interact with various fields in each entity, such as a calendar's name and ID, or an event's start and end date.

Let's look at an example that uses the Calendar API. In this example, we'll examine how to enumerate calendars and their events, as well as how to add a new event to a calendar.

Listing Calendars

First, let's examine how to enumerate the calendars that have been registered in the calendar app. To do this, we can instantiate a `CursorLoader`. Introduced in Android 3.0 (API 11), `CursorLoader` is the preferred way to consume a `ContentProvider`. At a minimum, we'll need to specify the content Uri for calendars and the columns we want to return; this column specification is known as a *projection*.

Calling the `CursorLoader.LoadInBackground` method allows us to query a content provider for data, such as the calendar provider. `LoadInBackground` performs the actual load operation and returns a `Cursor` with the results of the query.

The `CalendarContract` assists us in specifying both the content `Uri` and the projection. To get the content `Uri` for querying calendars, we can simply use the `CalendarContract.Calendars.ContentUri` property like this:

```
var calendarsUri = CalendarContract.Calendars.ContentUri;
```

Using the `CalendarContract` to specify which calendar columns we want is equally simple. We just add fields in the `CalendarContract.Calendars.InterfaceConsts` class to an array. For example, the following code includes the calendar's ID, display name, and account name:

```
string[] calendarsProjection = {
    CalendarContract.Calendars.InterfaceConsts.Id,
    CalendarContract.Calendars.InterfaceConsts.CalendarDisplayName,
    CalendarContract.Calendars.InterfaceConsts.AccountName
};
```

The `Id` is important to include if you are using a `SimpleCursorAdapter` to bind the data to the UI, as we will see shortly. With the content Uri and projection in place, we instantiate the `CursorLoader` and call the `CursorLoader.LoadInBackground` method to return a cursor with the calendar data as shown below:

```
var loader = new CursorLoader(this, calendarsUri, calendarsProjection, null, null, null);
var cursor = (ICursor)loader.LoadInBackground();
```

The UI for this example contains a `ListView`, with each item in the list representing a single calendar. The following XML shows the markup that includes the `ListView`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="fill_parent">
  <ListView
    android:id="@android:id/android:list"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" />
</LinearLayout>
```

Also, we need to specify the UI for each list item, which we place in a separate XML file as follows:

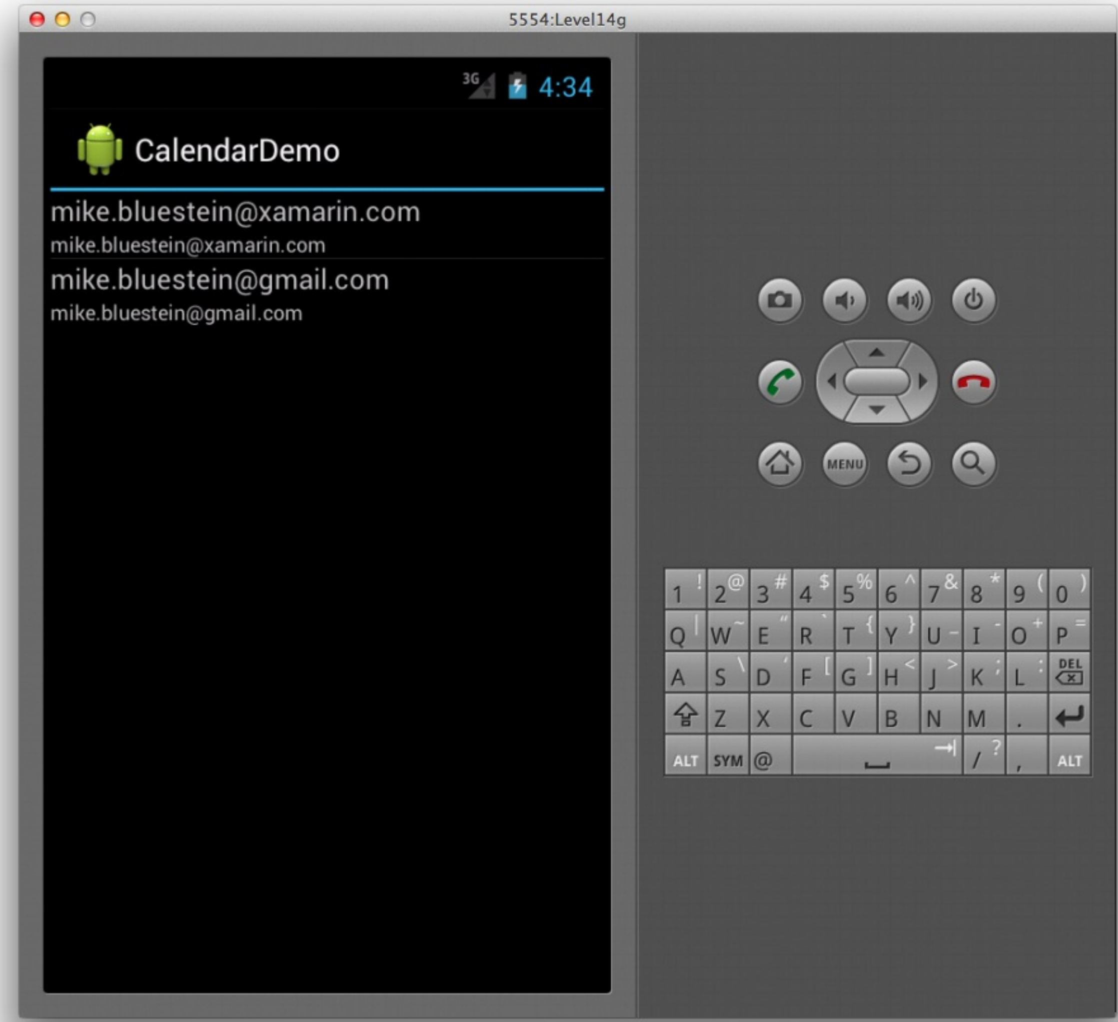
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="wrap_content">
  <TextView android:id="@+id/calDisplayName"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="16dip" />
  <TextView android:id="@+id/calAccountName"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="12dip" />
</LinearLayout>
```

From this point on, it's just normal Android code to bind the data from the cursor to the UI. We'll use a `SimpleCursorAdapter` as follows:

```
string[] sourceColumns = {
    CalendarContract.Calendars.InterfaceConsts.CalendarDisplayName,
    CalendarContract.Calendars.InterfaceConsts.AccountName };

int[] targetResources = {
    Resource.Id.calDisplayName, Resource.Id.calAccountName };

SimpleCursorAdapter adapter = new SimpleCursorAdapter (this,
    Resource.Layout.CalListItem, cursor, sourceColumns, targetResources);

ListAdapter = adapter;
```

In the above code, the adapter takes the columns specified in the `sourceColumns` array and writes them to the user interface elements in the `targetResources` array for each calendar entry in the cursor. The Activity used here is a subclass of `ListActivity`; it includes the `ListAdapter` property to which we set the adapter.

Here's a screenshot showing the end result, with the calendar info displayed in the `ListView`:



Listing Calendar Events

Next let's look at how to enumerate the events for a given calendar. Building upon the example above, we'll present a list of events when the user selects one of the calendars. Therefore, we'll need to handle the item selection in the previous code:

```
ListView.ItemClick += (sender, e) => {
    int i = (e as ItemEventArgs).Position;

    cursor.MoveToPosition(i);
    int calId =
        cursor.GetInt (cursor.GetColumnIndex (calendarsProjection [0]));

    var showEvents = new Intent(this, typeof(EventListActivity));
    showEvents.PutExtra("calId", calId);
    StartActivity(showEvents);
};
```

In this code, we're creating an Intent to open an Activity of type `EventListActivity`, passing the calendar's ID in the Intent. We will need the ID to know which calendar to query for events. In the `EventListActivity`'s `OnCreate` method, we can retrieve the ID from the `Intent` as shown below:

```
_calId = Intent.GetIntExtra ("calId", -1);
```

Now let's query events for this calendar ID. The process to query for events is similar to the way we queried for a list of calendars earlier, only this time we'll work with the `CalendarContract.Events` class. The following code creates a query to retrieve events:

```
var eventsUri = CalendarContract.Events.ContentUri;

string[] eventsProjection = {
    CalendarContract.Events.InterfaceConsts.Id,
    CalendarContract.Events.InterfaceConsts.Title,
    CalendarContract.Events.InterfaceConsts.Dtstart
};

var loader = new CursorLoader(this, eventsUri, eventsProjection,
                   String.Format ("calendar_id={0}", _calId), null, "dtstart ASC");
var cursor = (ICursor)loader.LoadInBackground();
```

In this code, we first get the content `Uri` for events from the `CalendarContract.Events.ContentUri` property. Then we specify the event columns we want to retrieve in the eventsProjection array. Finally, we instantiate a `CursorLoader` with this information and call the loader's `LoadInBackground` method to return a `Cursor` with the event data.

To display the event data in the UI, we can use markup and code just like we did before to display the list of calendars. Again, we use `SimpleCursorAdapter` to bind the data to a `ListView` as shown in the following code:

```
string[] sourceColumns = {
    CalendarContract.Events.InterfaceConsts.Title,
    CalendarContract.Events.InterfaceConsts.Dtstart };

int[] targetResources = {
    Resource.Id.eventTitle,
    Resource.Id.eventStartDate };

var adapter = new SimpleCursorAdapter (this, Resource.Layout.EventListItem,
    cursor, sourceColumns, targetResources);

adapter.ViewBinder = new ViewBinder ();
ListAdapter = adapter;
```

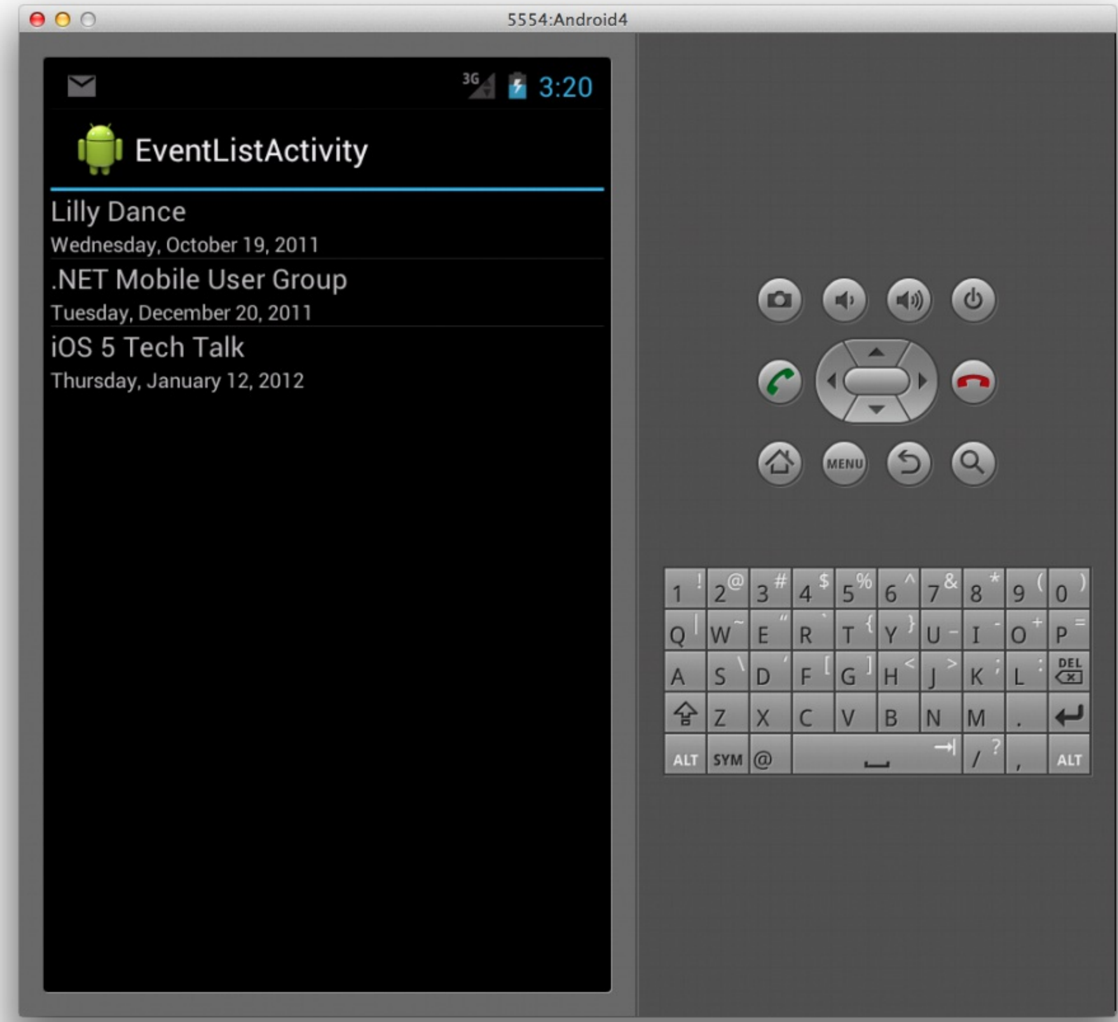
The main difference between this code and the code that we used earlier to show the calendar list is the use of a `ViewBinder`, which is set on the line:

```
adapter.ViewBinder = new ViewBinder ();
```

The `ViewBinder` class allows us to further control how we bind values to views. In this case, we use it to convert the event start time from milliseconds to a date string, as shown in the following implementation:

```
class ViewBinder : Java.Lang.Object, SimpleCursorAdapter.IViewBinder
{
    public bool SetViewValue (View view, Android.Database.ICursor cursor,
        int columnIndex)
    {
        if (columnIndex == 2) {
            long ms = cursor.GetLong (columnIndex);

            DateTime date = new DateTime (1970, 1, 1, 0, 0, 0,
                DateTimeKind.Utc).AddMilliseconds (ms).ToLocalTime ();

            TextView textView = (TextView)view;
            textView.Text = date.ToLongDateString ();

            return true;
        }
        return false;
    }
}
```

This displays a list of events as shown below:



Adding a Calendar Event

We've seen how to read calendar data. Now let's see how to add an event to a calendar. For this to work, be sure to include the `android.permission.WRITE_CALENDAR` permission we mentioned earlier. To add an event to a calendar, we will:

1. Create a `ContentValues` instance.
2. Use keys from the `CalendarContract.Events.InterfaceConsts` class to populate the `ContentValues` instance.
3. Set the time zones for the event start and end times.
4. Use a `ContentResolver` to insert the event data into the calendar.

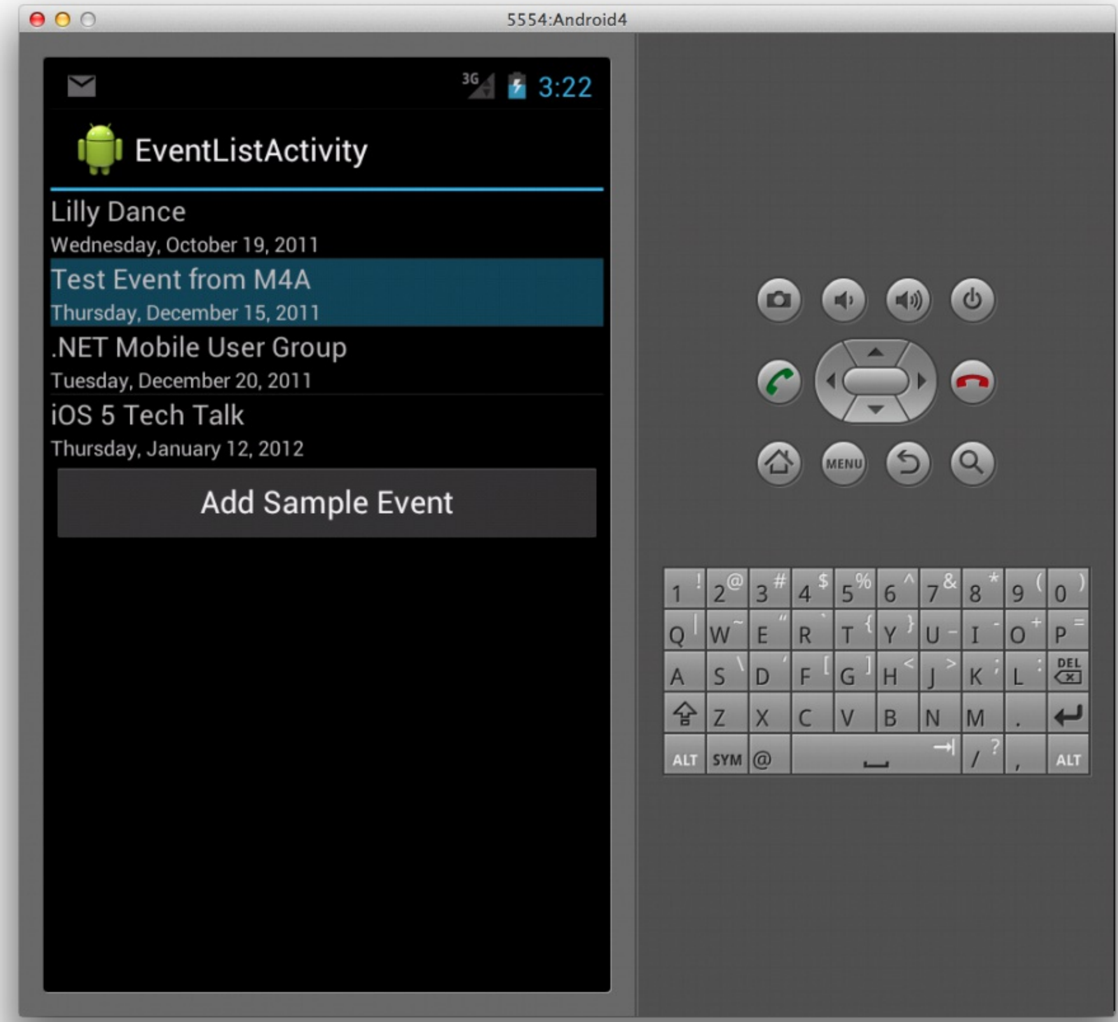
The code below illustrates these steps:

```
ContentValues eventValues = new ContentValues ();

eventValues.Put (CalendarContract.Events.InterfaceConsts.CalendarId,
    _calId);
eventValues.Put (CalendarContract.Events.InterfaceConsts.Title,
    "Test Event from M4A");
eventValues.Put (CalendarContract.Events.InterfaceConsts.Description,
    "This is an event created from Xamarin.Android");
eventValues.Put (CalendarContract.Events.InterfaceConsts.Dtstart,
    GetDateTimeMS (2011, 12, 15, 10, 0));
eventValues.Put (CalendarContract.Events.InterfaceConsts.Dtend,
    GetDateTimeMS (2011, 12, 15, 11, 0));

eventValues.Put(CalendarContract.Events.InterfaceConsts.EventTimezone,
    "UTC");
eventValues.Put(CalendarContract.Events.InterfaceConsts.EventEndTimezone,
    "UTC");

var uri = ContentResolver.Insert (CalendarContract.Events.ContentUri,
    eventValues);
```

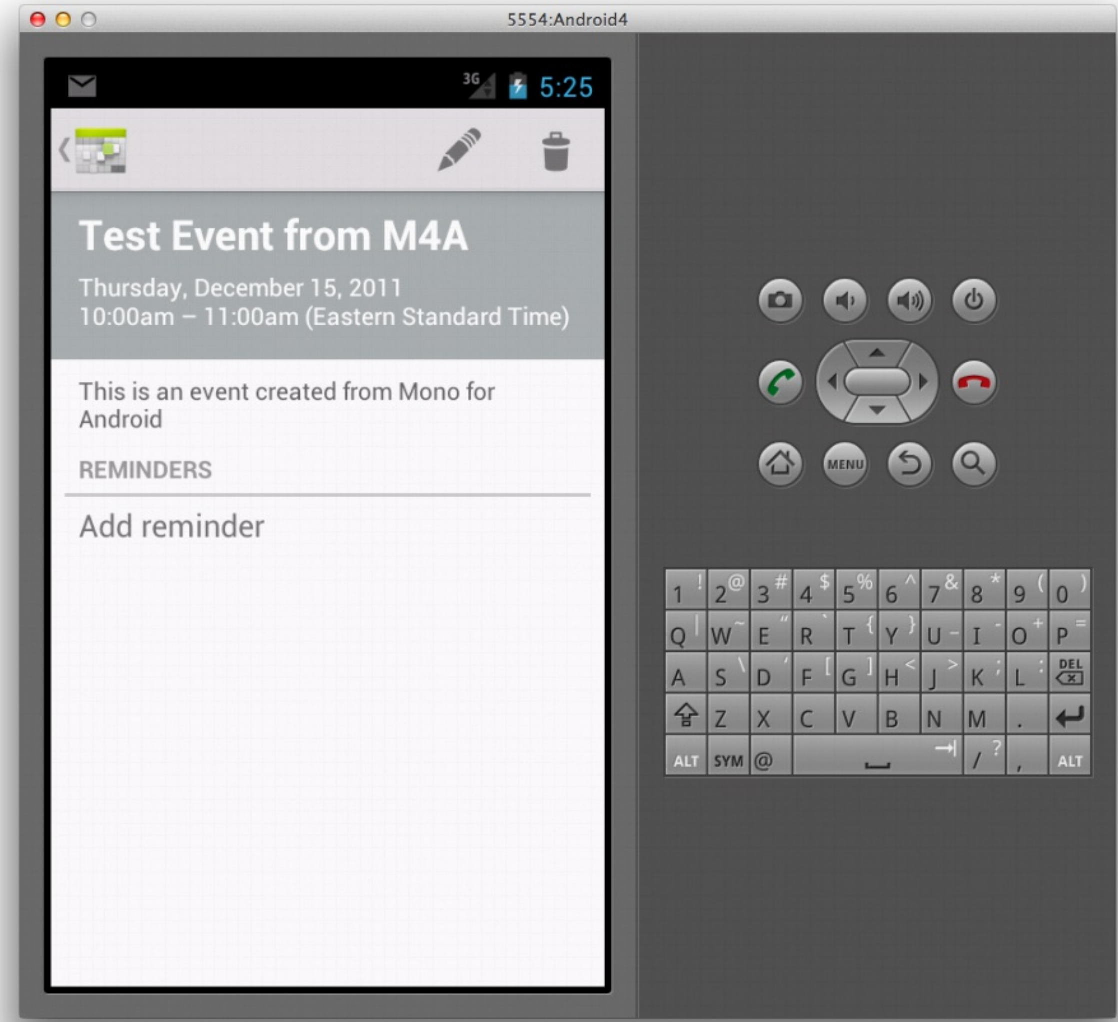
Note that if we do not set the time zone, an exception of type `Java.Lang.IllegalArgumentException` will be thrown. Because event time values must be expressed in milliseconds since epoch, we create a `GetDateTimeMS` method (in `EventListActivity`) to convert our date specifications into millisecond format:

```
long GetDateTimeMS (int yr, int month, int day, int hr, int min)
{
    Calendar c = Calendar.GetInstance (Java.Util.TimeZone.Default);

    c.Set (Java.Util.CalendarField.DayOfMonth, 15);
    c.Set (Java.Util.CalendarField.HourOfDay, hr);
    c.Set (Java.Util.CalendarField.Minute, min);
    c.Set (Java.Util.CalendarField.Month, Calendar.December);
    c.Set (Java.Util.CalendarField.Year, 2011);

    return c.TimeInMillis;
}
```

If we add a button to the event list UI and run the above code in the button's click event handler, the event is added to the calendar and updated in our list as shown below:



If we open the calendar app, then we will see that the event is written there as well:



As you can see, Android allows powerful and easy access to retrieve and persist calendar data, allowing applications to seamlessly integrate calendar capabilities.

Related Links

- Calendar Demo (sample)

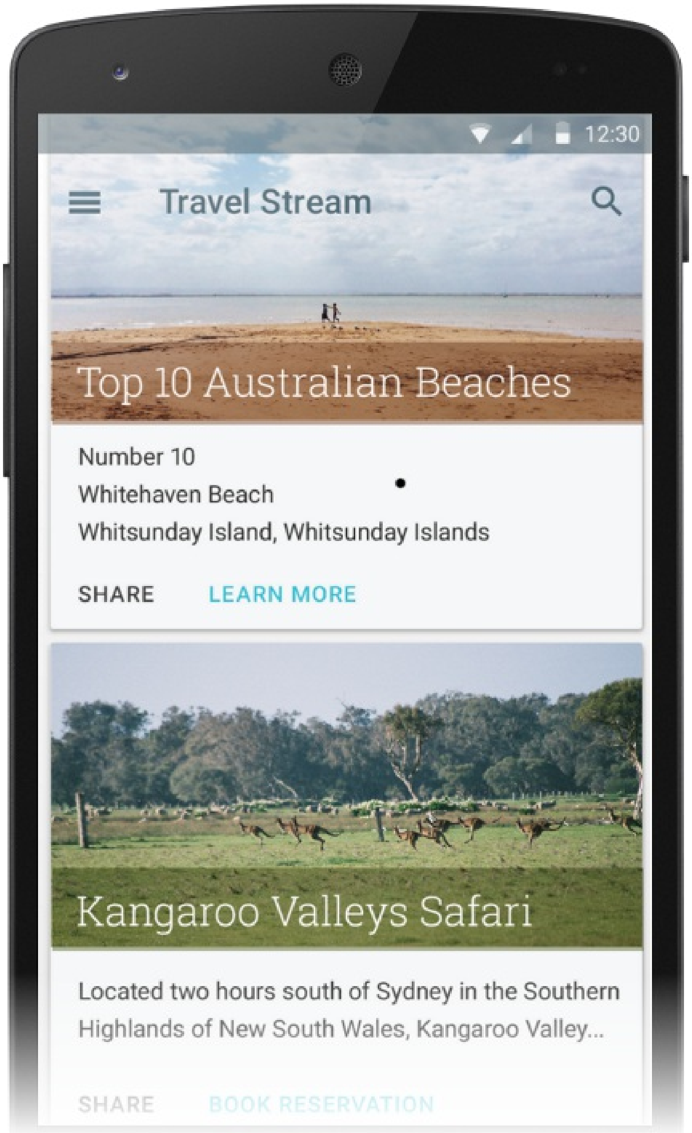
Xamarin.Android CardView

7/8/2021 • 8 minutes to read • Edit Online

The Cardview widget is a UI component that presents text and image content in views that resemble cards. This guide explains how to use and customize CardView in Xamarin.Android applications while maintaining backward compatibility with earlier versions of Android.

Overview

The `Cardview` widget, introduced in Android 5.0 (Lollipop), is a UI component that presents text and image content in views that resemble cards. `CardView` is implemented as a `FrameLayout` widget with rounded corners and a shadow. Typically, a `CardView` is used to present a single row item in a `ListView` or `GridView` view group. For example, the following screen shot is an example of a travel reservation app that implements `CardView` -based travel destination cards in a scrollable `ListView` :



This guide explains how to add the `CardView` package to your Xamarin.Android project, how to add `CardView` to your layout, and how to customize the appearance of `CardView` in your app. In addition, this guide provides a detailed list of `CardView` attributes that you can change, including attributes to help you use `CardView` on versions of Android earlier than Android 5.0 Lollipop.

Requirements

The following is required to use new Android 5.0 and later features (including `CardView`) in Xamarin-based apps:

- **Xamarin.Android** – Xamarin.Android 4.20 or later must be installed and configured with either Visual Studio or Visual Studio for Mac.
- **Android SDK** – Android 5.0 (API 21) or later must be installed via the Android SDK Manager.
- **Java JDK 1.8** – JDK 1.7 can be used if you are specifically targeting API level 23 and earlier. JDK 1.8 is available from Oracle.

Your app must also include the `Xamarin.Android.Support.v7.CardView` package. To add the `Xamarin.Android.Support.v7.CardView` package in Visual Studio for Mac:

1. Open your project, right-click **Packages** and select **Add Packages**.
2. In the **Add Packages** dialog, search for **CardView**.
3. Select **Xamarin Support Library v7 CardView**, then click **Add Package**.

To add the `Xamarin.Android.Support.v7.CardView` package in Visual Studio:

1. Open your project, right-click the **References** node (in the **Solution Explorer** pane) and select **Manage NuGet Packages...**.
2. When the **Manage NuGet Packages** dialog is displayed, enter **CardView** in the search box.
3. When **Xamarin Support Library v7 CardView** appears, click **Install**.

To learn how to configure an Android 5.0 app project, see Setting Up an Android 5.0 Project. For more information about installing NuGet packages, see Walkthrough: Including a NuGet in your project.

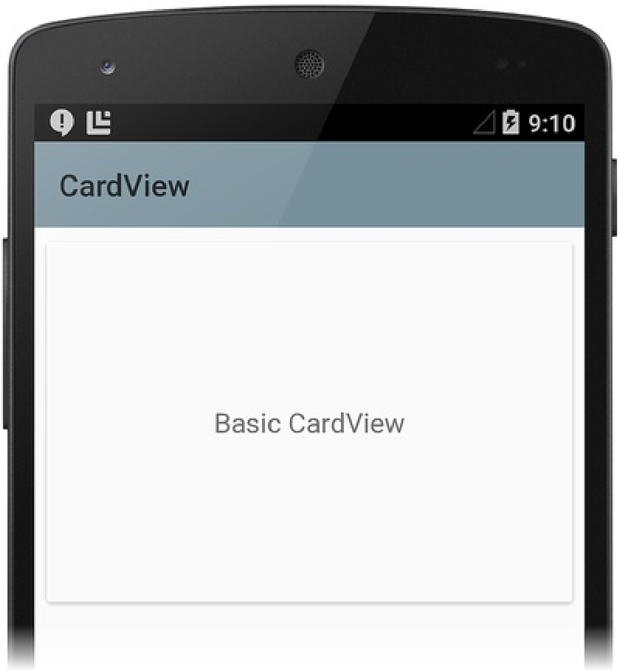
Introducing CardView

The default `CardView` resembles a white card with minimally rounded corners and a slight shadow. The following example **Main.axml** layout displays a single `CardView` widget that contains a `TextView`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:gravity="center_horizontal"
    android:padding="5dp">
    <android.support.v7.widget.CardView
        android:layout_width="fill_parent"
        android:layout_height="245dp"
        android:layout_gravity="center_horizontal">
        <TextView
            android:text="Basic CardView"
            android:layout_marginTop="0dp"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:gravity="center"
            android:layout_centerVertical="true"
            android:layout_alignParentRight="true"
            android:layout_alignParentEnd="true" />
    </android.support.v7.widget.CardView>
</LinearLayout>
```

If you use this XML to replace the existing contents of **Main.axml**, be sure to comment out any code in **MainActivity.cs** that refers to resources in the previous XML.

This layout example creates a default `CardView` with a single line of text as shown in the following screen shot:



In this example, the app style is set to the light Material Theme (`Theme.Material.Light`) so that the `CardView` shadows and edges are easier to see. For more information about theming Android 5.0 apps, see Material Theme. In the next section, we'll learn how to customize `CardView` for an application.

Customizing CardView

You can modify the basic `CardView` attributes to customize the appearance of the `CardView` in your app. For example, the elevation of the `CardView` can be increased to cast a larger shadow (which makes the card seem to float higher above the background). Also, the corner radius can be increased to make the corners of the card more rounded.

In the next layout example, a customized `CardView` is used to create a simulation of a print photograph (a "snapshot"). An `ImageView` is added to the `CardView` for displaying the image, and a `TextView` is positioned below the `ImageView` for displaying the title of the image. In this example layout, the `CardView` has the following customizations:

- The `cardElevation` is increased to 4dp to cast a larger shadow.
- The `cardCornerRadius` is increased to 5dp to make the corners appear more rounded.

Because `CardView` is provided by the Android v7 support library, its attributes are not available from the `android:` namespace. Therefore, you must define your own XML namespace and use that namespace as the `CardView` attribute prefix. In the layout example below, we will use this line to define a namespace called `cardview`:

```
xmlns:cardview="http://schemas.android.com/apk/res-auto"
```

You can call this namespace `card_view` or even `myapp` if you choose (it's accessible only within the scope of this file). Whatever you choose to call this namespace, you must use it to prefix the `CardView` attribute that you want to modify. In this layout example, the `cardview` namespace is the prefix for `cardElevation` and `cardCornerRadius`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:cardview="http://schemas.android.com/apk/res-auto"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:gravity="center_horizontal"
    android:padding="5dp">
    <android.support.v7.widget.CardView
        android:layout_width="fill_parent"
        android:layout_height="245dp"
        android:layout_gravity="center_horizontal"
        cardview:cardElevation="4dp"
        cardview:cardCornerRadius="5dp">
        <LinearLayout
            android:layout_width="fill_parent"
            android:layout_height="240dp"
            android:orientation="vertical"
            android:padding="8dp">
            <ImageView
                android:layout_width="fill_parent"
                android:layout_height="190dp"
                android:id="@+id/imageView"
                android:scaleType="centerCrop" />
            <TextView
                android:layout_width="fill_parent"
                android:layout_height="wrap_content"
                android:textAppearance="?android:attr/textAppearanceMedium"
                android:textColor="#333333"
                android:text="Photo Title"
                android:id="@+id/textView"
                android:layout_gravity="center_horizontal"
                android:layout_marginLeft="5dp" />
        </LinearLayout>
    </android.support.v7.widget.CardView>
</LinearLayout>
```

When this layout example is used to display an image in a photo viewing app, the `CardView` has the appearance of a photo snapshot, as depicted in the following screenshot:

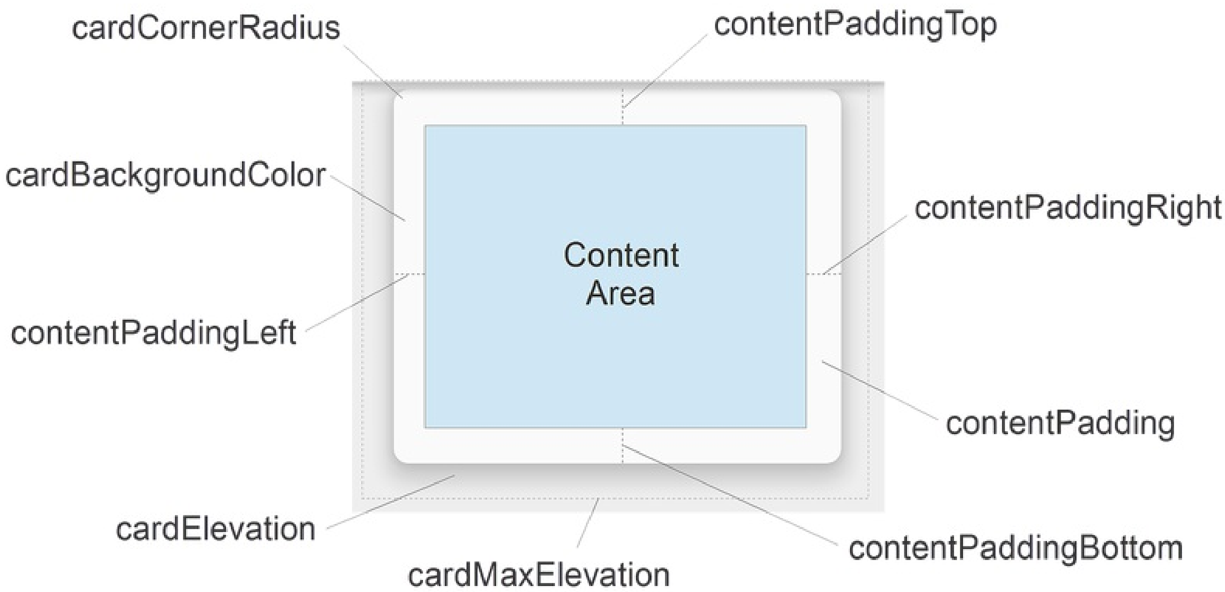
This screenshot is taken from the RecyclerViewer sample app, which uses a `RecyclerView` widget to present a scrolling list of `CardView` images for viewing photos. For more information about `RecyclerView`, see the

RecyclerView guide.

Notice that a `CardView` can display more than one child view in its content area. For example, in the above photo viewing app example, the content area is comprised of a `ListView` that contains an `ImageView` and a `TextView`. Although `CardView` instances are often arranged vertically, you can also arrange them horizontally (see Creating a Custom View Style for an example screenshot).

CardView Layout Options

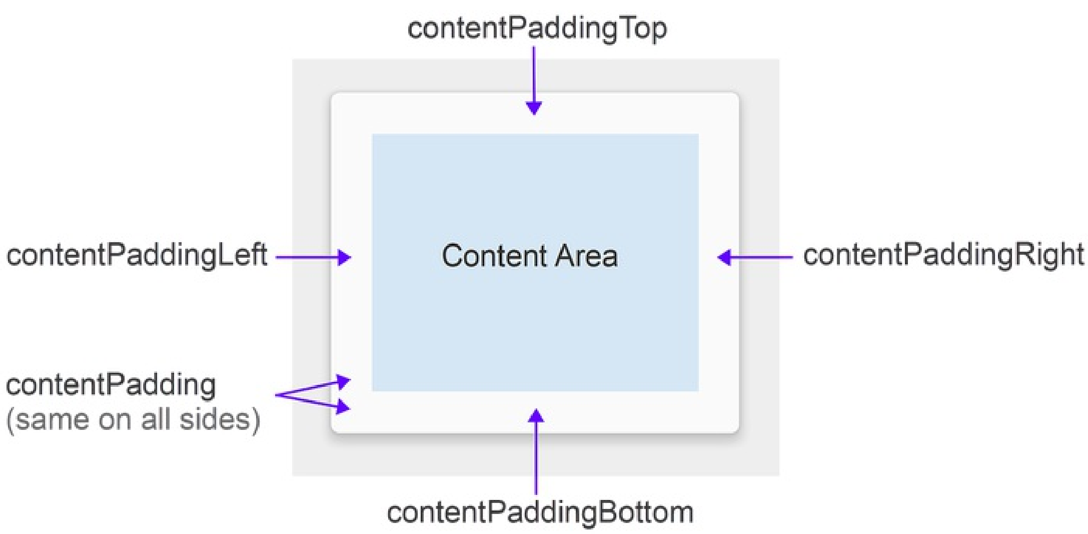
`CardView` layouts can be customized by setting one or more attributes that affect its padding, elevation, corner radius, and background color:



Each attribute can also be changed dynamically by calling a counterpart `CardView` method (for more information on `CardView` methods, see the CardView class reference). Note that these attributes (except for background color) accept a dimension value, which is a decimal number followed by the unit. For example, `11.5dp` specifies 11.5 density-independent pixels.

Padding

`CardView` offers five padding attributes to position content within the card. You can set them in your layout XML or you can call analogous methods in your code:



The padding attributes are explained as follows:

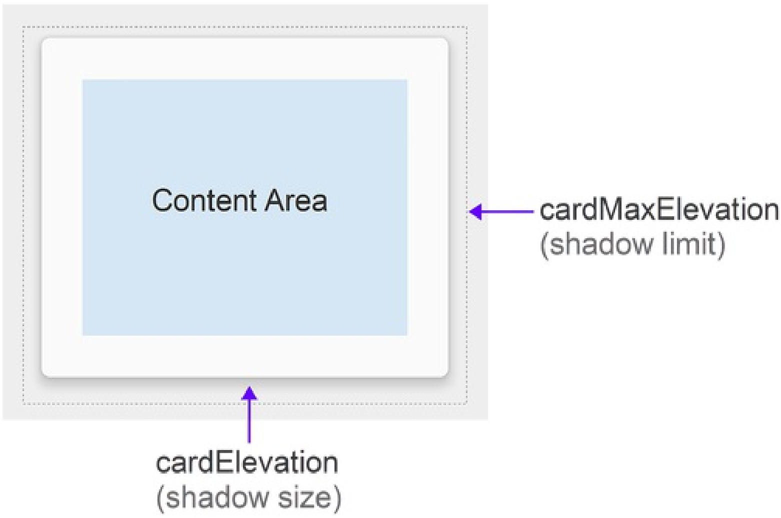
- `contentPadding` – Inner padding between the child views of the `CardView` and all edges of the card.

- `contentPaddingBottom` – Inner padding between the child views of the `CardView` and the bottom edge of the card.
- `contentPaddingLeft` – Inner padding between the child views of the `CardView` and the left edge of the card.
- `contentPaddingRight` – Inner padding between the child views of the `CardView` and the right edge of the card.
- `contentPaddingTop` – Inner padding between the child views of the `CardView` and the top edge of the card.

Content padding attributes are relative to the boundary of the content area rather than to any given widget located within the content area. For example, if `contentPadding` were sufficiently increased in the photo viewing app, the `CardView` would crop both the image and the text shown on the card.

Elevation

`CardView` offers two elevation attributes to control its elevation and, as a result, the size of its shadow:



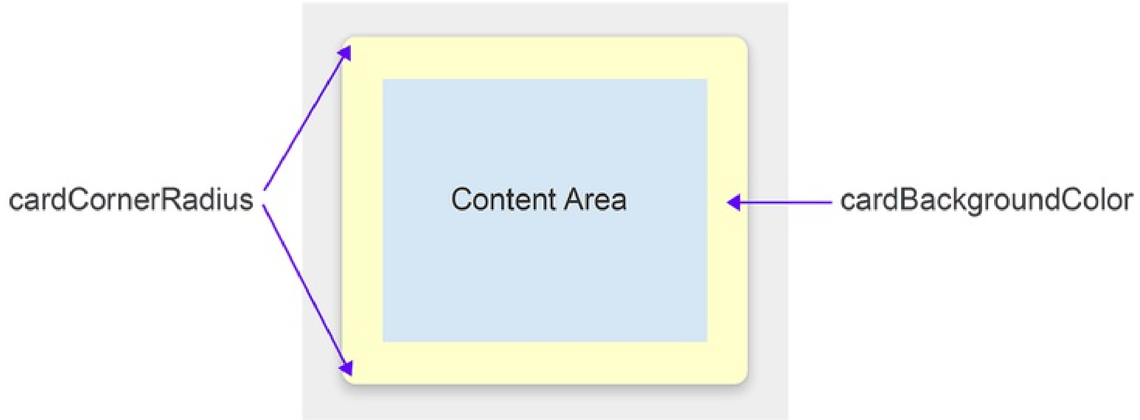
The elevation attributes are explained as follows:

- `cardElevation` – The elevation of the `CardView` (represents its Z axis).
- `cardMaxElevation` – The maximum value of the `CardView` 's elevation.

Larger values of `cardElevation` increase the shadow size to make `CardView` seem to float higher above the background. The `cardElevation` attribute also determines the drawing order of overlapping views; that is, the `CardView` will be drawn under another overlapping view with a higher elevation setting and above any overlapping views with a lower elevation setting. The `cardMaxElevation` setting is useful for when your app changes elevation dynamically – it prevents the shadow from extending past the limit that you define with this setting.

Corner Radius and Background Color

`CardView` offers attributes that you can use to control its corner radius and its background color. These two properties allow you change the overall style of the `CardView` :



These attributes are explained as follows:

- `cardCornerRadius` – The corner radius of all corners of the `CardView`.
- `cardBackgroundColor` – The background color of the `CardView`.

In this diagram, `cardCornerRadius` is set to a more rounded 10dp and `cardBackgroundColor` is set to `"#FFFFCC"` (light yellow).

Compatibility

You can use `CardView` on versions of Android earlier than Android 5.0 Lollipop. Because `CardView` is part of the Android v7 support library, you can use `CardView` with Android 2.1 (API level 7) and higher. However, you must install the `Xamarin.Android.Support.v7.CardView` package as described in Requirements, above.

`CardView` exhibits slightly different behavior on devices before Lollipop (API level 21):

- `CardView` uses a programmatic shadow implementation that adds additional padding.
- `CardView` does not clip child views that intersect with the `CardView`'s rounded corners.

To help in managing these compatibility differences, `CardView` provides several additional attributes that you can configure in your layout:

- `cardPreventCornerOverlap` – Set this attribute to `true` to add padding when your app is running on earlier Android versions (API level 20 and earlier). This setting prevents `CardView` content from intersecting with the `CardView`'s rounded corners.
- `cardUseCompatPadding` – Set this attribute to `true` to add padding when your app is running in versions of Android at or greater than API level 21. If you want to use `CardView` on pre-Lollipop devices and have it look the same on Lollipop (or later), set this attribute to `true`. When this attribute is enabled, `CardView` adds additional padding to draw shadows when it runs on pre-Lollipop devices. This helps to overcome the differences in padding that are introduced when pre-Lollipop programmatic shadow implementations are in effect.

For more information about maintaining compatibility with earlier versions of Android, see Maintaining Compatibility.

Summary

This guide introduced the new `CardView` widget included in Android 5.0 (Lollipop). It demonstrated the default `CardView` appearance and explained how to customize `CardView` by changing its elevation, corner roundness, content padding, and background color. It listed the `CardView` layout attributes (with reference diagrams), and explained how to use `CardView` on Android devices earlier than Android 5.0 Lollipop. For more information about `CardView`, see the CardView class reference.

Related Links

- RecyclerView (sample)
- Introduction to Lollipop
- CardView class reference

Xamarin.Android Edit Text

7/8/2021 • 2 minutes to read • Edit Online

In this section, you will use the EditText widget to create a text field for user input. Once text has been entered into the field, the **Enter** key will display the text in a toast message.

Open **Resources/layout/activity_main.axml** and add the EditText element to a containing layout. The following example **activity_main.axml** has an `EditText` that has been added to a `LinearLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <EditText
        android:id="@+id/edittext"
        android:layout_width="match_parent"
        android:imeOptions="actionGo"
        android:inputType="text"
        android:layout_height="wrap_content" />
</LinearLayout>
```

In this code example, the `EditText` attribute `android:imeOptions` is set to `actionGo`. This setting changes the default Done action to the Go action so that tapping the **Enter** key triggers the `KeyPress` input handler. (Typically, `actionGo` is used so that the **Enter** key takes the user to the target of a URL that is typed in.)

To handle user text input, add the following code to the end of the OnCreate method in **MainActivity.cs**:

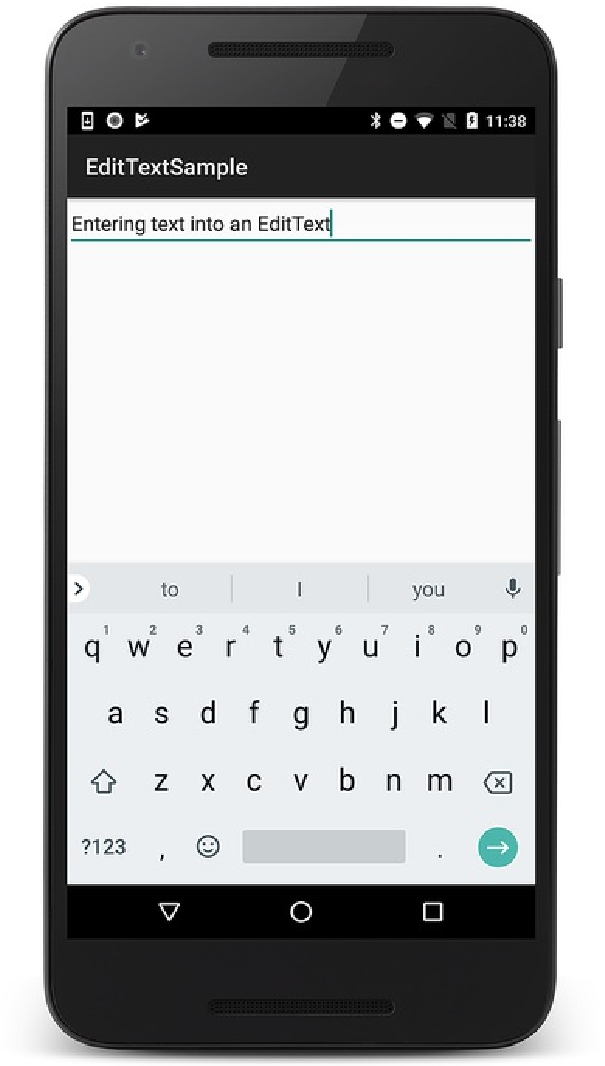
```
EditText edittext = FindViewById<EditText>(Resource.Id.edittext);
edittext.KeyPress += (object sender, View.KeyEventArgs e) => {
    e.Handled = false;
    if (e.Event.Action == KeyEventActions.Down && e.KeyCode == Keycode.Enter)
    {
        Toast.MakeText(this, edittext.Text, ToastLength.Short).Show();
        e.Handled = true;
    }
};
```

In addition, add the following `using` statement to the top of **MainActivity.cs** if it is not already present:

```
using Android.Views;
```

This code example inflates the EditText element from the layout and adds a KeyPress handler that defines the action to be made when a key is pressed while the widget has focus. In this case, the method is defined to listen for the **Enter** key (when tapped) and then pop up a Toast message with the text that has been entered. Note that the Handled property should always be `true` if the event has been handled. This is necessary to prevent the event from bubbling up (which would result in a carriage return in the text field).

Run the application and enter some text into the text field. When you press the **Enter** key, the toast will be displayed as shown on the right:



Portions of this page are modifications based on work created and shared by the Android Open Source Project and used according to terms described in the [Creative Commons 2.5 Attribution License](). This tutorial is based on the [Android Form Stuff tutorial]().

Related Links

- [EditTextSample]()

Xamarin.Android Gallery control

7/8/2021 • 3 minutes to read • Edit Online

`Gallery` is a layout widget used to display items in a horizontally scrolling list and positions the current selection at the center of the view.

> **IMPORTANT**
>
> This widget was deprecated in Android 4.1 (API level 16).

In this tutorial, you'll create a gallery of photos and then display a toast message each time a gallery item is selected.

After the `Main.axml` layout is set for the content view, the `Gallery` is captured from the layout with `FindViewById`. The `Adapter` property is then used to set a custom adapter (`ImageAdapter`) as the source for all items to be displayed in the dallery. The `ImageAdapter` is created in the next step.

To do something when an item in the gallery is clicked, an anonymous delegate is subscribed to the `ItemClick` event. It shows a `Toast` that displays the index position (zero-based) of theselected item (in a real world scenario, the position could be used to get the full sized image for some other task).

First, there are a few member variables, including an array of IDs that reference the images saved in the drawable resources directory (**Resources/drawable**).

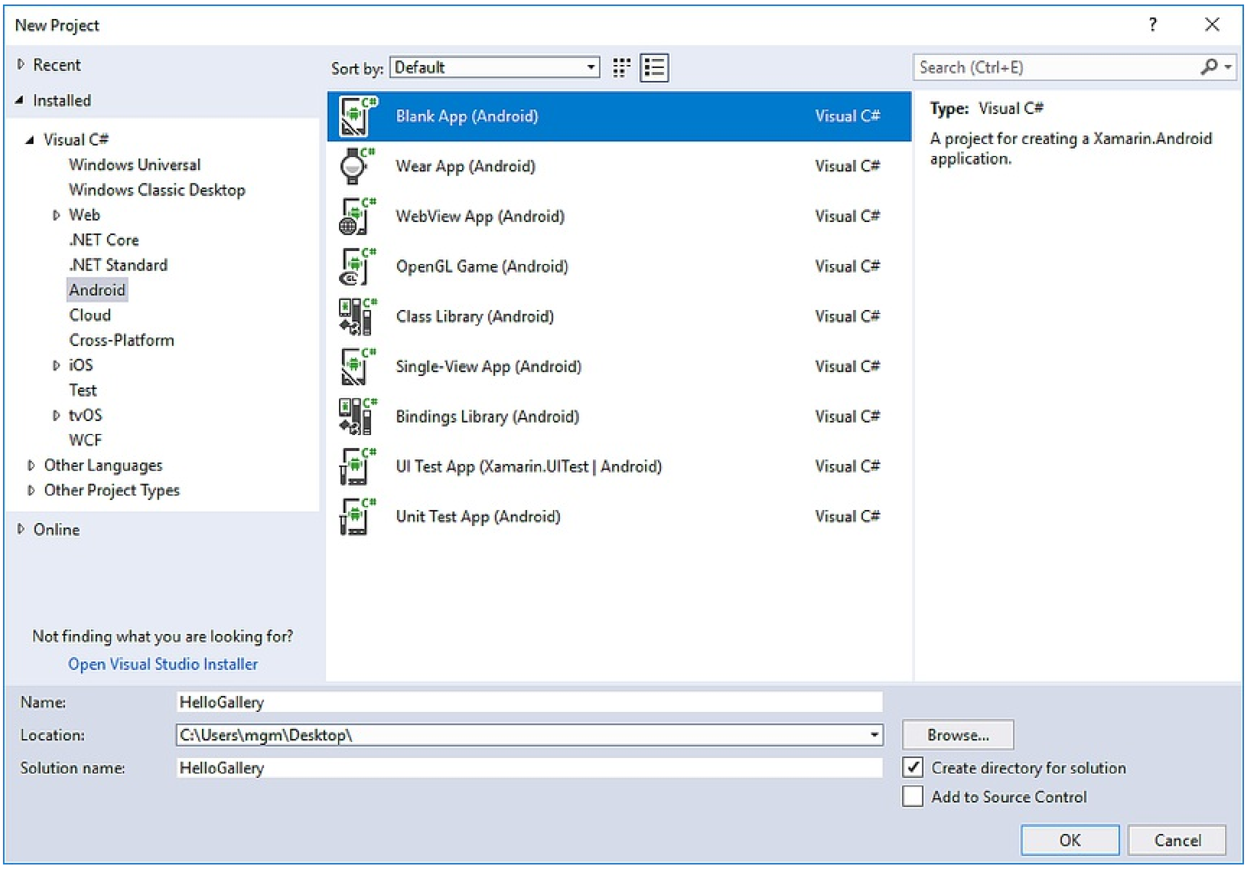
Next is the class constructor, where the `Context` for an `ImageAdapter` instance is defined and saved to a local field. Next, this implements some required methods inherited from `BaseAdapter`. The constructor and the `Count` property are self-explanatory. Normally, `GetItem(int)` should return the actual object at the specified position in the adapter, but it's ignored for this example. Likewise, `GetItemId(int)` should return the row id of the item, but it's not needed here.

The method does the work to apply an image to an `ImageView` that will be embedded in the `Gallery` In this method, the member `Context` is used to create a new `ImageView`. The `ImageView` is prepared by applying an image from the local array of drawable resources, setting the `Gallery.LayoutParams` height and width for the image, setting the scale to fit the `ImageView` dimensions, and then finally setting the background to use the styleable attribute acquired in the constructor.

See `ImageView.ScaleType` for other image scaling options.

Walkthrough

Start a new project named *HelloGallery*.



Find some photos you'd like to use, or download these sample images. Add the image files to the project's **Resources/Drawable** directory. In the **Properties** window, set the Build Action for each to **AndroidResource**.

Open **Resources/Layout/Main.axml** and insert the following:

```
<?xml version="1.0" encoding="utf-8"?>
<Gallery xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/gallery"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
/>
```

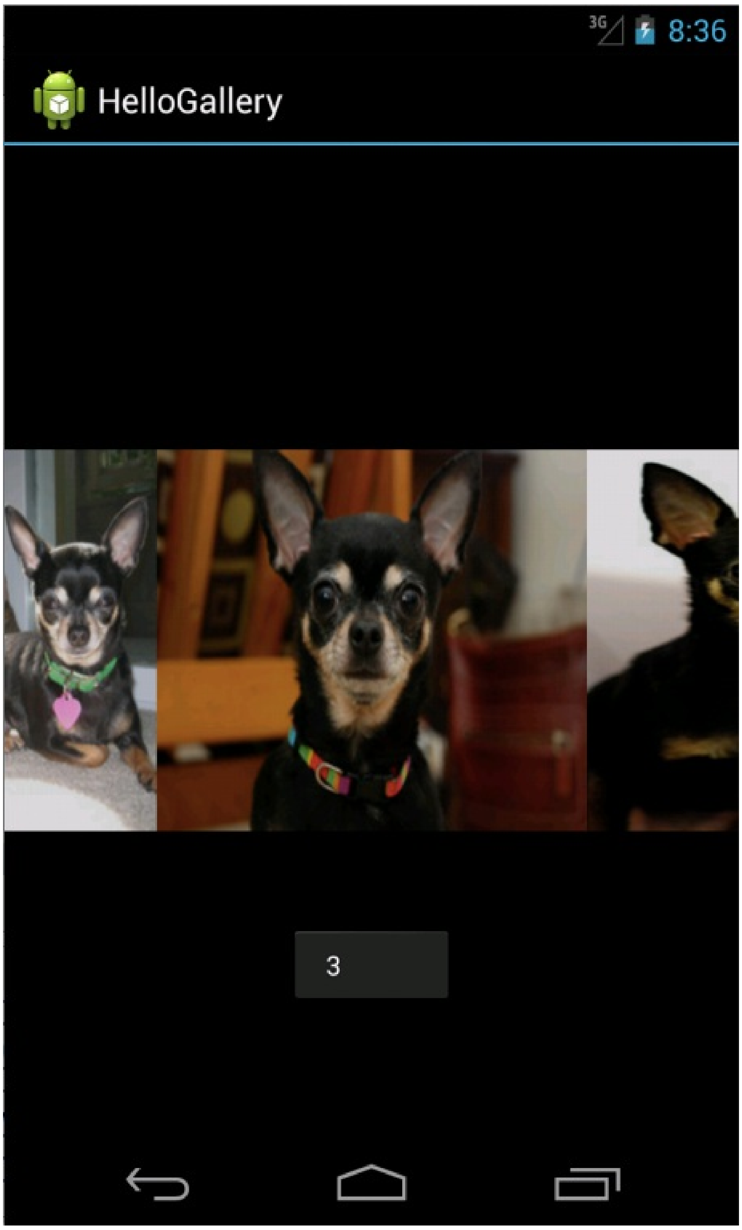
Open `MainActivity.cs` and insert the following code for the `OnCreate()` method:

```
protected override void OnCreate (Bundle bundle)
{
    base.OnCreate (bundle);

    // Set our view from the "main" layout resource
    SetContentView (Resource.Layout.Main);

    Gallery gallery = (Gallery) FindViewById<Gallery>(Resource.Id.gallery);

    gallery.Adapter = new ImageAdapter (this);

    gallery.ItemClick += delegate (object sender, Android.Widget.AdapterView.ItemClickEventArgs args) {
        Toast.MakeText (this, args.Position.ToString (), ToastLength.Short).Show ();
    };
}
```

Create a new class called `ImageAdapter` that subclasses `BaseAdapter`:

```
public class ImageAdapter : BaseAdapter
{
    Context context;

    public ImageAdapter (Context c)
    {
          context = c;
    }

    public override int Count { get { return thumbIds.Length; } }

    public override Java.Lang.Object GetItem (int position)
    {
          return null;
    }

    public override long GetItemId (int position)
    {
          return 0;
    }

    // create a new ImageView for each item referenced by the Adapter
    public override View GetView (int position, View convertView, ViewGroup parent)
    {
          ImageView i = new ImageView (context);

          i.SetImageResource (thumbIds[position]);
          i.LayoutParameters = new Gallery.LayoutParams (150, 100);
          i.SetScaleType (ImageView.ScaleType.FitXy);

          return i;
    }

    // references to our images
    int[] thumbIds = {
            Resource.Drawable.sample_1,
            Resource.Drawable.sample_2,
            Resource.Drawable.sample_3,
            Resource.Drawable.sample_4,
            Resource.Drawable.sample_5,
            Resource.Drawable.sample_6,
            Resource.Drawable.sample_7
     };
}
```

Run the application. It should look like the screenshot below:



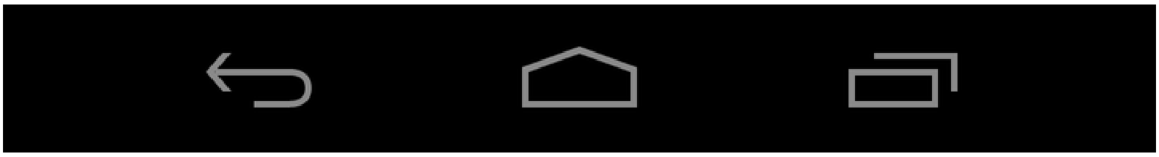
References

- `BaseAdapter`
- `Gallery`
- `ImageView`

Xamarin.Android Navigation Bar

7/8/2021 • 2 minutes to read • Edit Online

Android 4 introduced a new system user interface feature called a *Navigation Bar*, which provides navigation controls on devices that don't include hardware buttons for **Home**, **Back**, and **Menu**. The following screenshot shows the Navigation Bar from a Nexus Prime device:

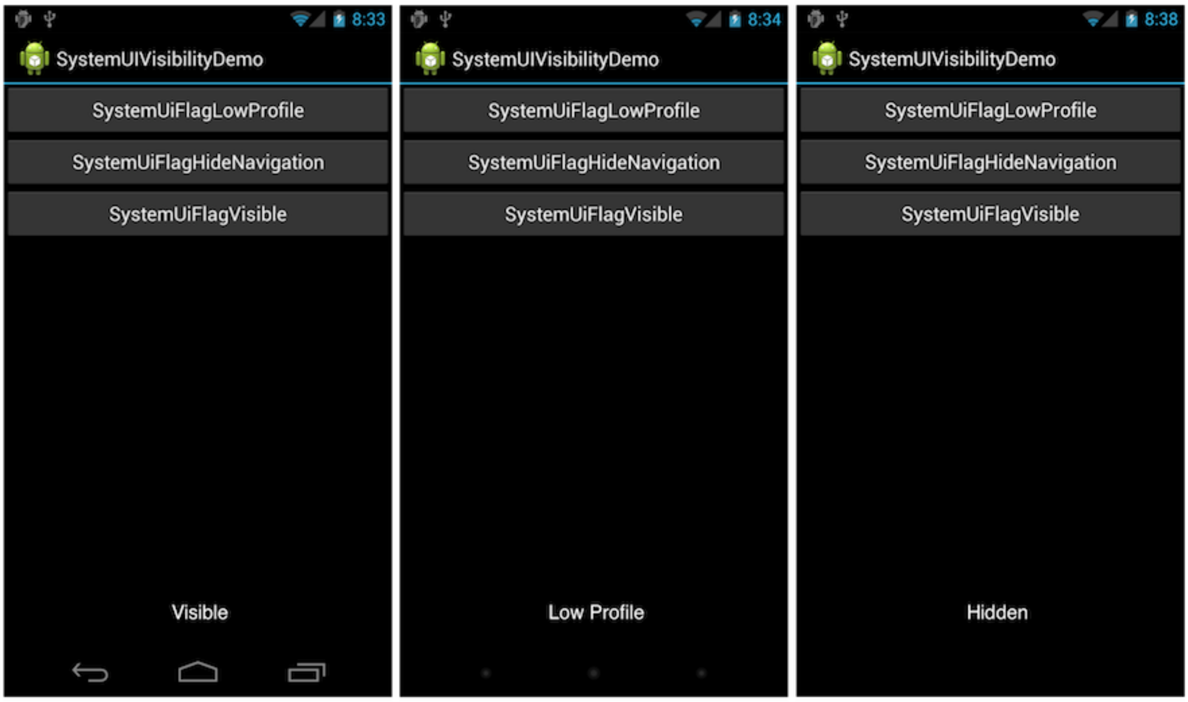


Several new flags are available that control the visibility of the Navigation Bar and its controls, as well as the visibility of the System Bar that was introduced in Android 3. The flags are defined in the `Android.View.View` class and are listed below:

- `SystemUiFlagVisible` – Makes the Navigation Bar visible.
- `SystemUiFlagLowProfile` – Dims out controls in the Navigation Bar.
- `SystemUiFlagHideNavigation` – Hides the Navigation Bar.

These flags can be applied to any view in the view hierarchy by setting the `SystemUiVisibility` property. If multiple views have this property set, the system combines them with an OR operation and applies them so long as the window in which the flags are set retains focus. When you remove a view, any flags it has set will also be removed.

The following example shows a simple application where clicking any of the buttons changes the `SystemUiVisibility`:



The code to change the `SystemUiVisibility` sets the property on a `TextView` from each button's click event handler as shown below:

```
var tv = FindViewById<TextView> (Resource.Id.systemUiFlagTextView);
var lowProfileButton = FindViewById<Button>(Resource.Id.lowProfileButton);
var hideNavButton = FindViewById<Button> (Resource.Id.hideNavigation);
var visibleButton = FindViewById<Button> (Resource.Id.visibleButton);

lowProfileButton.Click += delegate {
    tv.SystemUiVisibility =
        (StatusBarVisibility)View.SystemUiFlagLowProfile;
};

hideNavButton.Click += delegate {
    tv.SystemUiVisibility =
       (StatusBarVisibility)View.SystemUiFlagHideNavigation;
};

visibleButton.Click += delegate {
    tv.SystemUiVisibility = (StatusBarVisibility)View.SystemUiFlagVisible;
}
```

Also, a `SystemUiVisibility` change raises a `SystemUiVisibilityChange` event. Just like setting the `SystemUiVisibility` property, a handler for the `SystemUiVisibilityChange` event can be registered for any view in the hierarchy. For example, the code below uses the `TextView` instance to register for the event:

```
tv.SystemUiVisibilityChange +=
  delegate(object sender, View.SystemUiVisibilityChangeEventArgs e) {
        tv.Text = String.Format ("Visibility = {0}", e.Visibility);
  };
```

Related Links

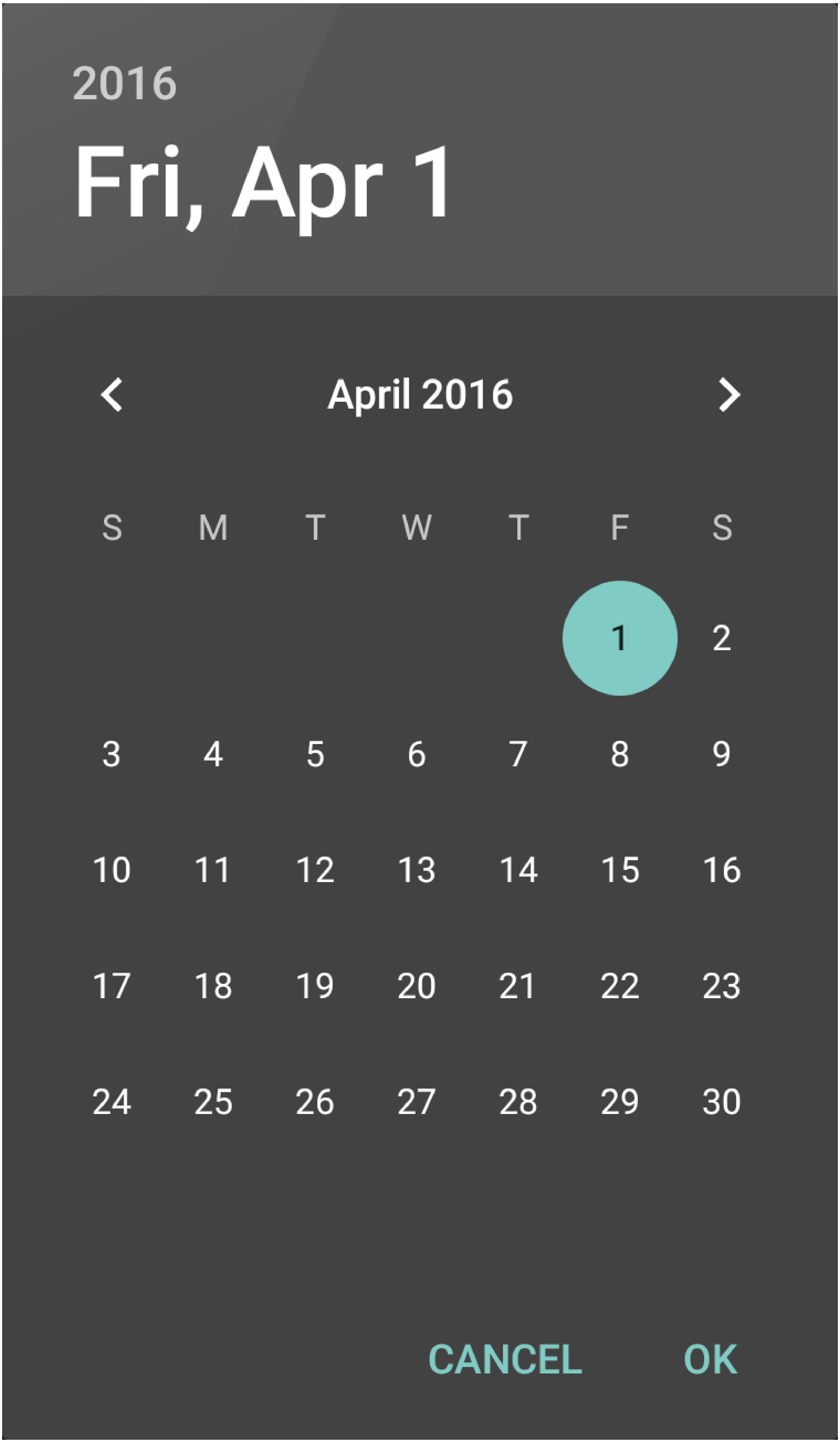
- SystemUIVisibilityDemo (sample)

Picker controls for Xamarin.Android

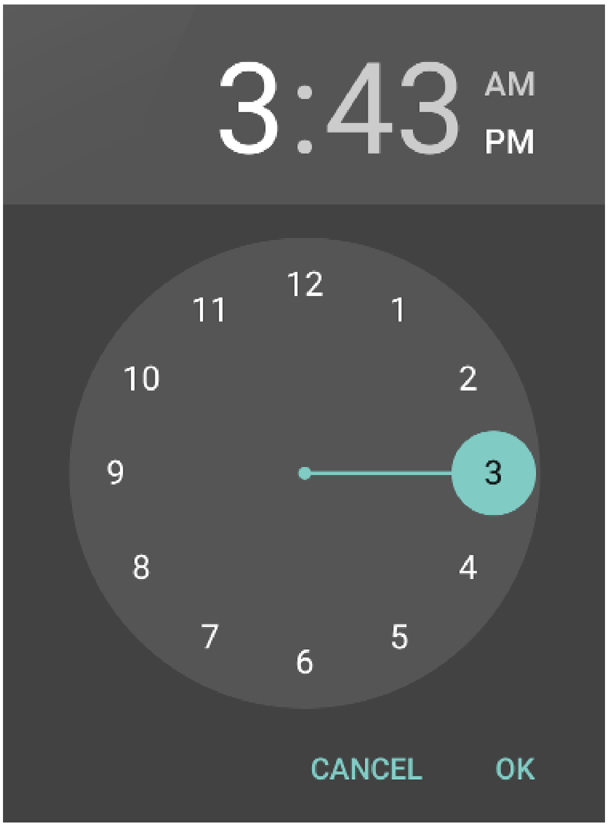
7/8/2021 • 2 minutes to read • Edit Online

Pickers are UI elements that allow the user to pick a date or a time by using dialogs that are provided by Android:

- Date Picker is used to select a date (year, month, and day).



- Time Picker is used to select a time (hour, minute, and AM/PM).


3:43
AM
PM
12
1
2
3
4
5
6
7
8
9
10
11
CANCEL
OK

Android Date Picker

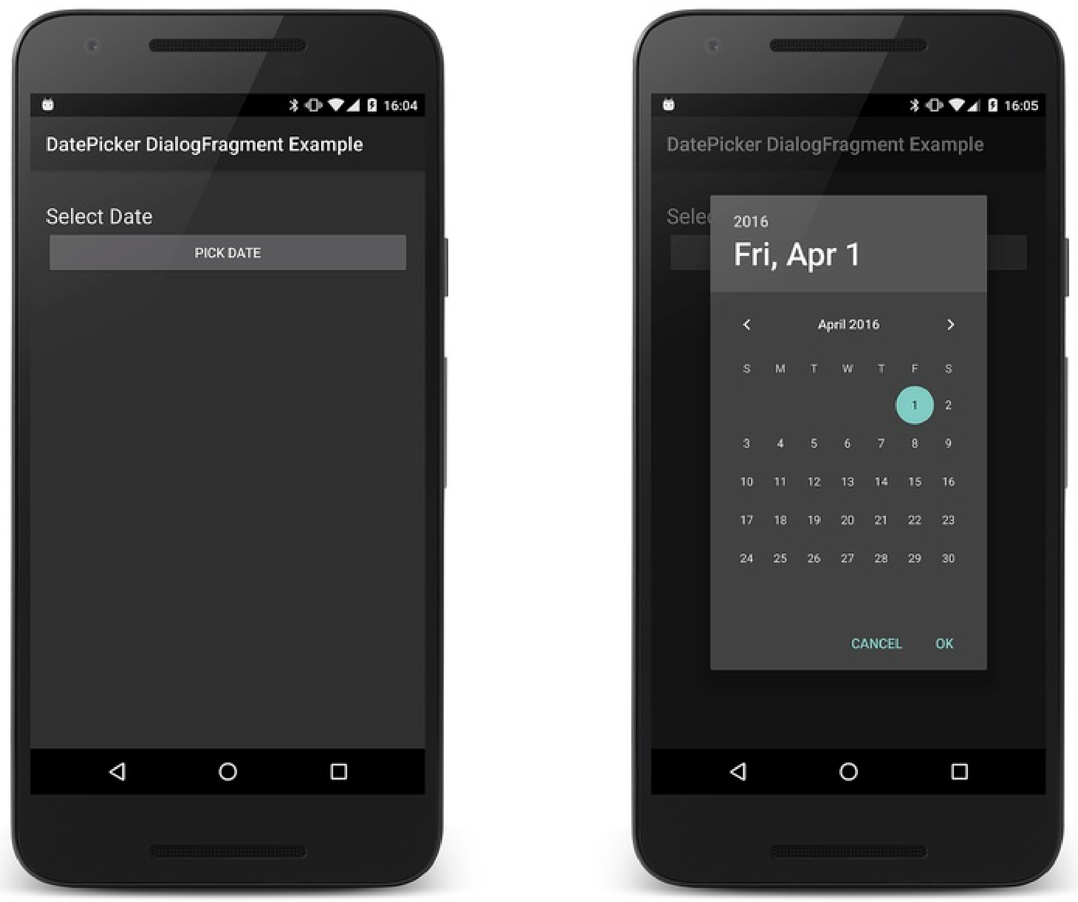
7/8/2021 • 4 minutes to read • Edit Online

Overview

There are occasions when a user must input data into an Android application. To assist with this, the Android framework provides the `DatePicker` widget and the `DatePickerDialog`. The `DatePicker` allows users to select the year, month, and day in a consistent interface across devices and applications. The `DatePickerDialog` is a helper class that encapsulates the `DatePicker` in a dialog.

Modern Android applications should display the `DatePickerDialog` in a `DialogFragment`. This will allow an application to display the DatePicker as a popup dialog or embedded in an Activity. In addition, the `DialogFragment` will manage the lifecycle and display of the dialog, reducing the amount of code that must be implemented.

This guide will demonstrate how to use the `DatePickerDialog`, wrapped in a `DialogFragment`. The sample application will display the `DatePickerDialog` as a modal dialog when the user clicks a button on an Activity. When the date is set by the user, a `TextView` will update with the date that was selected.



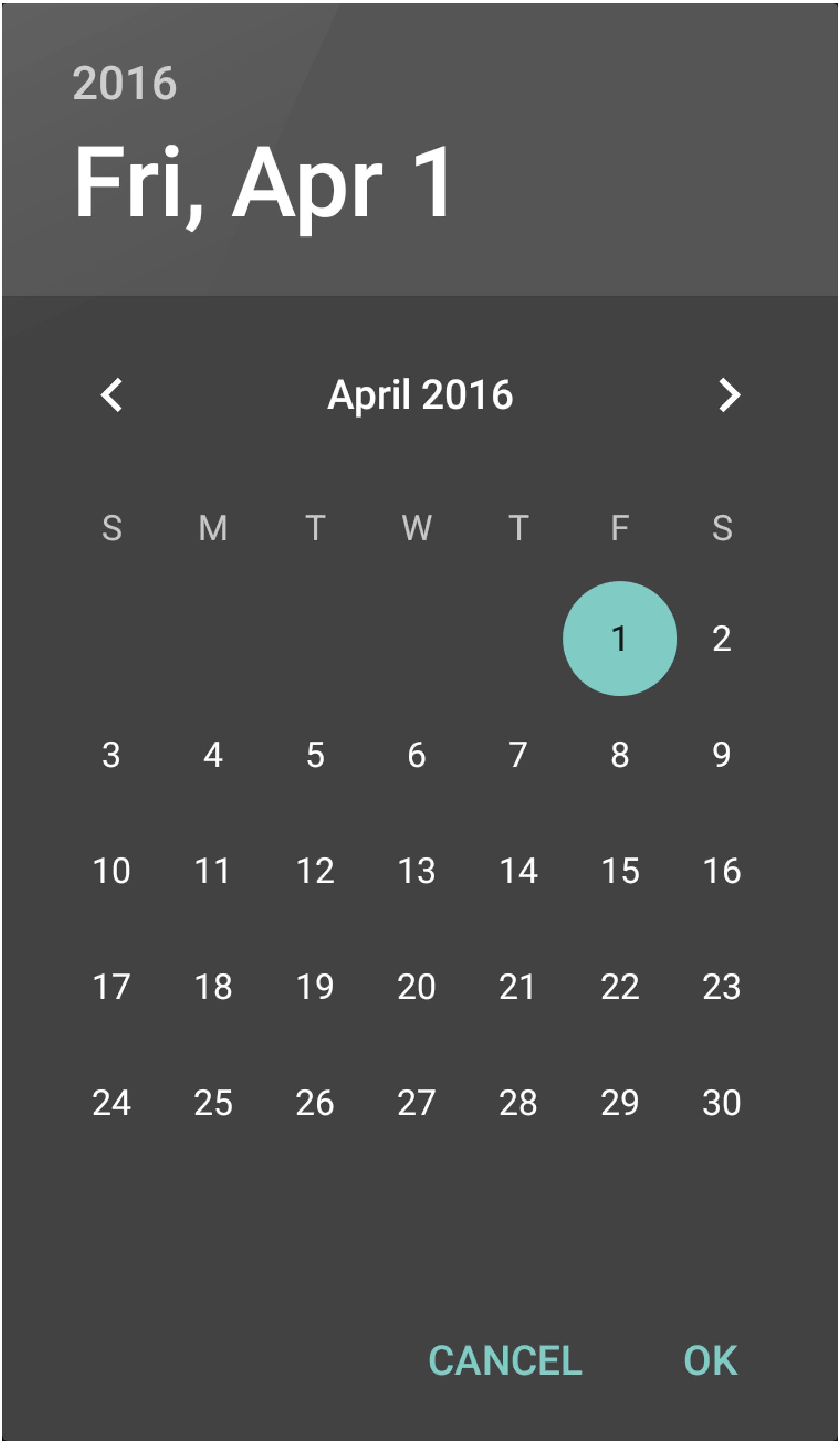
Requirements

The sample application for this guide targets Android 4.1 (API level 16) or higher, but is applicable to Android 3.0

(API level 11 or higher). It is possible to support older versions of Android with the addition of the Android Support Library v4 to the project and some code changes.

Using the DatePicker

This sample will extend `DialogFragment`. The subclass will host and display a `DatePickerDialog`:


2016
Fri, Apr 1
April 2016
S M T W T F S
1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
CANCEL
OK

When the user selects a date and clicks the **OK** button, the `DatePickerDialog` will call the method `IOnDateSetListener.OnDateSet`. This interface is implemented by the hosting `DialogFragment`. If the user clicks the **Cancel** button, then fragment and dialog will dismiss themselves.

There are several ways the `DialogFragment` can return the selected date to the hosting activity:

1. **Invoke a method or set a property** – The Activity can provide a property or method specifically for setting this value.

2. **Raise an event** – The `DialogFragment` can define an event that will be raised when `OnDateSet` is invoked.

3. **Use an `Action`** – The `DialogFragment` can invoke an `Action<DateTime>` to display the date in the Activity. The Activity will provide the `Action<DateTime` when instantiating the `DialogFragment`. This sample will use the third technique, and require that the Activity supply an `Action<DateTime>` to the `DialogFragment`.

Extending DialogFragment

The first step in displaying a `DatePickerDialog` is to subclass `DialogFragment` and have it implement the `IOnDateSetListener` interface:

```
public class DatePickerFragment : DialogFragment,
                                  DatePickerDialog.IOnDateSetListener
{
    // TAG can be any string of your choice.
    public static readonly string TAG = "X:" + typeof (DatePickerFragment).Name.ToUpper();

    // Initialize this value to prevent NullReferenceExceptions.
    Action<DateTime> _dateSelectedHandler = delegate { };

    public static DatePickerFragment NewInstance(Action<DateTime> onDateSelected)
    {
        DatePickerFragment frag = new DatePickerFragment();
        frag._dateSelectedHandler = onDateSelected;
        return frag;
    }

    public override Dialog OnCreateDialog(Bundle savedInstanceState)
    {
        DateTime currently = DateTime.Now;
        DatePickerDialog dialog = new DatePickerDialog(Activity,
                                                       this,
                                                       currently.Year,
                                                       currently.Month - 1,
                                                       currently.Day);
        return dialog;
    }

    public void OnDateSet(DatePicker view, int year, int monthOfYear, int dayOfMonth)
    {
        // Note: monthOfYear is a value between 0 and 11, not 1 and 12!
        DateTime selectedDate = new DateTime(year, monthOfYear + 1, dayOfMonth);
        Log.Debug(TAG, selectedDate.ToLongDateString());
        _dateSelectedHandler(selectedDate);
    }
}
```

The `NewInstance` method is invoked to instantiate a new `DatePickerFragment`. This method takes an `Action<DateTime>` that will be invoked when the user clicks on the **OK** button in the `DatePickerDialog`.

When the fragment is to be displayed, Android will call the method `OnCreateDialog`. This method will create a new `DatePickerDialog` object and initialize it with the current date and the callback object (which is the current instance of the `DatePickerFragment`).

> **NOTE**
>
> Be aware that the value of the month when `IOnDateSetListener.OnDateSet` is invoked is in the range of 0 to 11, and not 1 to 12. The day of the month will be in the range of 1 to 31 (depending on which month was selected).

Showing the DatePickerFragment

Now that the `DialogFragment` has been implemented, this section will examine how to use the fragment in an Activity. In the sample app that accompanies this guide, the Activity will instantiate the `DialogFragment` using the `NewInstance` factory method and then display it invoke `DialogFragment.Show`. As a part of instantiating the `DialogFragment`, the Activity passes an `Action<DateTime>`, which will display the date in a `TextView` that is hosted by the Activity:

```
[Activity(Label = "@string/app_name", MainLauncher = true, Icon = "@drawable/icon")]
public class MainActivity : Activity
{
    TextView _dateDisplay;
    Button _dateSelectButton;

    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);
        SetContentView(Resource.Layout.Main);

        _dateDisplay = FindViewById<TextView>(Resource.Id.date_display);
        _dateSelectButton = FindViewById<Button>(Resource.Id.date_select_button);
        _dateSelectButton.Click += DateSelect_OnClick;
    }

    void DateSelect_OnClick(object sender, EventArgs eventArgs)
    {
        DatePickerFragment frag = DatePickerFragment.NewInstance(delegate(DateTime time)
                                                                 {
                                                                     _dateDisplay.Text =
time.ToLongDateString();
                                                                 });
        frag.Show(FragmentManager, DatePickerFragment.TAG);
    }
}
```

Summary

This sample discussed how to display a `DatePicker` widget as a popup modal dialog as a part of an Android Activity. It provided a sample DialogFragment implementation and discussed the `IOnDateSetListener` interface. This sample also demonstrated how the DialogFragment may interact with the host Activity to display the selected date.

Related Links

- DialogFragment
- DatePicker
- DatePickerDialog
- DatePickerDialog.IOnDateSetListener
- Select A Date

Android Time Picker

7/8/2021 • 7 minutes to read • Edit Online

To provide a way for the user to select a time, you can use TimePicker. Android apps typically use `TimePicker` with TimePickerDialog for selecting a time value – this helps to ensure a consistent interface across devices and applications. `TimePicker` allows users to select the time of day in either 24-hour or 12-hour AM/PM mode. `TimePickerDialog` is a helper class that encapsulates the `TimePicker` in a dialog.

Overview

Modern Android applications display the `TimePickerDialog` in a DialogFragment. This makes it possible for an application to display the `TimePicker` as a popup dialog or embed it in an Activity. In addition, the `DialogFragment` manages the lifecycle and display of the dialog, reducing the amount of code that must be implemented.

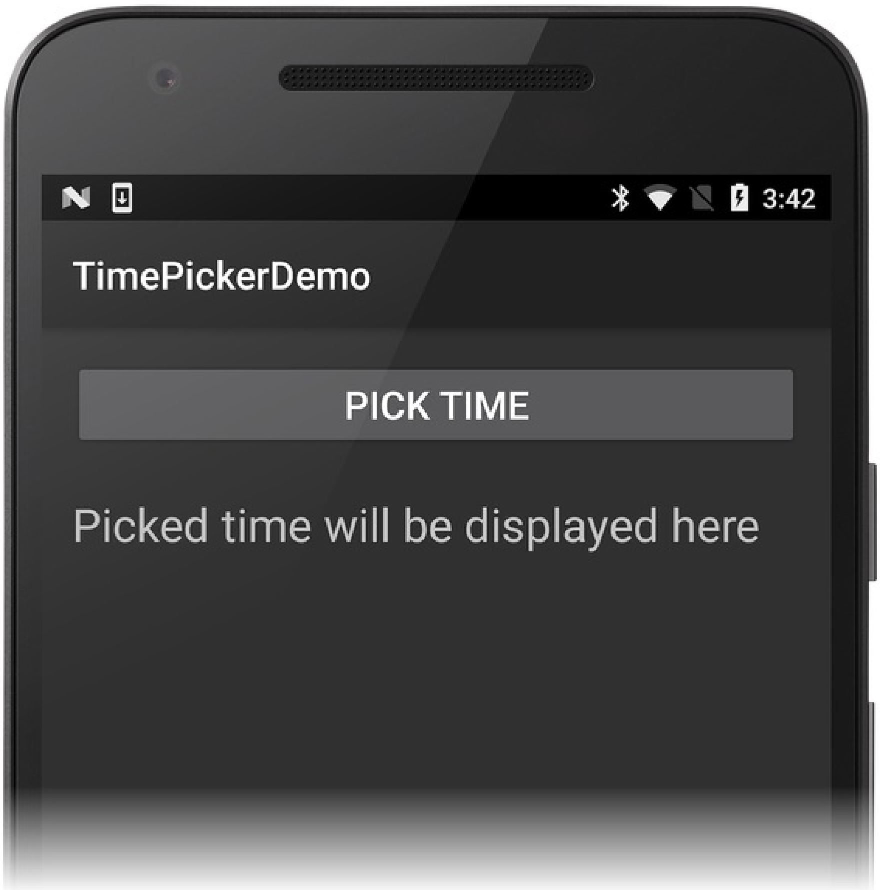
This guide demonstrates how to use the `TimePickerDialog`, wrapped in a `DialogFragment`. The sample application displays the `TimePickerDialog` as a modal dialog when the user clicks a button on an Activity. When the time is set by the user, the dialog exits and a handler updates a `TextView` on the Activity screen with the time that was selected.

Requirements

The sample application for this guide targets Android 4.1 (API level 16) or higher, but is can be used with Android 3.0 (API level 11 or higher). It is possible to support older versions of Android with the addition of the Android Support Library v4 to the project and some code changes.

Using the TimePicker

This example extends `DialogFragment`; the subclass implementation of `DialogFragment` (called `TimePickerFragment` below) hosts and displays a `TimePickerDialog`. When the sample app is first launched, it displays a **PICK TIME** button above a `TextView` that will be used to display the selected time:



When you click the **PICK TIME** button, the example app launches the `TimePickerDialog` as seen in this screenshot:

In the `TimePickerDialog`, selecting a time and clicking the **OK** button causes the `TimePickerDialog` to invoke the method IOnTimeSetListener.OnTimeSet. This interface is implemented by the hosting `DialogFragment` (`TimePickerFragment`, described below). Clicking the **Cancel** button causes the fragment and dialog to be dismissed.

`DialogFragment` returns the selected time to the hosting Activity in one of three ways:

1. **Invoking a method or setting a property** – The Activity can provide a property or method specifically for setting this value.

2. **Raising an event** – The `DialogFragment` can define an event that will be raised when `OnTimeSet` is invoked.

3. **Using an `Action`** – The `DialogFragment` can invoke an `Action<DateTime>` to display the time in the Activity. The Activity will provide the `Action<DateTime` when instantiating the `DialogFragment`.

This sample will use the third technique, which requires that the Activity supply an `Action<DateTime>` handler to the `DialogFragment` .

Start an App Project

Start a new Android project called **TimePickerDemo** (if you are not familiar with creating Xamarin.Android projects, see Hello, Android to learn how to create a new project).

Edit **Resources/layout/Main.axml** and replace its contents with the following XML:

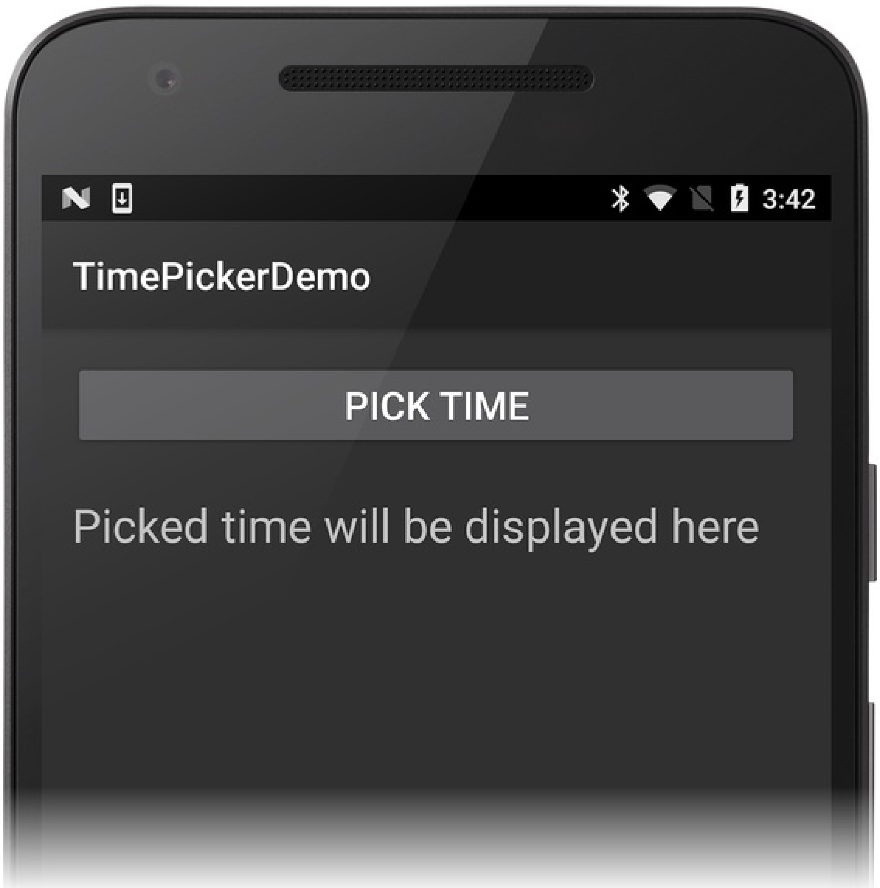
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_gravity="center_horizontal"
    android:padding="16dp">
    <Button
        android:id="@+id/select_button"
        android:paddingLeft="24dp"
        android:paddingRight="24dp"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="PICK TIME"
        android:textSize="20dp" />
    <TextView
        android:id="@+id/time_display"
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:paddingTop="22dp"
        android:text="Picked time will be displayed here"
        android:textSize="24dp" />
</LinearLayout>
```

This is a basic LinearLayout with a TextView that displays the time and a Button that opens the `TimePickerDialog` . Note that this layout uses hard-coded strings and dimensions to make the app simpler and easier to understand – a production app normally uses resources for these values (as can be seen in the DatePicker code example).

Edit **MainActivity.cs** and replace its contents with the following code:

```
using Android.App;
using Android.Widget;
using Android.OS;
using System;
using Android.Util;
using Android.Text.Format;

namespace TimePickerDemo
{
    [Activity(Label = "TimePickerDemo", MainLauncher = true, Icon = "@drawable/icon")]
    public class MainActivity : Activity
    {
        TextView timeDisplay;
        Button timeSelectButton;

        protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);
            SetContentView(Resource.Layout.Main);
            timeDisplay = FindViewById<TextView>(Resource.Id.time_display);
            timeSelectButton = FindViewById<Button>(Resource.Id.select_button);
        }
    }
}
```

When you build and run this example, you should see an initial screen similar to the following screen shot:



Clicking the **PICK TIME** button does nothing because the `DialogFragment` has not yet been implemented to display the `TimePicker`. The next step is to create this `DialogFragment`.

Extending DialogFragment

To extend `DialogFragment` for use with `TimePicker`, it is necessary to create a subclass that is derived from

`DialogFragment` and implements `TimePickerDialog.IOnTimeSetListener`. Add the following class to **MainActivity.cs**:

```
public class TimePickerFragment : DialogFragment, TimePickerDialog.IOnTimeSetListener
{
    public static readonly string TAG = "MyTimePickerFragment";
    Action<DateTime> timeSelectedHandler = delegate { };

    public static TimePickerFragment NewInstance(Action<DateTime> onTimeSelected)
    {
        TimePickerFragment frag = new TimePickerFragment();
        frag.timeSelectedHandler = onTimeSelected;
        return frag;
    }

    public override Dialog OnCreateDialog (Bundle savedInstanceState)
    {
        DateTime currentTime = DateTime.Now;
        bool is24HourFormat = DateFormat.Is24HourFormat(Activity);
        TimePickerDialog dialog = new TimePickerDialog
            (Activity, this, currentTime.Hour, currentTime.Minute, is24HourFormat);
        return dialog;
    }

    public void OnTimeSet(TimePicker view, int hourOfDay, int minute)
    {
        DateTime currentTime = DateTime.Now;
        DateTime selectedTime = new DateTime(currentTime.Year, currentTime.Month, currentTime.Day,
hourOfDay, minute, 0);
        Log.Debug(TAG, selectedTime.ToLongTimeString());
        timeSelectedHandler (selectedTime);
    }
}
```

This `TimePickerFragment` class is broken down into smaller pieces and explained in the next section.

DialogFragment Implementation

`TimePickerFragment` implements several methods: a factory method, a Dialog instantiation method, and the `OnTimeSet` handler method required by `TimePickerDialog.IOnTimeSetListener`.

- `TimePickerFragment` is a subclass of `DialogFragment`. It also implements the `TimePickerDialog.IOnTimeSetListener` interface (that is, it supplies the required `OnTimeSet` method):

  ```
  public class TimePickerFragment : DialogFragment, TimePickerDialog.IOnTimeSetListener
  ```

- `TAG` is initialized for logging purposes (*MyTimePickerFragment* can be changed to whatever string you want to use). The `timeSelectedHandler` Action is initialized to an empty delegate to prevent null reference exceptions:

  ```
  public static readonly string TAG = "MyTimePickerFragment";
  Action<DateTime> timeSelectedHandler = delegate { };
  ```

- The `NewInstance` factory method is called to instantiate a new `TimePickerFragment`. This method takes an `Action<DateTime>` handler that is invoked when the user clicks the **OK** button in the `TimePickerDialog`:

```
public static TimePickerFragment NewInstance(Action<DateTime> onTimeSelected)
{
    TimePickerFragment frag = new TimePickerFragment();
    frag.timeSelectedHandler = onTimeSelected;
    return frag;
}
```

- When the fragment is to be displayed, Android calls the `DialogFragment` method OnCreateDialog. This method creates a new `TimePickerDialog` object and initializes it with the Activity, the callback object (which is the current instance of the `TimePickerFragment`), and the current time:

```
public override Dialog OnCreateDialog (Bundle savedInstanceState)
{
    DateTime currentTime = DateTime.Now;
    bool is24HourFormat = DateFormat.Is24HourFormat(Activity);
    TimePickerDialog dialog = new TimePickerDialog
        (Activity, this, currentTime.Hour, currentTime.Minute, is24HourFormat);
    return dialog;
}
```

- When the user changes the time setting in the `TimePicker` dialog, the `OnTimeSet` method is invoked. `OnTimeSet` creates a `DateTime` object using the current date and merges in the time (hour and minute) selected by the user:

```
public void OnTimeSet(TimePicker view, int hourOfDay, int minute)
{
    DateTime currentTime = DateTime.Now;
    DateTime selectedTime = new DateTime(currentTime.Year, currentTime.Month, currentTime.Day,
hourOfDay, minute, 0);
```

- This `DateTime` object is passed to the `timeSelectedHandler` that is registered with the `TimePickerFragment` object at creation time. `OnTimeSet` invokes this handler to update the Activity's time display to the selected time (this handler is implemented in the next section):

```
timeSelectedHandler (selectedTime);
```

Displaying the TimePickerFragment

Now that the `DialogFragment` has been implemented, it is time to instantiate the `DialogFragment` using the `NewInstance` factory method and display it by invoking DialogFragment.Show:

Add the following method to `MainActivity`:

```
void TimeSelectOnClick (object sender, EventArgs eventArgs)
{
    TimePickerFragment frag = TimePickerFragment.NewInstance (
        delegate (DateTime time)
        {
            timeDisplay.Text = time.ToShortTimeString();
        });

    frag.Show(FragmentManager, TimePickerFragment.TAG);
}
```

After `TimeSelectOnClick` instantiates a `TimePickerFragment`, it creates and passes in a delegate for an anonymous method that updates the Activity's time display with the passed-in time value. Finally, it launches the `TimePicker` dialog fragment (via `DialogFragment.Show`) to display the `TimePicker` to the user.

At the end of the `OnCreate` method, add the following line to attach the event handler to the **PICK TIME** button that launches the dialog:

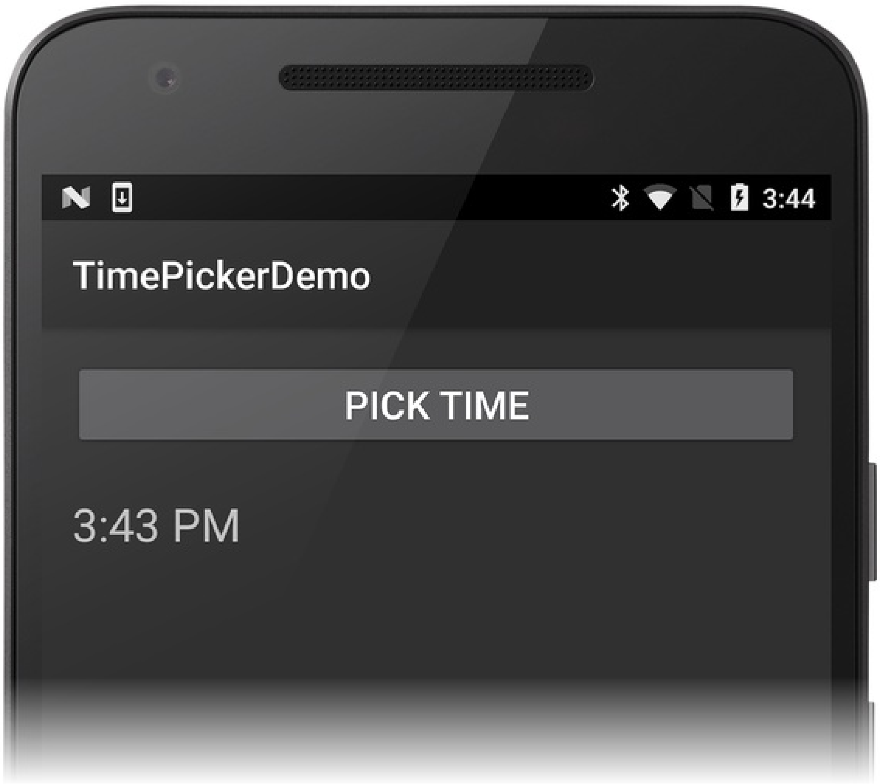
```
timeSelectButton.Click += TimeSelectOnClick;
```

When the **PICK TIME** button is clicked, `TimeSelectOnClick` will be invoked to display the `TimePicker` dialog fragment to the user.

Try It!

Build and run the app. When you click the **PICK TIME** button, the `TimePickerDialog` is displayed in the default time format for the Activity (in this case, 12-hour AM/PM mode):

When you click **OK** in the `TimePicker` dialog, the handler updates the Activity's `TextView` with the chosen time and then exits:



Next, add the following line of code to `OnCreateDialog` immediately after `is24HourFormat` is declared and initialized:

```
is24HourFormat = true;
```

This change forces the flag passed to the `TimePickerDialog` constructor to be `true` so that 24-hour mode is used instead of the time format of the hosting Activity. When you build and run the app again, click the **PICK TIME** button, the `TimePicker` dialog is now displayed in 24 hour format:

Because the handler calls DateTime.ToShortTimeString to print the time to the Activity's `TextView`, the time is still printed in the default 12-hour AM/PM format.

Summary

This article explained how to display a `TimePicker` widget as a popup modal dialog from an Android Activity. It provided a sample `DialogFragment` implementation and discussed the `IOnTimeSetListener` interface. This sample also demonstrated how the `DialogFragment` can interact with the host Activity to display the selected time.

Related Links

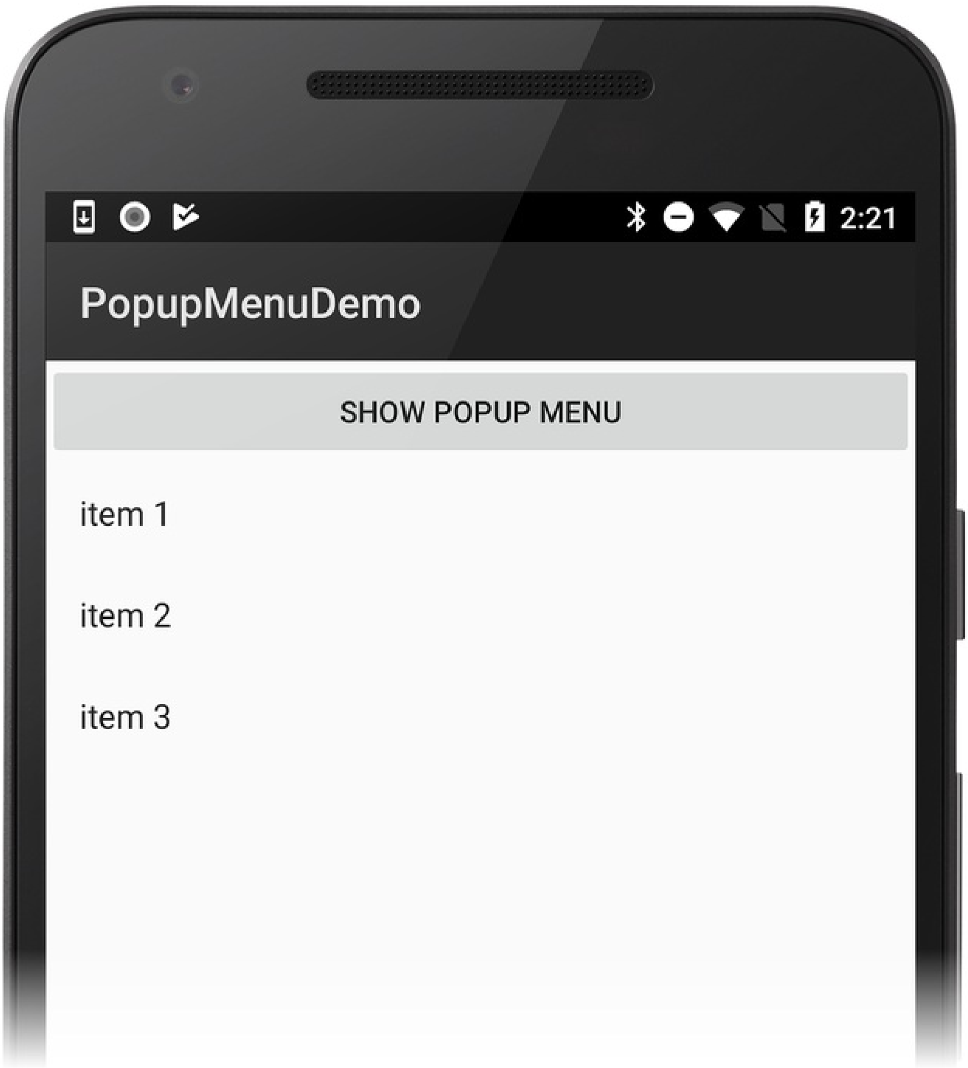
- DialogFragment
- TimePicker

- TimePickerDialog
- TimePickerDialog.IOnTimeSetListener
- TimePickerDemo (sample)

Xamarin.Android PopUp Menu

7/8/2021 • 2 minutes to read • Edit Online

The PopupMenu (also called a *shortcut menu*) is a menu that is anchored to a particular view. In the following example, a single Activity contains a button. When the user taps the button, a three-item popup menu is displayed:



Creating a Popup Menu

The first step is to create a menu resource file for the menu and place it in **Resources/menu**. For example, the following XML is the code for the three-item menu displayed in the previous screenshot, **Resources/menu/popup_menu.xml**:

```xml
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/item1"
          android:title="item 1" />
    <item android:id="@+id/item1"
          android:title="item 2" />
    <item android:id="@+id/item1"
          android:title="item 3" />
</menu>
```

Next, create an instance of `PopupMenu` and anchor it to its view. When you create an instance of `PopupMenu`, you pass its constructor a reference to the `Context` as well as the view to which the menu will be attached. As a result, the popup menu is anchored to this view during its construction.

In the following example, the `PopupMenu` is created in the click event handler for the button (which is named `showPopupMenu`). This button is also the view to which the `PopupMenu` is anchored, as shown in the following code example:

```csharp
showPopupMenu.Click += (s, arg) => {
    PopupMenu menu = new PopupMenu (this, showPopupMenu);
};
```

Finally, the popup menu must be *inflated* with the menu resource that was created earlier. In the following example, the call to the menu's Inflate method is added and its Show method is called to display it:

```csharp
showPopupMenu.Click += (s, arg) => {
    PopupMenu menu = new PopupMenu (this, showPopupMenu);
    menu.Inflate (Resource.Menu.popup_menu);
    menu.Show ();
};
```

Handling Menu Events

When the user selects a menu item, the MenuItemClick click event will be raised and the menu will be dismissed. Tapping anywhere outside the menu will simply dismiss it. In either case, when the menu is dismissed, its DismissEvent will be raised. The following code adds event handlers for both the `MenuItemClick` and `DismissEvent` events:

```csharp
showPopupMenu.Click += (s, arg) => {
    PopupMenu menu = new PopupMenu (this, showPopupMenu);
    menu.Inflate (Resource.Menu.popup_menu);

    menu.MenuItemClick += (s1, arg1) => {
        Console.WriteLine ("{0} selected", arg1.Item.TitleFormatted);
    };

    menu.DismissEvent += (s2, arg2) => {
        Console.WriteLine ("menu dismissed");
    };
    menu.Show ();
};
```

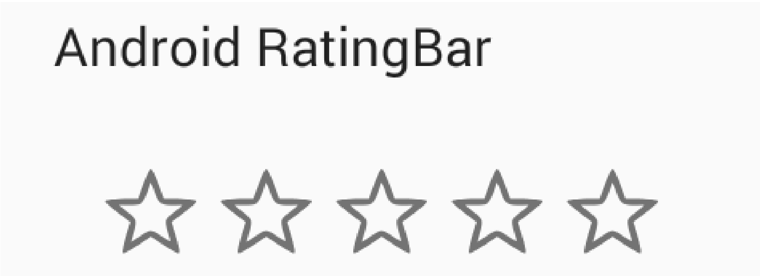
Related Links

- PopupMenuDemo (sample)

Xamarin.Android RatingBar

7/8/2021 • 2 minutes to read • Edit Online

A RatingBar is a UI widget that displays a rating from one to five stars. The user may select a rating by taping on a star In this section, you'll create a widget that allows the user to provide a rating, with the `RatingBar` widget.



Creating a RatingBar

1. Open the **Resource/layout/Main.axml** file and add the `RatingBar` element (inside the `LinearLayout`):

```
<RatingBar android:id="@+id/ratingbar"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:numStars="5"
        android:stepSize="1.0"/>
```

   The `android:numStars` attribute defines how many stars to display for the rating bar. The `android:stepSize` attribute defines the granularity for each star (for example, a value of `0.5` would allow half-star ratings).

2. To do something when a new rating has been set, add the following code to the end of the `OnCreate()` method:

```
RatingBar ratingbar = FindViewById<RatingBar>(Resource.Id.ratingbar);

ratingbar.RatingBarChange += (o, e) => {
        Toast.MakeText(this, "New Rating: " + ratingbar.Rating.ToString (), ToastLength.Short).Show
();
};
```

   This captures the `RatingBar` widget from the layout with `FindViewById` and then sets an event method then defines the action to perform when the user sets a rating. In this case, a simple `Toast` message displays the new rating.

3. Run the application.

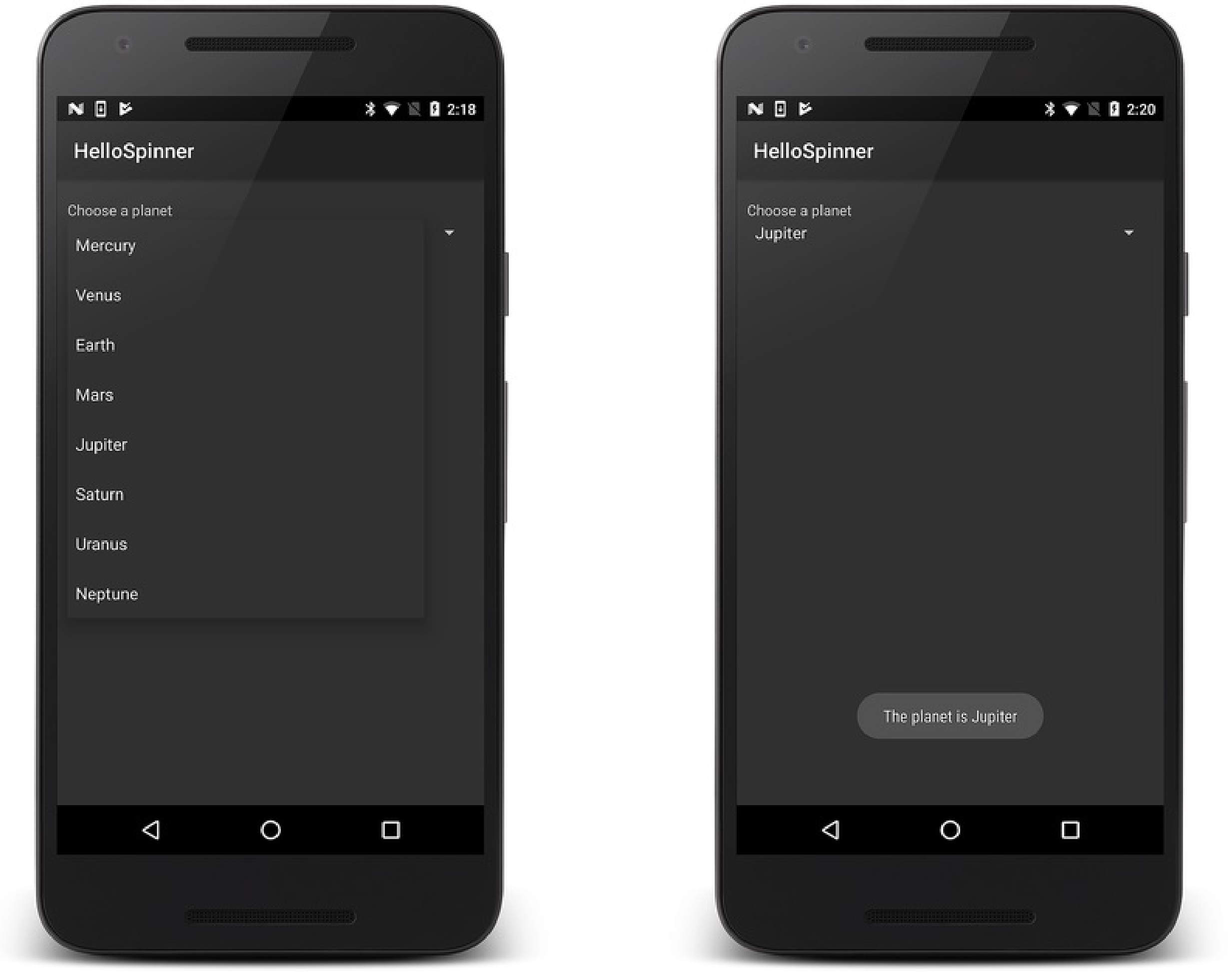
Xamarin.Android Spinner

7/8/2021 • 4 minutes to read • Edit Online

`Spinner` is a widget that presents a drop-down list for selecting items. This guide explains how to create a simple app that displays a list of choices in a Spinner, followed by modifications that display other values associated with the selected choice.

Basic Spinner

In the first part of this tutorial, you'll create a simple spinner widget that displays a list of planets. When a planet is selected, a toast message displays the selected item:



Start a new project named **HelloSpinner**.

Open **Resources/Layout/Main.axml** and insert the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:padding="10dip"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dip"
        android:text="@string/planet_prompt"
    />
    <Spinner
        android:id="@+id/spinner"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:prompt="@string/planet_prompt"
    />
</LinearLayout>
```

Notice that the `TextView`'s `android:text` attribute and the `Spinner`'s `android:prompt` attribute both reference the same string resource. This text behaves as a title for the widget. When applied to the `Spinner`, the title text will appear in the selection dialog that appears upon selecting the widget.

Edit **Resources/Values/Strings.xml** and modify the file to look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">HelloSpinner</string>
  <string name="planet_prompt">Choose a planet</string>
  <string-array name="planets_array">
    <item>Mercury</item>
    <item>Venus</item>
    <item>Earth</item>
    <item>Mars</item>
    <item>Jupiter</item>
    <item>Saturn</item>
    <item>Uranus</item>
    <item>Neptune</item>
  </string-array>
</resources>
```

The second `<string>` element defines the title string referenced by the `TextView` and `Spinner` in the layout above. The `<string-array>` element defines the list of strings that will be displayed as the list in the `Spinner` widget.

Now open **MainActivity.cs** and add the following `using` statement:

```
using System;
```

Next, insert the following code for the `OnCreate()`) method:

```
protected override void OnCreate (Bundle bundle)
{
    base.OnCreate (bundle);

    // Set our view from the "Main" layout resource
    SetContentView (Resource.Layout.Main);

    Spinner spinner = FindViewById<Spinner> (Resource.Id.spinner);

    spinner.ItemSelected += new EventHandler<AdapterView.ItemSelectedEventArgs> (spinner_ItemSelected);
    var adapter = ArrayAdapter.CreateFromResource (
            this, Resource.Array.planets_array, Android.Resource.Layout.SimpleSpinnerItem);

    adapter.SetDropDownViewResource (Android.Resource.Layout.SimpleSpinnerDropDownItem);
    spinner.Adapter = adapter;
}
```

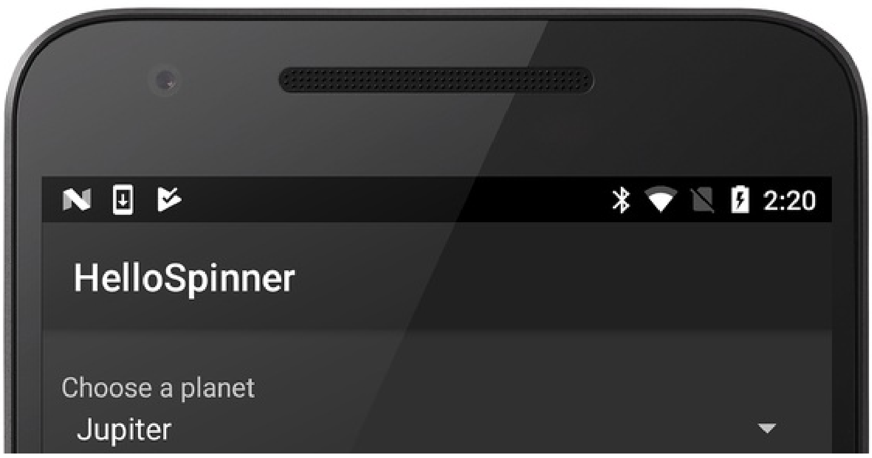
After the `Main.axml` layout is set as the content view, the `Spinner` widget is captured from the layout with `FindViewById<>(int)`. The `CreateFromResource()` method then creates a new `ArrayAdapter`, which binds each item in the string array to the initial appearance for the `Spinner` (which is how each item will appear in the spinner when selected). The `Resource.Array.planets_array` ID references the `string-array` defined above and the `Android.Resource.Layout.SimpleSpinnerItem` ID references a layout for the standard spinner appearance, defined by the platform. `SetDropDownViewResource` is called to define the appearance for each item when the widget is opened. Finally, the `ArrayAdapter` is set to associate all of its items with the `Spinner` by setting the `Adapter` property.

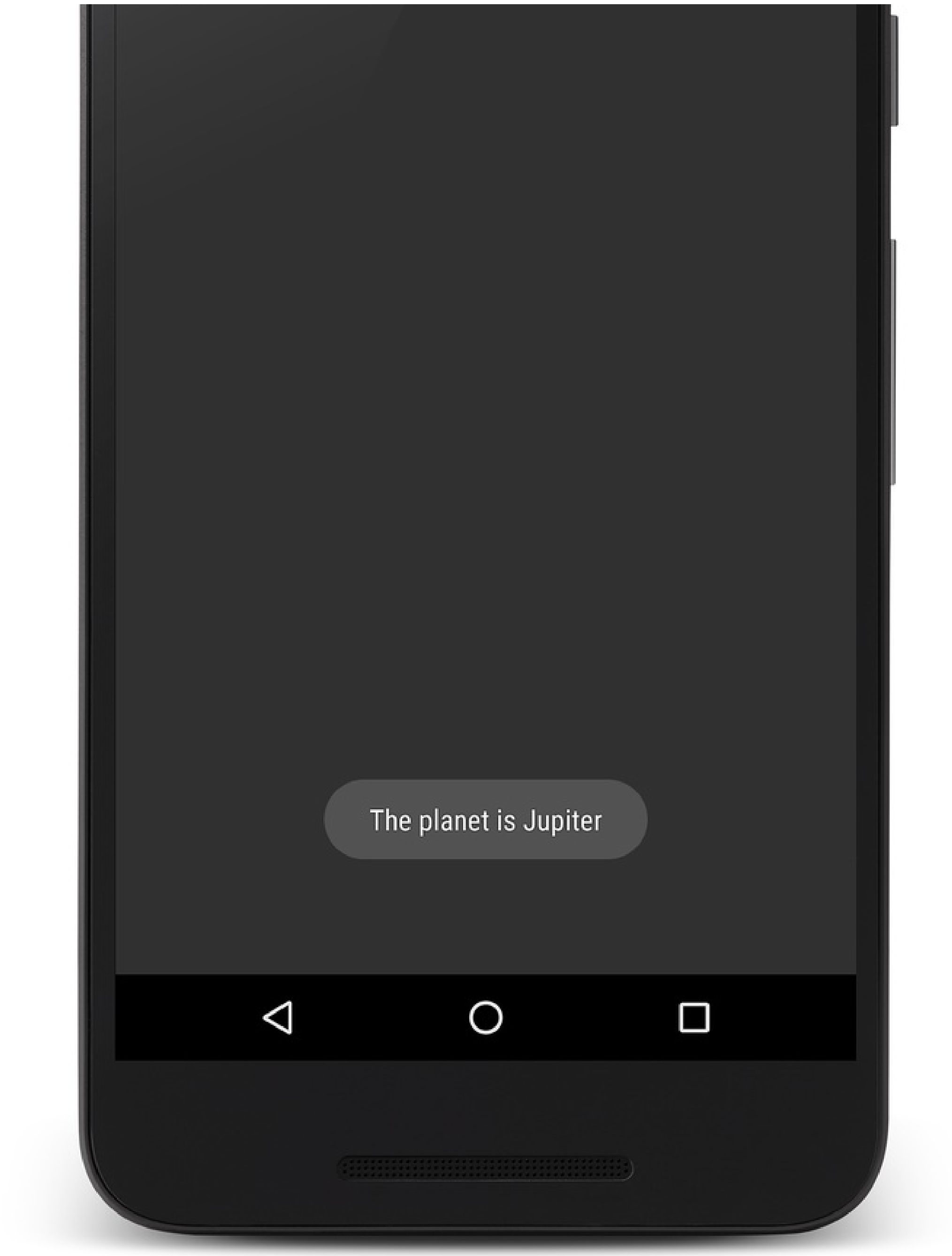
Now provide a callback method that notifys the application when an item has been selected from the `Spinner`. Here's what this method should look like:

```
private void spinner_ItemSelected (object sender, AdapterView.ItemSelectedEventArgs e)
{
    Spinner spinner = (Spinner)sender;
    string toast = string.Format ("The planet is {0}", spinner.GetItemAtPosition (e.Position));
    Toast.MakeText (this, toast, ToastLength.Long).Show ();
}
```

When an item is selected, the sender is cast to a `Spinner` so that items can be accessed. Using the `Position` property on the `ItemEventArgs`, you can find out the text of the selected object, and use it to display a `Toast`.

Run the application; it should look like this:





Spinner Using Key/Value Pairs

Often it is necessary to use `Spinner` to display key values that are associated with some kind of data used by your app. Because `Spinner` does not work directly with key/value pairs, you must store the key/value pair separately, populate the `Spinner` with key values, then use the position of the selected key in the Spinner to look up the associated data value.

In the following steps, the **HelloSpinner** app is modified to display the mean temperature for the selected planet:

Add the following `using` statement to **MainActivity.cs**:

```
using System.Collections.Generic;
```

Add the following instance variable to the `MainActivity` class. This list will hold key/value pairs for the planets and their mean temperatures:

```
private List<KeyValuePair<string, string>> planets;
```

In the `OnCreate` method, add the following code before `adapter` is declared:

```
planets = new List<KeyValuePair<string, string>>
{
    new KeyValuePair<string, string>("Mercury", "167 degrees C"),
    new KeyValuePair<string, string>("Venus", "464 degrees C"),
    new KeyValuePair<string, string>("Earth", "15 degrees C"),
    new KeyValuePair<string, string>("Mars", "-65 degrees C"),
    new KeyValuePair<string, string>("Jupiter" , "-110 degrees C"),
    new KeyValuePair<string, string>("Saturn", "-140 degrees C"),
    new KeyValuePair<string, string>("Uranus", "-195 degrees C"),
    new KeyValuePair<string, string>("Neptune", "-200 degrees C")
};
```

This code creates a simple store for planets and their associated mean temperatures. (In a real-world app, a database is typically used to store keys and their associated data.)

Immediately after the above code, add the following lines to extract the keys and put them into a list (in order):

```
List<string> planetNames = new List<string>();
foreach (var item in planets)
    planetNames.Add (item.Key);
```

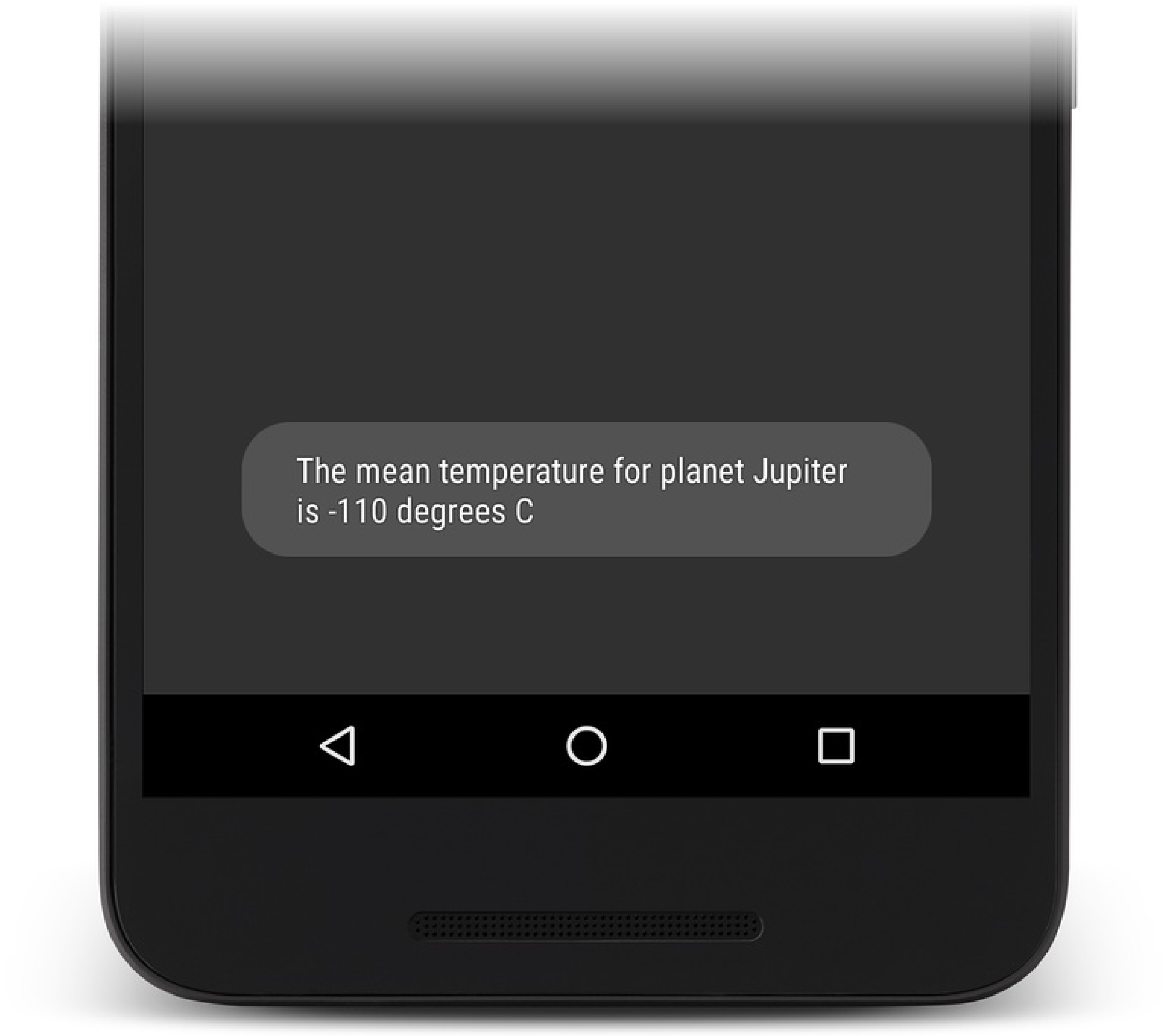
Pass this list to the `ArrayAdapter` constructor (instead of the `planets_array` resource):

```
var adapter = new ArrayAdapter<string>(this,
    Android.Resource.Layout.SimpleSpinnerItem, planetNames);
```

Modify `spinner_ItemSelected` so that the selected position is used to look up the value (the temperature) associated with the selected planet:

```
private void spinner_ItemSelected(object sender, AdapterView.ItemSelectedEventArgs e)
{
    Spinner spinner = (Spinner)sender;
    string toast = string.Format("The mean temperature for planet {0} is {1}",
        spinner.GetItemAtPosition(e.Position), planets[e.Position].Value);
    Toast.MakeText(this, toast, ToastLength.Long).Show();
}
```

Run the application; the toast should look like this:



Resources

- `Resource.Layout`
- `ArrayAdapter`
- `Spinner`

Xamarin.Android Switch

7/8/2021 • 2 minutes to read • Edit Online

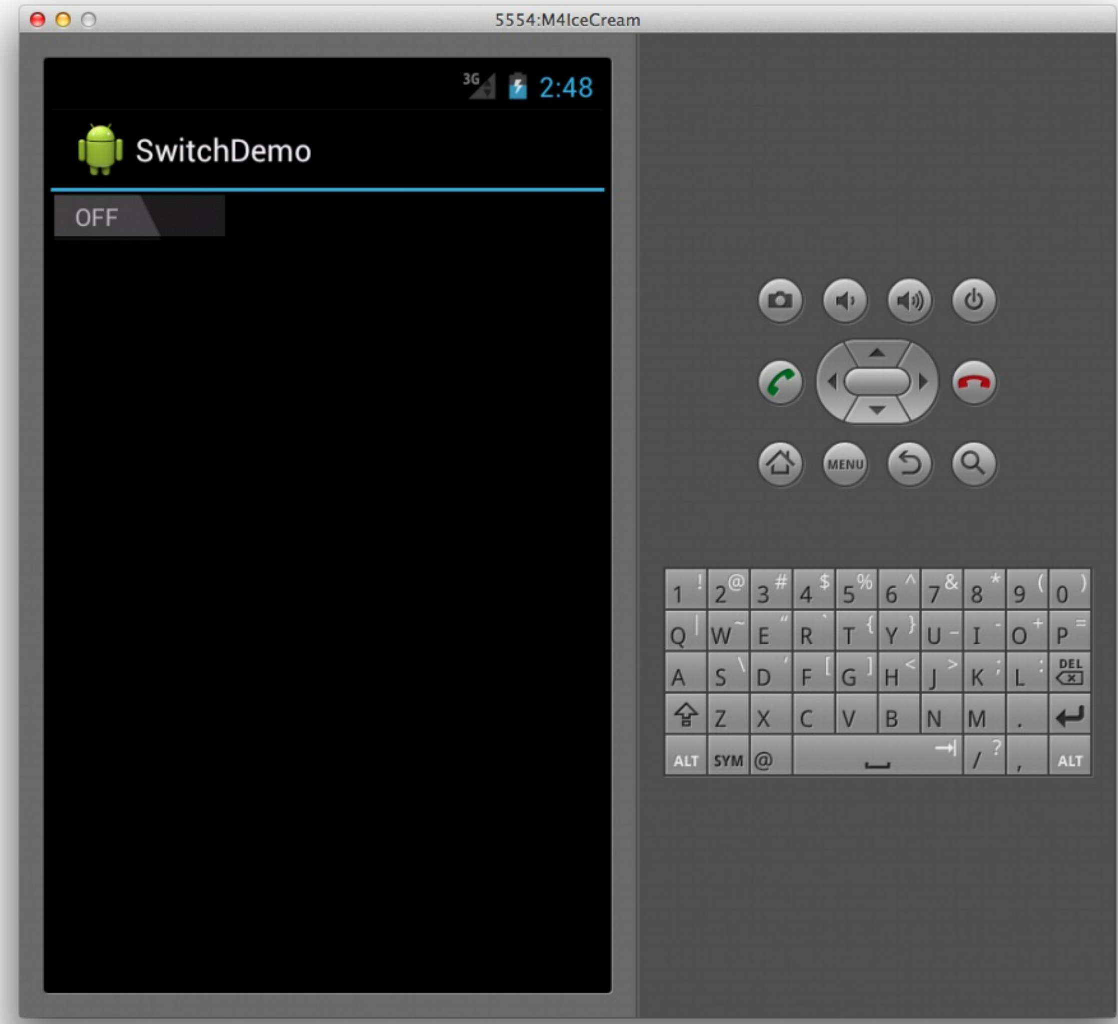
The `Switch` widget (shown below) allows a user to toggle between two states, such as ON or OFF. The `Switch` default value is OFF. The widget is shown below in both its ON and OFF states:

Creating a Switch

To create a switch, simply declare a `Switch` element in XML as follows:

```
<Switch android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
```

This creates a basic switch as shown below:



Changing Default Values

Both the text that the control displays for the ON and OFF states and the default value are configurable. For example, to make the Switch default to ON and read NO/YES instead of OFF/ON, we can set the `checked`, `textOn`, and `textOff` attributes in the following XML.

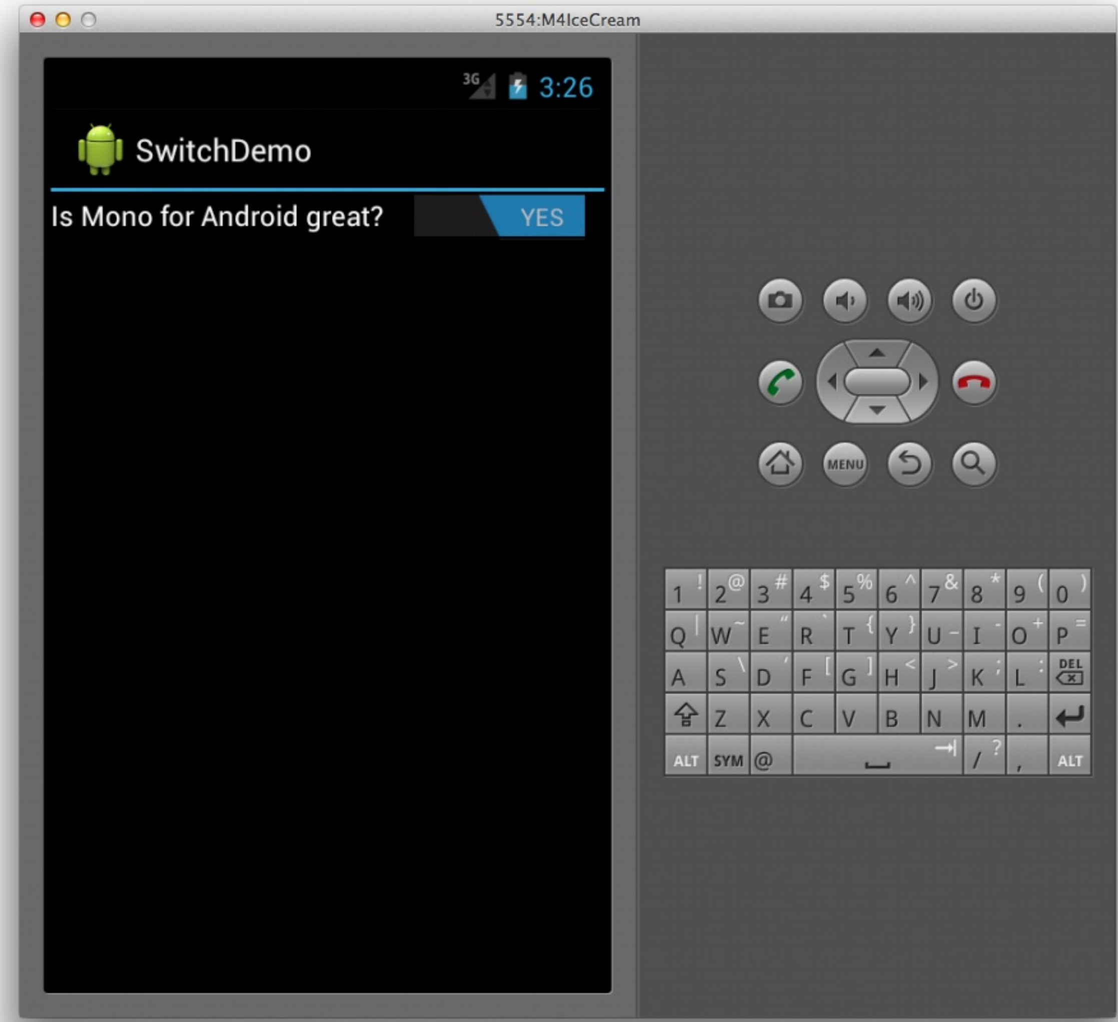
```
<Switch android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:checked="true"
        android:textOn="YES"
        android:textOff="NO" />
```

Providing a Title

The `Switch` widget also supports including a text label by setting the `text` attribute as follows:

```
<Switch android:text="Is Xamarin.Android great?"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:checked="true"
        android:textOn="YES"
        android:textOff="NO" />
```

This markup produces the following screenshot at runtime:



When a `Switch`'s value changes, it raises a `CheckedChange` event. For example, in the following code we capture this event and present a `Toast` widget with a message based upon the `isChecked` value of `Switch`, which is passed to the event handler as part of the `CompoundButton.CheckedChangeEventArg` argument.

```
Switch s = FindViewById<Switch> (Resource.Id.monitored_switch);

s.CheckedChange += delegate(object sender, CompoundButton.CheckedChangeEventArgs e) {
    var toast = Toast.MakeText (this, "Your answer is " +
        (e.IsChecked ?  "correct" : "incorrect"), ToastLength.Short);
    toast.Show ();
};
```

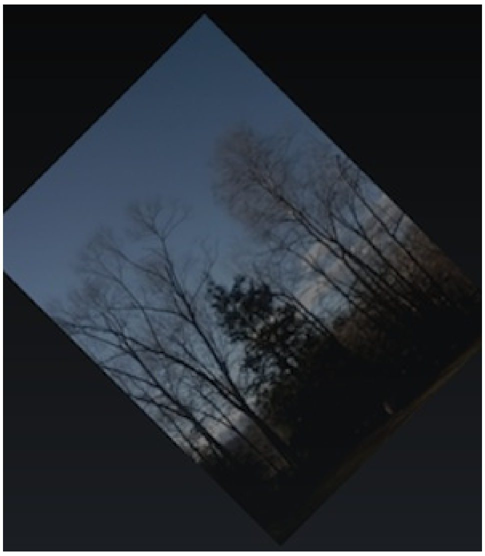
Related Links

- SwitchDemo (sample)
- Tab Layout Tutorial

Xamarin.Android TextureView

7/8/2021 • 2 minutes to read • Edit Online

The `TextureView` class is a view that uses hardware-accelerated 2D rendering to enable a video or OpenGL content stream to be displayed. For example, the following screenshot shows the `TextureView` displaying a live feed from the device's camera:



Unlike the `SurfaceView` class, which can also be used to display OpenGL or video content, the TextureView is not rendered into a separate window. Therefore, `TextureView` is able to support view transformations like any other view. For example, rotating a `TextureView` can be accomplished by simply setting its `Rotation` property, its transparency by setting its `Alpha` property, and so on.

Therefore, with the `TextureView` we can now do things like display a live stream from the camera and transform it, as shown in the following code:

```
public class TextureViewActivity : Activity,
    TextureView.ISurfaceTextureListener
{
    Camera _camera;
    TextureView _textureView;

    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);
        _textureView = new TextureView (this);
        _textureView.SurfaceTextureListener = this;

        SetContentView (_textureView);
    }

    public void OnSurfaceTextureAvailable (
        Android.Graphics.SurfaceTexture surface,
        int width, int height)
    {
        _camera = Camera.Open ();
        var previewSize = _camera.GetParameters ().PreviewSize;
        _textureView.LayoutParameters =
            new FrameLayout.LayoutParams (previewSize.Width,
                previewSize.Height, (int)GravityFlags.Center);

        try {
            _camera.SetPreviewTexture (surface);
            _camera.StartPreview ();
        } catch (Java.IO.IOException ex) {
            Console.WriteLine (ex.Message);
        }

        // this is the sort of thing TextureView enables
        _textureView.Rotation = 45.0f;
        _textureView.Alpha = 0.5f;
    }
    …
}
```

The above code creates a `TextureView` instance in the Activity's `OnCreate` method and sets the Activity as the `TextureView`'s `SurfaceTextureListener`. To be the `SurfaceTextureListener`, the Activity implements the `TextureView.ISurfaceTextureListener` interface. The system will call the `OnSurfaceTextAvailable` method when the `SurfaceTexture` is ready for use. In this method, we take the `SurfaceTexture` that is passed in and set it to the camera's preview texture. Then we are free to perform normal view-based operations, such as setting the `Rotation` and `Alpha`, as in the example above. The resulting application, running on a device, is shown below:

To use the `TextureView`, hardware acceleration must be enabled, which it will be by default as of API Level 14. Also, since this example uses the camera, both the `android.permission.CAMERA` permission and the `android.hardware.camera` feature must be set in the **AndroidManifest.xml**.

Related Links

- TextureViewDemo (sample)/)

Toolbar

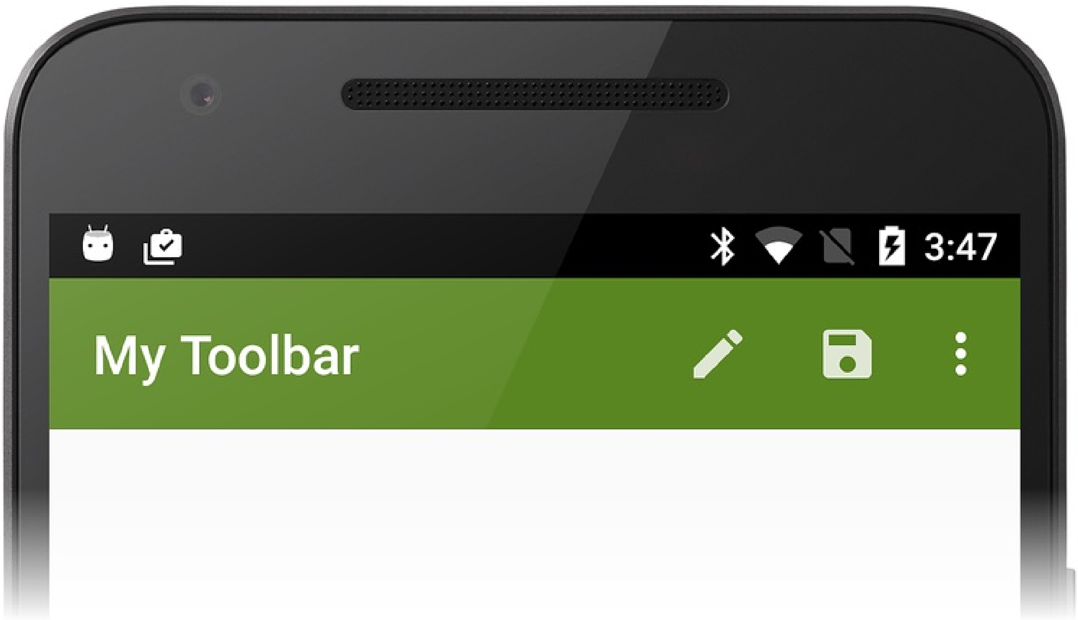
7/8/2021 • 2 minutes to read • Edit Online

The Toolbar is an action bar component that provides more flexibility than the default action bar: it can be placed anywhere in the app, its size can be changed, and it can use a color scheme that is different from the app's theme. Also, each app screen can have multiple Toolbars.

Overview

A key design element of any Android activity is an *action bar*. The action bar is the UI component that is used for navigation, search, menus, and branding in an Android app. In Android versions before Android 5.0 Lollipop, the action bar (also known as the *app bar*) was the recommended component for providing this functionality.

The `Toolbar` widget (introduced in Android 5.0 Lollipop) can be thought of as a generalization of the action bar interface – it is intended to replace the action bar. The `Toolbar` can be used anywhere in an app layout, and it is much more customizable than an action bar. The following screenshot illustrates the customized `Toolbar` example created in this guide:



There are some important differences between the `Toolbar` and the action bar:

- A `Toolbar` can be placed anywhere in the user interface.
- Multiple toolbars can be displayed on the same screen.
- If fragments are used, each fragment can have its own `Toolbar`.
- A `Toolbar` can be configured to span only a partial width of the screen.
- Because the `Toolbar` is not bound to the color scheme of the Activity's window decor, it can have a visually distinct color scheme.
- Unlike the action bar, the `Toolbar` does not include an icon on the left. Its menus on the right use less space.

- The `Toolbar` height is adjustable.
- Other views can be included inside the `Toolbar`.

A `Toolbar` can contain one or more of the following elements:

- Navigation button
- A branded logo image
- Title and subtitle
- Custom views
- Action menu
- Overflow menu

Google's Material Design guidelines recommends taking advantage of these elements to give apps a distinct look (rather than relying solely on an application icon and title).

This guide covers the most commonly-used `Toolbar` scenarios:

- Replacing an Activity's default action bar with a `Toolbar`.
- Adding a second `Toolbar` to an Activity.
- Using the **Android Support Library v7 AppCompat** library (referred to as *AppCompat* in the rest of this guide) to deploy `Toolbar` on earlier versions of Android.

Requirements

`Toolbar` is available on Android 5.0 Lollipop (API 21) and later. When targeting Android releases earlier than Android 5.0, use the Android Support Library v7 AppCompat, which provides backwards-compatible `Toolbar` support in a NuGet package. Toolbar Compatibility explains how to use this library.

Related Links

- Lollipop Toolbar (sample)
- AppCompat Toolbar (sample)

Replacing the Action Bar

7/8/2021 • 7 minutes to read • Edit Online

Overview

One of the most common uses for the `Toolbar` is to replace the default action bar with a custom `Toolbar` (when a new Android project is created, it uses the default action bar). Because the `Toolbar` provides the ability to add branded logos, titles, menu items, navigation buttons, and even custom views to the app bar section of an Activity's UI, it offers a significant upgrade over the default action bar.

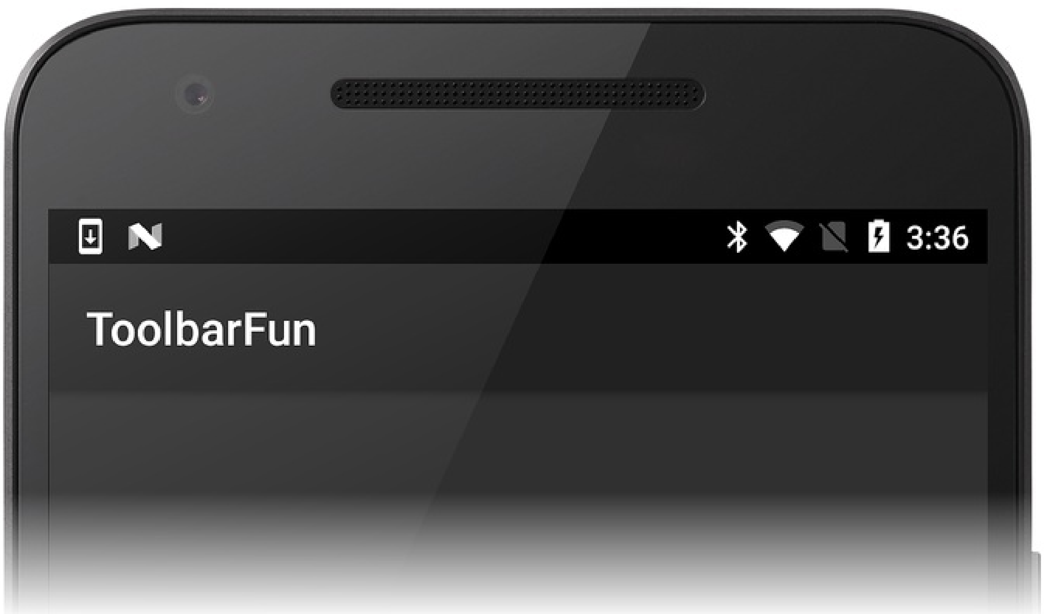
To replace an app's default action bar with a `Toolbar`:

1. Create a new custom theme and modify the app's properties so that it uses this new theme.
2. Disable the `windowActionBar` attribute in the custom theme and enable the `windowNoTitle` attribute.
3. Define a layout for the `Toolbar`.
4. Include the `Toolbar` layout in the Activity's **Main.axml** layout file.
5. Add code to the Activity's `OnCreate` method to locate the `Toolbar` and call `SetActionBar` to install the `ToolBar` as the action bar.

The following sections explain this process in detail. A simple app is created and its action bar is replaced with a customized `Toolbar`.

Start an App Project

Create a new Android project called **ToolbarFun** (see Hello, Android for more information about creating a new Android project). After this project is created, set the target and minimum Android API levels to **Android 5.0 (API Level 21 - Lollipop)** or later. For more information about setting Android version levels, see Understanding Android API Levels. When the app is built and run, it displays the default action bar as seen in this screenshot:



Create a Custom Theme

Open the **Resources/values** directory and a create a new file called **styles.xml**. Replace its contents with the following XML:

```
<?xml version="1.0" encoding="utf-8" ?>
<resources>
  <style name="MyTheme" parent="@android:style/Theme.Material.Light.DarkActionBar">
    <item name="android:windowNoTitle">true</item>
    <item name="android:windowActionBar">false</item>
    <item name="android:colorPrimary">#5A8622</item>
  </style>
</resources>
```

This XML defines a new custom theme called **MyTheme** that is based on the **Theme.Material.Light.DarkActionBar** theme in Lollipop. The `windowNoTitle` attribute is set to `true` to hide the title bar:

```
<item name="android:windowNoTitle">true</item>
```

To display the custom toolbar, the default `ActionBar` must be disabled:

```
<item name="android:windowActionBar">false</item>
```

An olive-green `colorPrimary` setting is used for the background color of the toolbar:

```
<item name="android:colorPrimary">#5A8622</item>
```

Apply the Custom Theme

Edit **Properties/AndroidManifest.xml** and add the following `android:theme` attribute to the `<application>` element so that the app uses the `MyTheme` custom theme:

```
<application android:label="@string/app_name" android:theme="@style/MyTheme"></application>
```

For more information about applying a custom theme to an app, see Using Custom Themes.

Define a Toolbar Layout

In the **Resources/layout** directory, create a new file called **toolbar.xml**. Replace its contents with the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<Toolbar xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:minHeight="?android:attr/actionBarSize"
    android:background="?android:attr/colorPrimary"
    android:theme="@android:style/ThemeOverlay.Material.Dark.ActionBar"/>
```

This XML defines the custom `Toolbar` that replaces the default action bar. The minimum height of the `Toolbar` is set to the size of the action bar that it replaces:

```
android:minHeight="?android:attr/actionBarSize"
```

The background color of the `Toolbar` is set to the olive-green color defined earlier in **styles.xml**:

```
android:background="?android:attr/colorPrimary"
```

Beginning with Lollipop, the `android:theme` attribute can be used to style an individual view. The `ThemeOverlay.Material` themes introduced in Lollipop make it possible to overlay the default `Theme.Material` themes, overwriting relevant attributes to make them either light or dark. In this example, the `Toolbar` uses a dark theme so that its contents are light in color:

```
android:theme="@android:style/ThemeOverlay.Material.Dark.ActionBar"
```

This setting is used so that menu items contrast with the darker background color.

Include the Toolbar Layout

Edit the layout file **Resources/layout/Main.axml** and replace its contents with the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <include
        android:id="@+id/toolbar"
        layout="@layout/toolbar" />
</RelativeLayout>
```

This layout includes the `Toolbar` defined in **toolbar.xml** and uses a `RelativeLayout` to specify that the `Toolbar` is to be placed at the very top of the UI (above the button).

Find and Activate the Toolbar

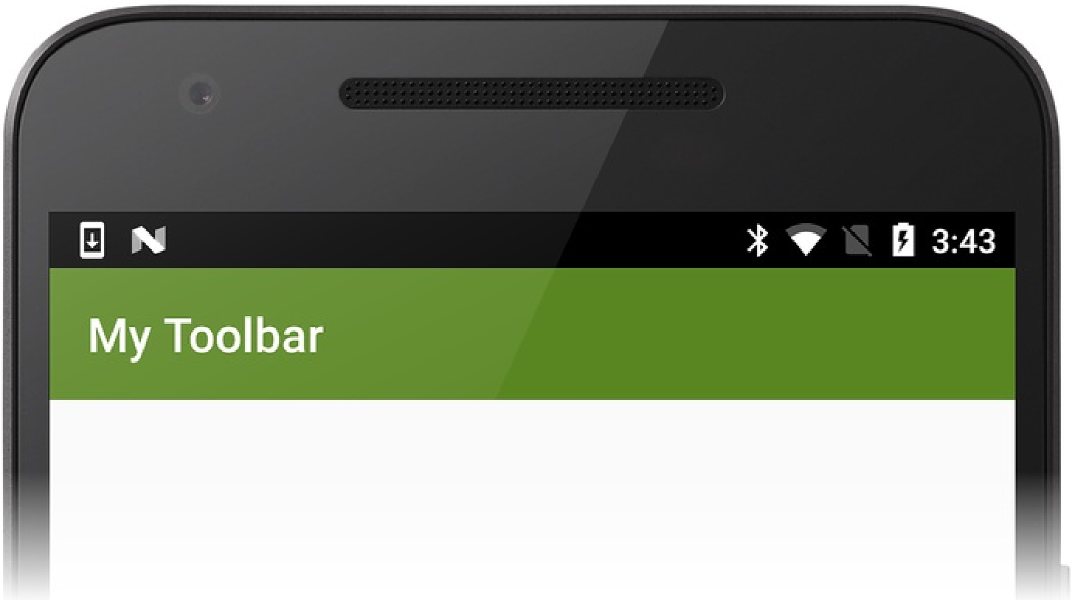
Edit **MainActivity.cs** and add the following using statement:

```
using Android.Views;
```

Also, add the following lines of code to the end of the `OnCreate` method:

```
var toolbar = FindViewById<Toolbar>(Resource.Id.toolbar);
SetActionBar(toolbar);
ActionBar.Title = "My Toolbar";
```

This code finds the `Toolbar` and calls `SetActionBar` so that the `Toolbar` will take on default action bar characteristics. The title of the Toolbar is changed to **My Toolbar**. As seen in this code example, the `ToolBar` can be directly referenced as an action bar. Compile and run this app – the customized `Toolbar` is displayed in place of the default action bar:



Notice that the `Toolbar` is styled independently of the `Theme.Material.Light.DarkActionBar` theme that is applied to the remainder of the app.

If an exception occurs while running the app, see the Troubleshooting section below.

Add Menu Items

In this section, menus are added to the `Toolbar`. The upper right area of the `ToolBar` is reserved for menu items – each menu item (also called an *action item*) can perform an action within the current activity or it can perform an action on behalf of the entire app.

To add menus to the `Toolbar`:

1. Add menu icons (if required) to the `mipmap-` folders of the app project. Google provides a set of free menu icons on the Material icons page.

2. Define the contents of the menu items by adding a new menu resource file under **Resources/menu**.

3. Implement the `OnCreateOptionsMenu` method of the Activity – this method inflates the menu items.

4. Implement the `OnOptionsItemSelected` method of the Activity – this method performs an action when a menu item is tapped.

The following sections demonstrate this process in detail by adding **Edit** and **Save** menu items to the customized `Toolbar`.

Install Menu Icons

Continuing with the `ToolbarFun` example app, add menu icons to the app project. Download toolbar icons, unzip, and copy the contents of the extracted *mipmap-* folders to the project *mipmap-* folders under **ToolbarFun/Resources** and include each added icon file in the project.

Define a Menu Resource

Create a new **menu** subdirectory under **Resources**. In the **menu** subdirectory, create a new menu resource file called **top_menus.xml** and replace its contents with the following XML:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
       android:id="@+id/menu_edit"
       android:icon="@mipmap/ic_action_content_create"
       android:showAsAction="ifRoom"
       android:title="Edit" />
  <item
       android:id="@+id/menu_save"
       android:icon="@mipmap/ic_action_content_save"
       android:showAsAction="ifRoom"
       android:title="Save" />
  <item
       android:id="@+id/menu_preferences"
       android:showAsAction="never"
       android:title="Preferences" />
</menu>
```

This XML creates three menu items:

- An **Edit** menu item that uses the `ic_action_content_create.png` icon (a pencil).
- A **Save** menu item that uses the `ic_action_content_save.png` icon (a diskette).
- A **Preferences** menu item that does not have an icon.

The `showAsAction` attributes of the **Edit** and **Save** menu items are set to `ifRoom` – this setting causes these menu items to appear in the `Toolbar` if there is sufficient room for them to be displayed. The **Preferences** menu item sets `showAsAction` to `never` – this causes the **Preferences** menu to appear in the *overflow* menu (three vertical dots).

Implement OnCreateOptionsMenu

Add the following method to **MainActivity.cs**:

```
public override bool OnCreateOptionsMenu(IMenu menu)
{
    MenuInflater.Inflate(Resource.Menu.top_menus, menu);
    return base.OnCreateOptionsMenu(menu);
}
```

Android calls the `OnCreateOptionsMenu` method so that the app can specify the menu resource for an activity. In this method, the **top_menus.xml** resource is inflated into the passed `menu`. This code causes the new **Edit**, **Save**, and **Preferences** menu items to appear in the `Toolbar`.

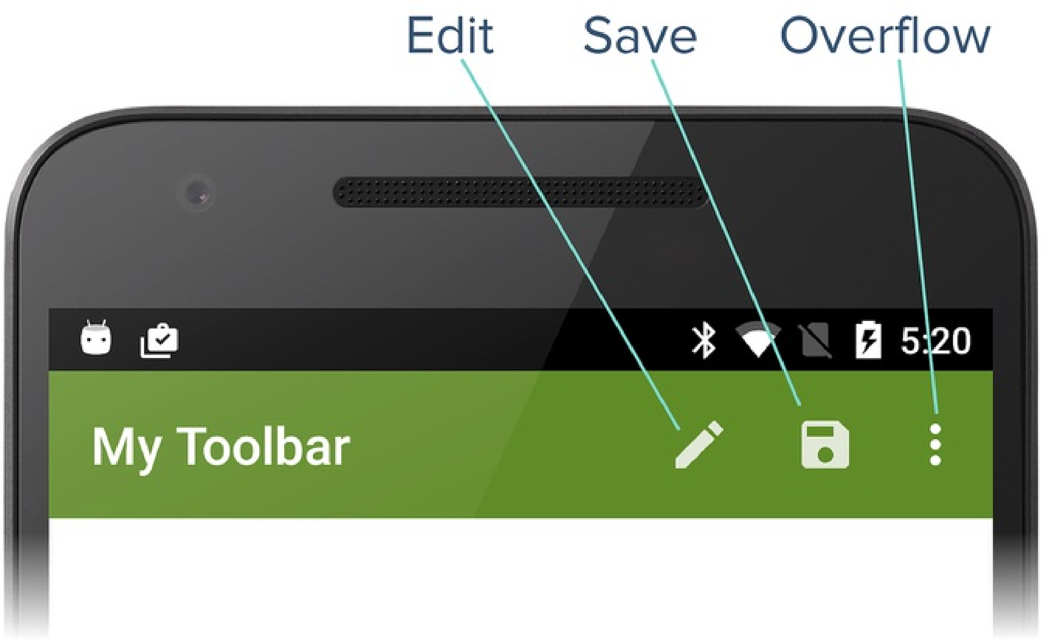
Implement OnOptionsItemSelected

Add the following method to **MainActivity.cs**:

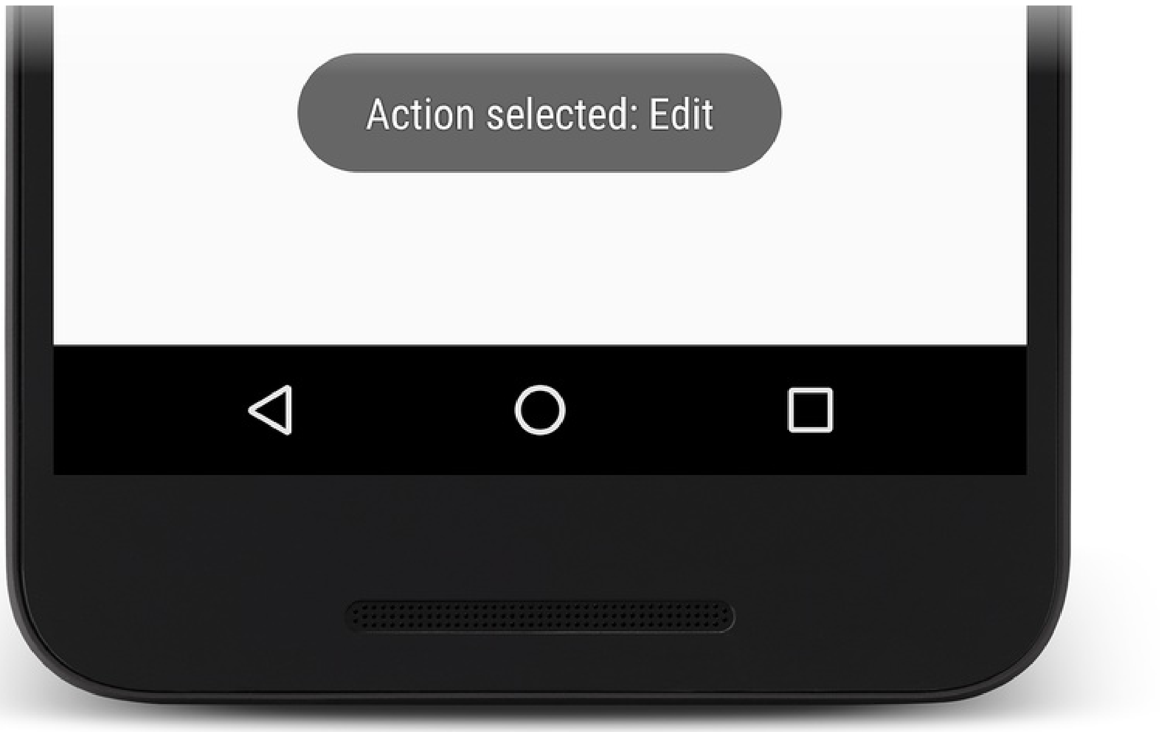
```
public override bool OnOptionsItemSelected(IMenuItem item)
{
    Toast.MakeText(this, "Action selected: " + item.TitleFormatted,
        ToastLength.Short).Show();
    return base.OnOptionsItemSelected(item);
}
```

When a user taps a menu item, Android calls the `OnOptionsItemSelected` method and passes in the menu item that was selected. In this example, the implementation just displays a toast to indicate which menu item was tapped.

Build and run `ToolbarFun` to see the new menu items in the toolbar. The `Toolbar` now displays three menu icons as seen in this screenshot:



When a user taps the **Edit** menu item, a toast is displayed to indicate that the `OnOptionsItemSelected` method was called:



When a user taps the overflow menu, the **Preferences** menu item is displayed. Typically, less-common actions should be placed in the overflow menu – this example uses the overflow menu for **Preferences** because it is not used as often as **Edit** and **Save**:

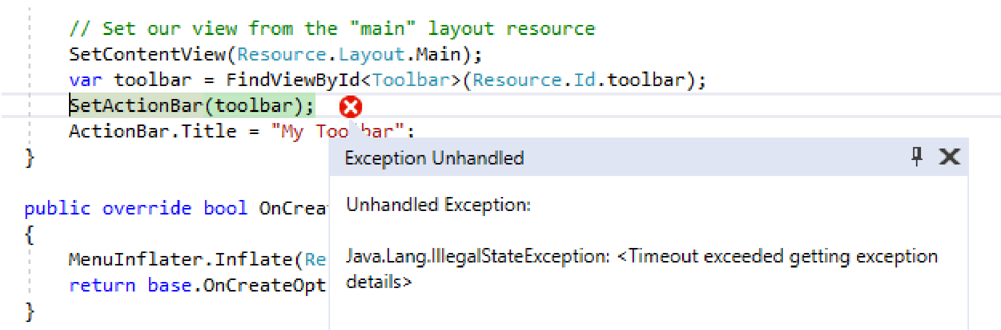
For more information about Android menus, see the Android Developer Menus topic.

Troubleshooting

The following tips can help to debug problems that may occur while replacing the action bar with a toolbar.

Activity Already Has an Action Bar

If the app is not properly configured to use a custom theme as explained in Apply the Custom Theme, the following exception may occur while running the app:



In addition, an error message such as the following may be produced: *Java.Lang.IllegalStateException: This Activity already has an action bar supplied by the window decor.*

To correct this error, verify that the `android:theme` attribute for the custom theme is added to `<application>` (in **Properties/AndroidManifest.xml**) as described earlier in Apply the Custom Theme. In addition, this error may be caused if the `Toolbar` layout or custom theme is not configured properly.

Related Links

- Lollipop Toolbar (sample)
- AppCompat Toolbar (sample)

Adding a Second Toolbar

7/8/2021 • 4 minutes to read • Edit Online

Overview

The `Toolbar` can do more than replace the action bar – it can be used multiple times within an Activity, it can be customized for placement anywhere on the screen, and it can be configured to span only a partial width of the screen. The examples below illustrate how to create a second `Toolbar` and place it at the bottom of the screen. This `Toolbar` implements **Copy**, **Cut**, and **Paste** menu items.

Define the Second Toolbar

Edit the layout file **Main.axml** and replace its contents with with the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <include
        android:id="@+id/toolbar"
        layout="@layout/toolbar" />
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:id="@+id/main_content"
        android:layout_below="@id/toolbar">
      <ImageView
          android:layout_width="fill_parent"
          android:layout_height="0dp"
          android:layout_weight="1" />
      <Toolbar
          android:id="@+id/edit_toolbar"
          android:minHeight="?android:attr/actionBarSize"
          android:background="?android:attr/colorAccent"
          android:theme="@android:style/ThemeOverlay.Material.Dark.ActionBar"
          android:layout_width="match_parent"
          android:layout_height="wrap_content" />
    </LinearLayout>
</RelativeLayout>
```

This XML adds a second `Toolbar` to the bottom of the screen with an empty `ImageView` filling the middle of the screen. The height of this `Toolbar` is set to the height of an action bar:

```
android:minHeight="?android:attr/actionBarSize"
```

The background color of this `Toolbar` is set to an accent color that will be defined next:

```
android:background="?android:attr/colorAccent
```

Notice that this `Toolbar` is based on a different theme (**ThemeOverlay.Material.Dark.ActionBar**) than that used by the `Toolbar` created in Replacing the Action Bar – it isn't bound to the Activity's window decor or to the theme used in the first `Toolbar`.

Edit **Resources/values/styles.xml** and add the following accent color to the style definition:

```
<item name="android:colorAccent">#C7A935</item>
```

This gives the bottom toolbar a dark amber color. Building and running the app displays a blank second toolbar at the bottom of the screen:

Add Edit Menu Items

This section explains how to add edit menu items to the bottom `Toolbar`.

To add menu items to a secondary `Toolbar`:

1. Add menu icons to the `mipmap-` folders of the app project (if required).
2. Define the contents of the menu items by adding an additional menu resource file to **Resources/menu**.
3. In the Activity's `OnCreate` method, find the `Toolbar` (by calling `FindViewById`) and inflate the `Toolbar`'s menus.
4. Implement a click handler in `OnCreate` for the new menu items.

The following sections demonstrate this process in detail: **Cut**, **Copy**, and **Paste** menu items are added to the bottom `Toolbar`.

Define the Edit Menu Resource

In the **Resources/menu** subdirectory, create a new XML file called **edit_menus.xml** and replace the contents with the following XML:

```
<?xml version="1.0" encoding="utf-8" ?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
       android:id="@+id/menu_cut"
       android:icon="@mipmap/ic_menu_cut_holo_dark"
       android:showAsAction="ifRoom"
       android:title="Cut" />
  <item
       android:id="@+id/menu_copy"
       android:icon="@mipmap/ic_menu_copy_holo_dark"
       android:showAsAction="ifRoom"
       android:title="Copy" />
  <item
       android:id="@+id/menu_paste"
       android:icon="@mipmap/ic_menu_paste_holo_dark"
       android:showAsAction="ifRoom"
       android:title="Paste" />
</menu>
```

This XML creates the **Cut**, **Copy**, and **Paste** menu items (using icons that were added to the `mipmap-` folders in Replacing the Action Bar).

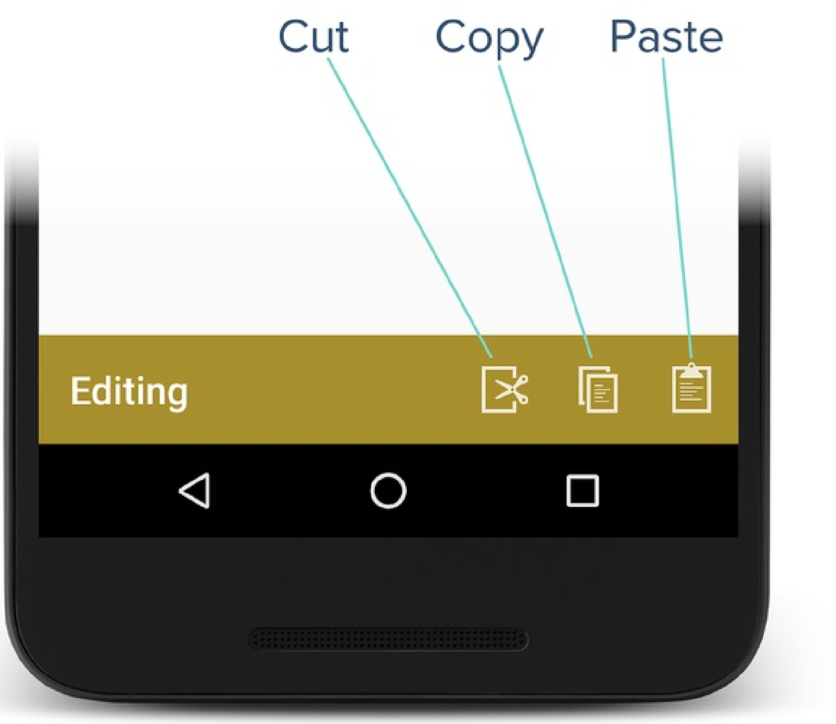
Inflate the Menus

At the end of the `OnCreate` method in **MainActivity.cs**, add the following lines of code:

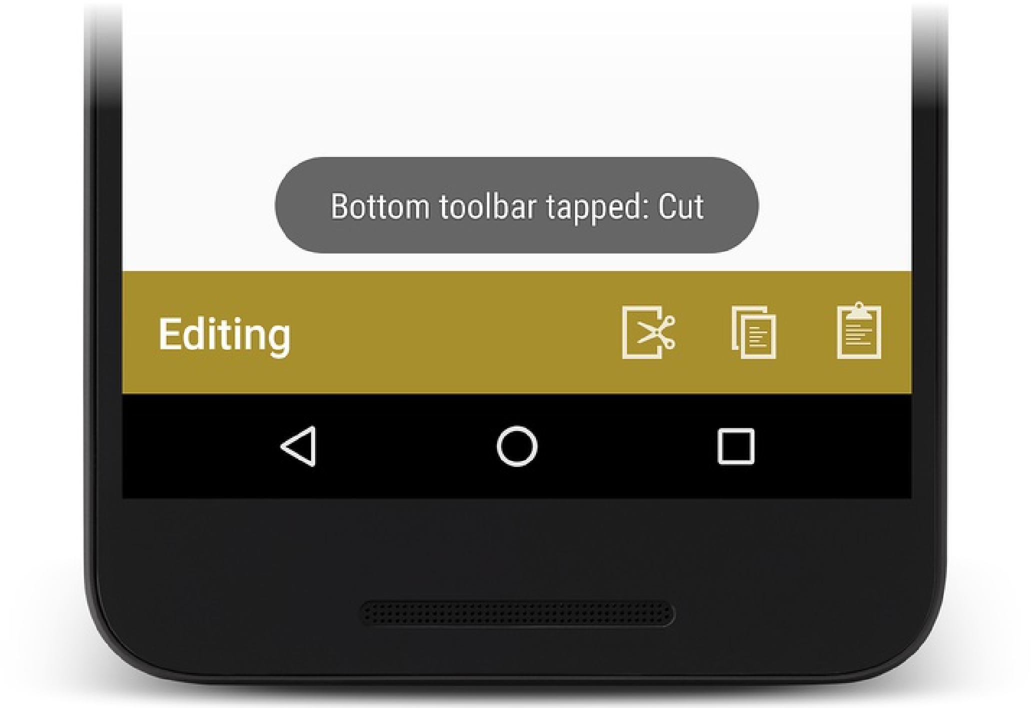
```
var editToolbar = FindViewById<Toolbar>(Resource.Id.edit_toolbar);
editToolbar.Title = "Editing";
editToolbar.InflateMenu (Resource.Menu.edit_menus);
editToolbar.MenuItemClick += (sender, e) => {
    Toast.MakeText(this, "Bottom toolbar tapped: " + e.Item.TitleFormatted, ToastLength.Short).Show();
};
```

This code locates the `edit_toolbar` view defined in **Main.axml**, sets its title to **Editing**, and inflates its menu items (defined in **edit_menus.xml**). It defines a menu click handler that displays a toast to indicate which editing icon was tapped.

Build and run the app. When the app runs, the text and icons added above will appear as shown here:



Tapping the **Cut** menu icon causes the following toast to be displayed:



Tapping menu items on either toolbar displays the resulting toasts:

The Up Button

Most Android apps rely on the **Back** button for app navigation; pressing the **Back** button takes the user to the previous screen. However, you may also want to provide an **Up** button that makes it easy for users to navigate "up" to the app's main screen. When the user selects the **Up** button, the user moves up to a higher level in the app hierarchy – that is, the app pops the user back multiple activities in the back stack rather than popping back to the previously-visited Activity.

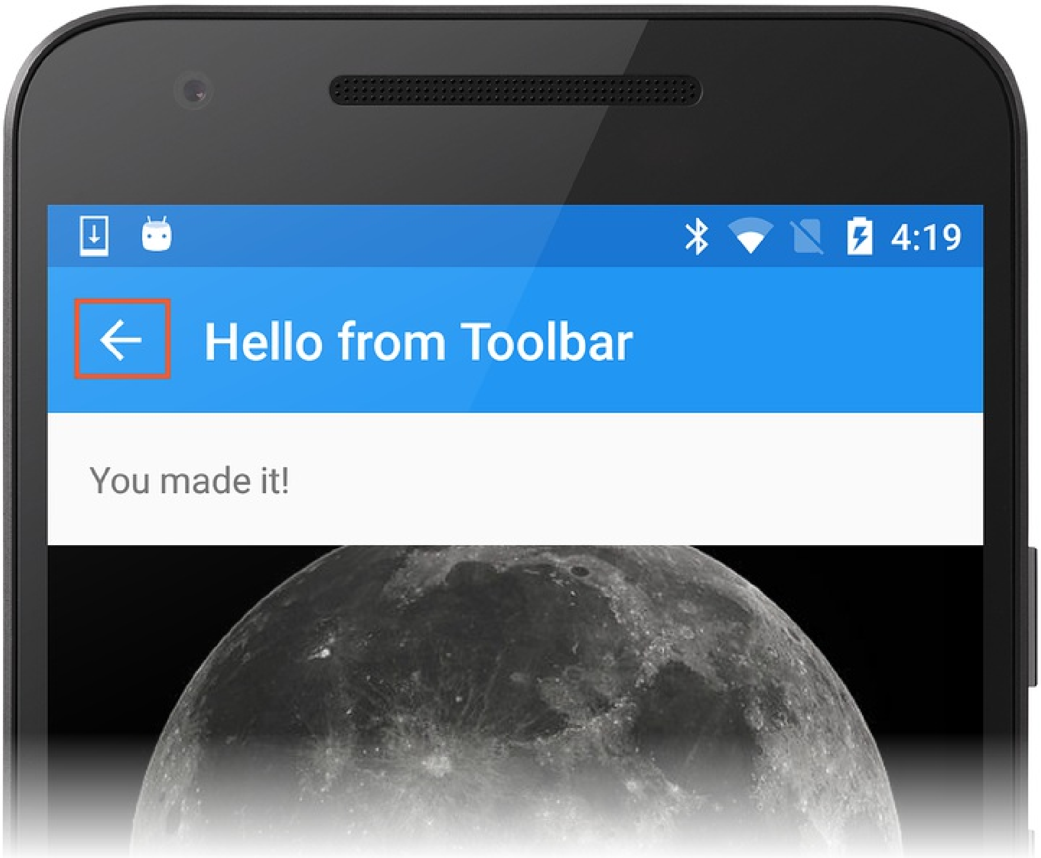
To enable the **Up** button in a second activity that uses a `Toolbar` as its action bar, call the `SetDisplayHomeAsUpEnabled` and `SetHomeButtonEnabled` methods in the second activity's `OnCreate` method:

```
SetActionBar (toolbar);
...
ActionBar.SetDisplayHomeAsUpEnabled (true);
ActionBar.SetHomeButtonEnabled (true);
```

The Support v7 Toolbar code sample demonstrates the **Up** button in action. This sample (which uses the AppCompat library described next) implements a second activity that uses the Toolbar **Up** button for hierarchical navigation back to the previous activity. In this example, the `DetailActivity` home button enables the **Up** button by making the following `SupportActionBar` method calls:

```
SetSupportActionBar (toolbar);
...
SupportActionBar.SetDisplayHomeAsUpEnabled (true);
SupportActionBar.SetHomeButtonEnabled (true);
```

When the user navigates from `MainActivity` to `DetailActivity`, the `DetailActivity` displays an **Up** button (left pointing arrow) as shown in the screenshot:



Tapping this **Up** button causes the app to return to `MainActivity`. In a more complex app with multiple levels of hierarchy, tapping this button would return the user to the next highest level in the app rather than to the previous screen.

Related Links

- Lollipop Toolbar (sample)
- AppCompat Toolbar (sample)

Toolbar Compatibility

7/8/2021 • 5 minutes to read • Edit Online

Overview

This section explains how to use `Toolbar` on versions of Android earlier than Android 5.0 Lollipop. If your app does not support versions of Android earlier than Android 5.0, you can skip this section.

Because `Toolbar` is part of the Android v7 support library, it can be used on devices running Android 2.1 (API level 7) and higher. However, the Android Support Library v7 AppCompat NuGet must be installed and the code modified so that it uses the `Toolbar` implementation provided in this library. This section explains how to install this NuGet and modify the **ToolbarFun** app from Adding a Second Toolbar so that it runs on versions of Android earlier than Lollipop 5.0.

To modify an app to use the AppCompat version of Toolbar:

1. Set the Minimum and Target Android versions for the app.
2. Install the AppCompat NuGet Package.
3. Use an AppCompat theme instead of a built-in Android theme.
4. Modify `MainActivity` so that it subclasses `AppCompatActivity` rather than `Activity`.

Each of these steps is explained in detail in the following sections.

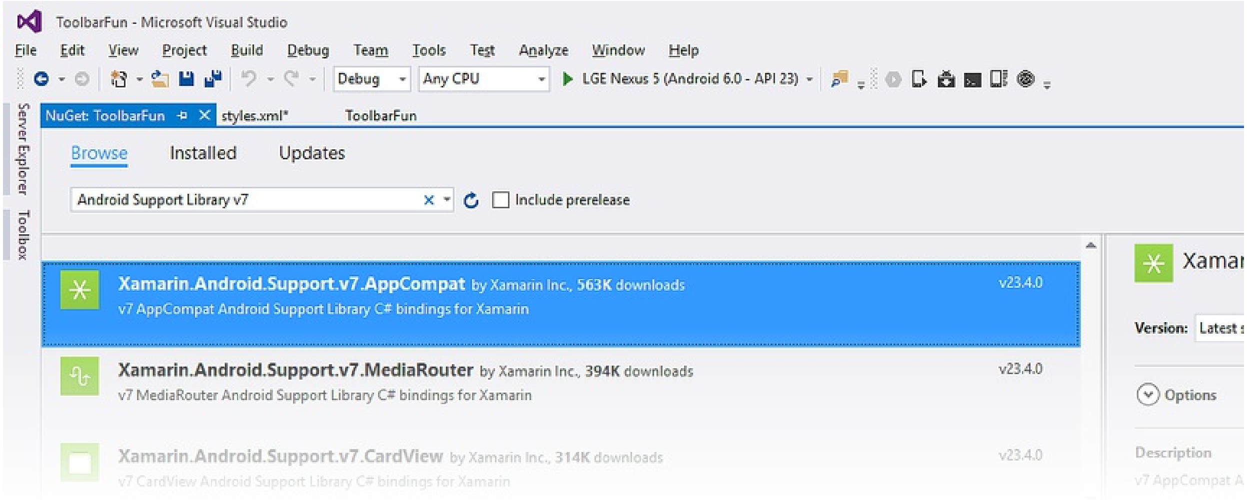
Set the Minimum and Target Android Version

The app's Target Framework must be set to API Level 21 or greater or the app will not deploy properly. If an error such as **No resource identifier found for attribute 'tileModeX' in package 'android'** is seen while deploying the app, this is because the Target Framework is not set to **Android 5.0 (API Level 21 - Lollipop)** or greater.

Set the Target Framework level to API Level 21 or greater and set the Android API level project settings to the minimum Android Version that the app is to support. For more information about setting Android API levels, see Understanding Android API Levels. In the `ToolbarFun` example, the Minimum Android Version is set to KitKat (API Level 4.4).

Install the AppCompat NuGet Package

Next, add the Android Support Library v7 AppCompat package to the project. In Visual Studio, right-click **References** and select **Manage NuGet Packages...**. Click **Browse** and search for **Android Support Library v7 AppCompat**. Select **Xamarin.Android.Support.v7.AppCompat** and click **Install**:



When this NuGet is installed, several other NuGet packages are also installed if not already present (such as **Xamarin.Android.Support.Animated.Vector.Drawable**, **Xamarin.Android.Support.v4**, and **Xamarin.Android.Support.Vector.Drawable**). For more information about installing NuGet packages, see Walkthrough: Including a NuGet in your project.

Use an AppCompat Theme and Toolbar

The AppCompat library comes with several `Theme.AppCompat` themes that can be used on any version of Android supported by the AppCompat library. The `ToolbarFun` example app theme is derived from `Theme.Material.Light.DarkActionBar`, which is not available on Android versions earlier than Lollipop. Therefore, `ToolbarFun` must be adapted to use the AppCompat counterpart for this theme, `Theme.AppCompat.Light.DarkActionBar`. Also, because `Toolbar` is not available on versions of Android earlier than Lollipop, we must use the AppCompat version of `Toolbar`. Therefore, layouts must use `android.support.v7.widget.Toolbar` instead of `Toolbar`.

Update Layouts

Edit **Resources/layout/Main.axml** and replace the `Toolbar` element with the following XML:

```
<android.support.v7.widget.Toolbar
    android:id="@+id/edit_toolbar"
    android:minHeight="?attr/actionBarSize"
    android:background="?attr/colorAccent"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

Edit **Resources/layout/toolbar.xml** and replace its contents with the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.Toolbar xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:minHeight="?attr/actionBarSize"
    android:background="?attr/colorPrimary"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"/>
```

Note that the `?attr` values are no longer prefixed with `android:` (recall that the `?` notation references a resource in the current theme). If `?android:attr` were still used here, Android would reference the attribute value from the currently running platform rather than from the AppCompat library. Because this example uses the `actionBarSize` defined by the AppCompat library, the `android:` prefix is dropped. Similarly, `@android:style` is changed to `@style` so that the `android:theme` attribute is set to a theme in the AppCompat library – the `ThemeOverlay.AppCompat.Dark.ActionBar` theme is used here rather than `ThemeOverlay.Material.Dark.ActionBar`.

Update the Style

Edit **Resources/values/styles.xml** and replace its contents with the following XML:

```
<?xml version="1.0" encoding="utf-8" ?>
<resources>
  <style name="MyTheme" parent="MyTheme.Base"> </style>
  <style name="MyTheme.Base" parent="Theme.AppCompat.Light.DarkActionBar">
    <item name="windowNoTitle">true</item>
    <item name="windowActionBar">false</item>
    <item name="colorPrimary">#5A8622</item>
    <item name="colorAccent">#A88F2D</item>
  </style>
</resources>
```

The item names and parent theme in this example are no longer prefixed with `android:` because we are using the AppCompat library. Also, the parent theme is changed to the AppCompat version of `Light.DarkActionBar`.

Update Menus

To support earlier versions of Android, the AppCompat library uses custom attributes that mirror the attributes of the `android:` namespace. However, some attributes (such as the `showAsAction` attribute used in the `<menu>` tag) do not exist in the Android framework on older devices – `showAsAction` was introduced in Android API 11 but is not available in Android API 7. For this reason, a custom namespace must be used to prefix all of the attributes defined by the support library. In the menu resource files, a namespace called `local` is defined for prefixing the `showAsAction` attribute.

Edit **Resources/menu/top_menus.xml** and replace its contents with the following XML:

```
<?xml version="1.0" encoding="utf-8" ?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:local="http://schemas.android.com/apk/res-auto">
  <item
       android:id="@+id/menu_edit"
       android:icon="@mipmap/ic_action_content_create"
       local:showAsAction="ifRoom"
       android:title="Edit" />
  <item
       android:id="@+id/menu_save"
       android:icon="@mipmap/ic_action_content_save"
       local:showAsAction="ifRoom"
       android:title="Save" />
  <item
       android:id="@+id/menu_preferences"
       local:showAsAction="never"
       android:title="Preferences" />
</menu>
```

The `local` namespace is added with this line:

```
xmlns:local="http://schemas.android.com/apk/res-auto">
```

The `showAsAction` attribute is prefaced with this `local:` namespace rather than `android:`

```
local:showAsAction="ifRoom"
```

Similarly, edit **Resources/menu/edit_menus.xml** and replace its contents with the following XML:

```
<?xml version="1.0" encoding="utf-8" ?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:local="http://schemas.android.com/apk/res-auto">
  <item
       android:id="@+id/menu_cut"
       android:icon="@mipmap/ic_menu_cut_holo_dark"
       local:showAsAction="ifRoom"
       android:title="Cut" />
  <item
       android:id="@+id/menu_copy"
       android:icon="@mipmap/ic_menu_copy_holo_dark"
       local:showAsAction="ifRoom"
       android:title="Copy" />
  <item
       android:id="@+id/menu_paste"
       android:icon="@mipmap/ic_menu_paste_holo_dark"
       local:showAsAction="ifRoom"
       android:title="Paste" />
</menu>
```

How does this namespace switch provide support for the `showAsAction` attribute on Android versions prior to API Level 11? The custom attribute `showAsAction` and all of its possible values are included in the app when the AppCompat NuGet is installed.

Subclass AppCompatActivity

The final step in the conversion is to modify `MainActivity` so that it is a subclass of `AppCompatActivity`. Edit **MainActivity.cs** and add the following `using` statements:

```
using Android.Support.V7.App;
using Toolbar = Android.Support.V7.Widget.Toolbar;
```

This declares `Toolbar` to be the AppCompat version of `Toolbar`. Next, change the class definition of `MainActivity`:

```
public class MainActivity : AppCompatActivity
```

To set the action bar to the AppCompat version of `Toolbar`, substitute the call to `SetActionBar` with `SetSupportActionBar`. In this example, the title is also changed to indicate that the AppCompat version of `Toolbar` is being used:

```
SetSupportActionBar (toolbar);
SupportActionBar.Title = "My AppCompat Toolbar";
```

Finally, change the Minimum Android level to the pre-Lollipop value that is to be supported (for example, API 19).

Build the app and run it on a pre-Lollipop device or Android emulator. The following screenshot shows the AppCompat version of **ToolbarFun** on a Nexus 4 running KitKat (API 19):

When the AppCompat library is used, themes do not have to be switched based on the Android version – the AppCompat library makes it possible to provide a consistent user experience across all supported Android versions.

Related Links

- Lollipop Toolbar (sample)
- AppCompat Toolbar (sample)

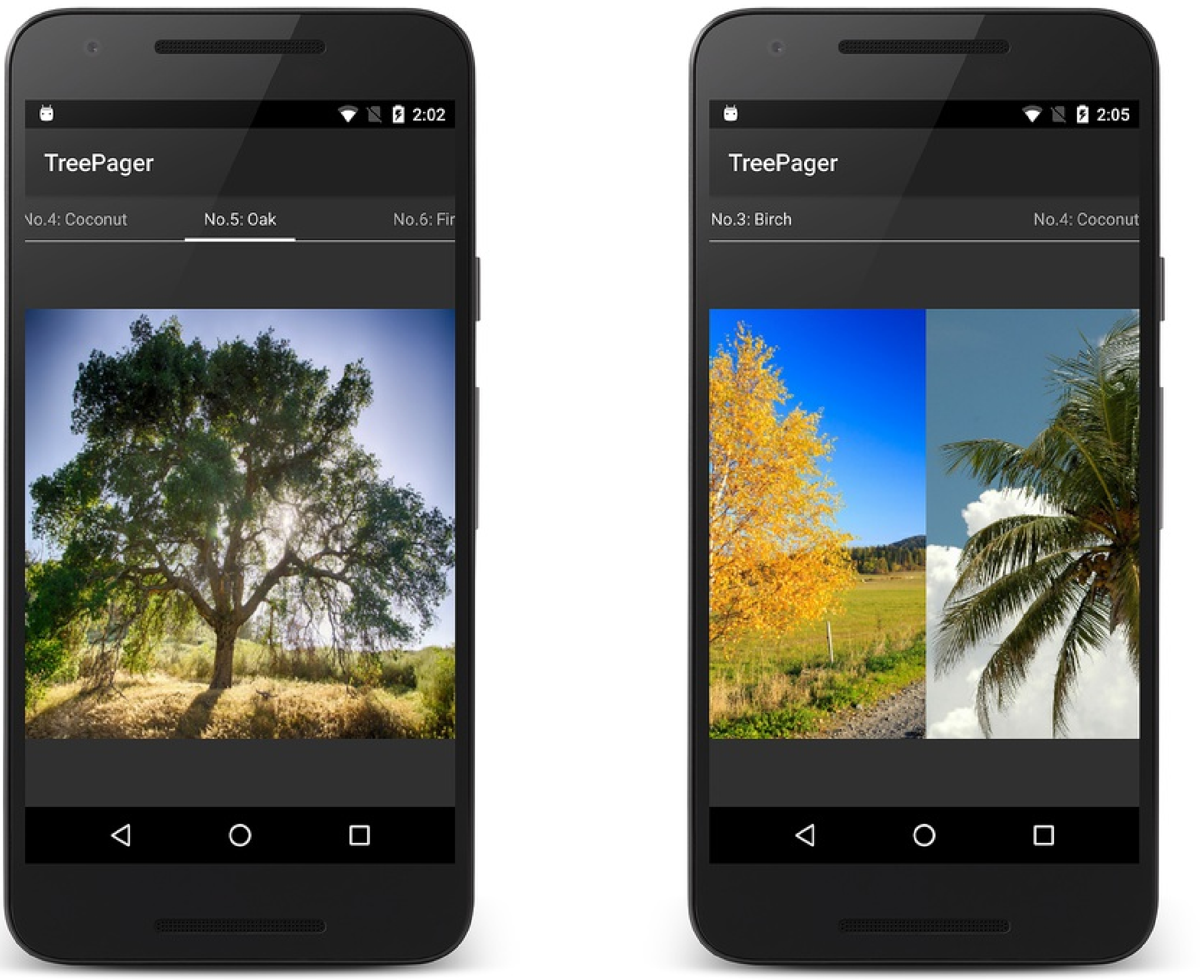
ViewPager

7/8/2021 • 3 minutes to read • Edit Online

ViewPager is a layout manager that lets you implement gestural navigation. Gestural navigation allows the user to swipe left and right to step through pages of data. This guide explains how to implement gestural navigation with ViewPager, with and without Fragments. It also describes how to add page indicators using PagerTitleStrip and PagerTabStrip.

Overview

A common scenario in app development is the need to provide users with gestural navigation between sibling views. In this approach, the user swipes left or right to access pages of content (for example, in a setup wizard or a slide show). You can create these swipe views by using the `ViewPager` widget, available in Android Support Library v4. The `ViewPager` is a layout widget made up of multiple child views where each child view constitutes a page in the layout:



Typically, `ViewPager` is used in conjunction with Fragments; however, there are some situations where you might want to use `ViewPager` without the added complexity of `Fragment` s.

`ViewPager` uses an adapter pattern to provide it with the views to display. The adapter used here is conceptually similar to that used by RecyclerView – you supply an implementation of `PagerAdapter` to generate the pages that the `ViewPager` displays to the user. The pages displayed by `ViewPager` can be `View` s or `Fragment` s. When

`View`s are displayed, the adapter subclasses Android's `PagerAdapter` base class. If `Fragment`s are displayed, the adapter subclasses Android's `FragmentPagerAdapter`. The Android support library also includes `FragmentPagerAdapter` (a subclass of `PagerAdapter`) to help with the details of connecting `Fragment`s to data.

This guide demonstrates both approaches:

- In Viewpager with Views, a TreePager app is developed to demonstrate how to use `ViewPager` to display views of a tree catalog (an image gallery of deciduous and evergreen trees). `PagerTabStrip` and `PagerTitleStrip` are used to display titles that help with page navigation.
- In Viewpager with Fragments, a slightly more complex FlashCardPager app is developed to demonstrate how to use `ViewPager` with `Fragment`s to build an app that presents math problems as flash cards and responds to user input.

Requirements

To use `ViewPager` in your app project, you must install the Android Support Library v4 package. For more information about installing NuGet packages, see Walkthrough: Including a NuGet in your project.

Architecture

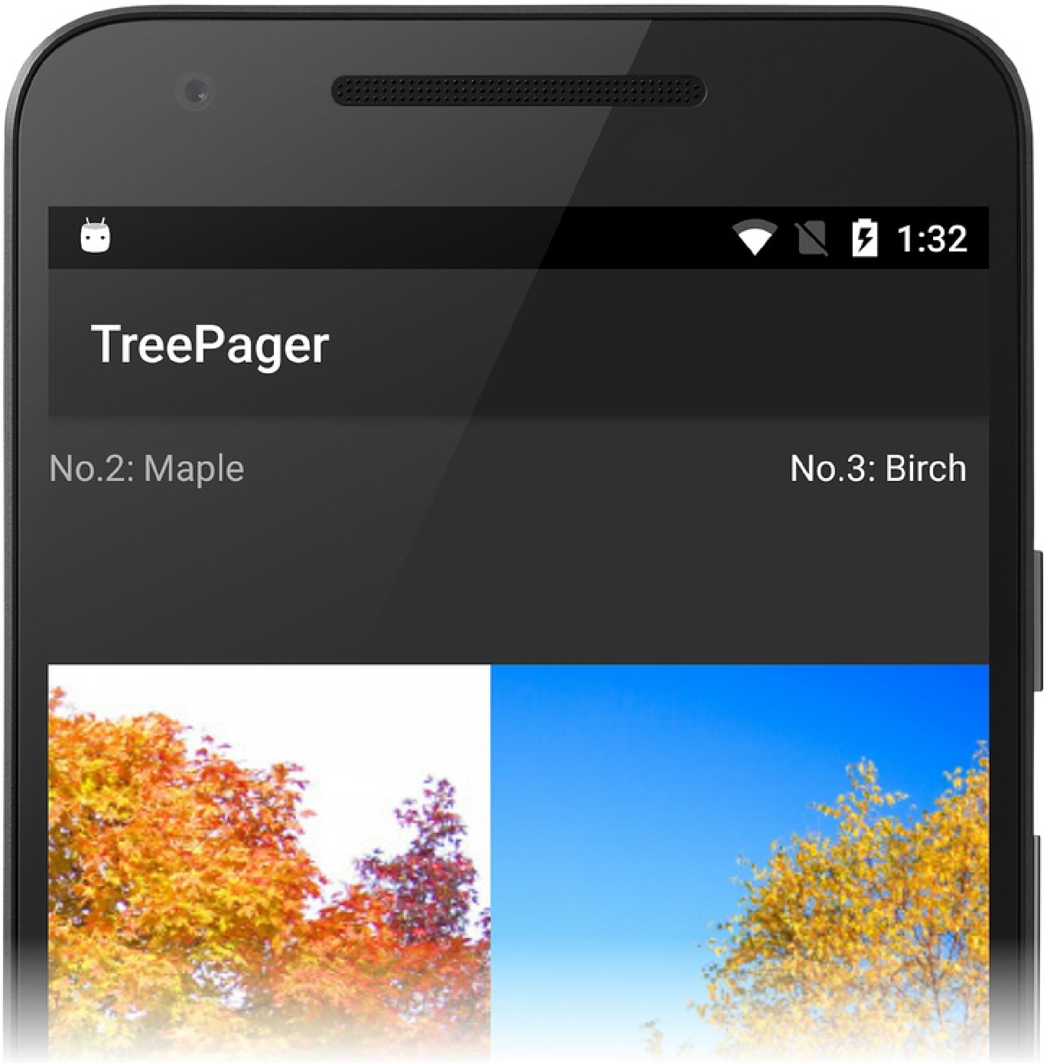
Three components are used for implementing gestural navigation with `ViewPager`:

- ViewPager
- Adapter
- Pager Indicator

Each of these components is summarized below.

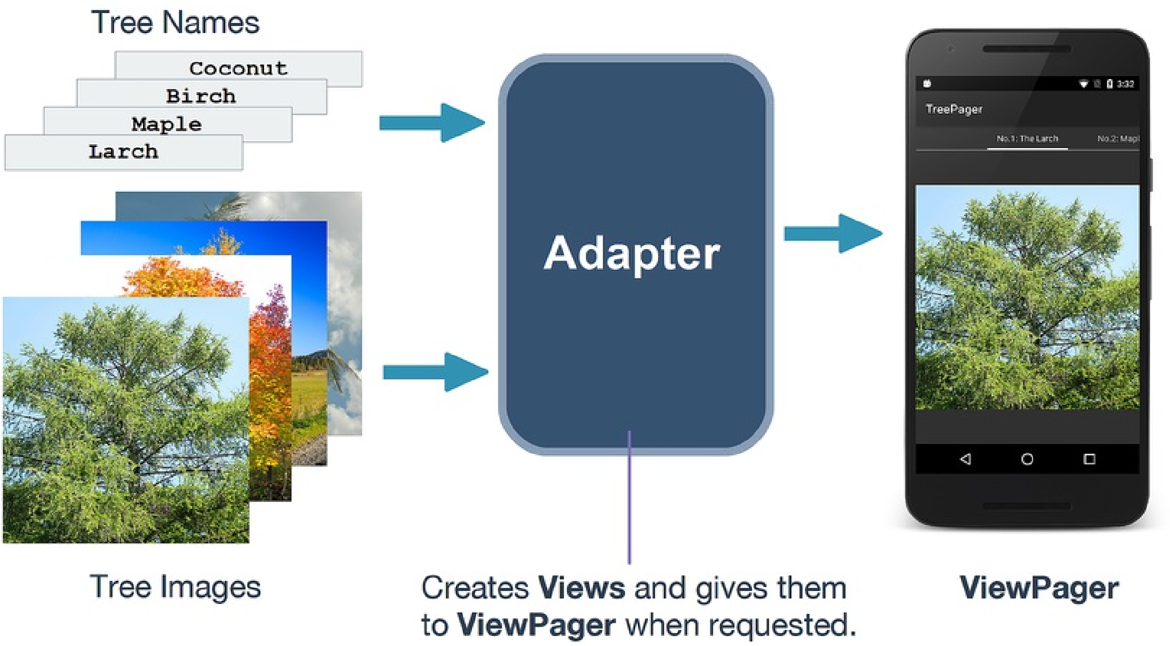
ViewPager

`ViewPager` is a layout manager that displays a collection of `View`s one at a time. Its job is to detect the user's swipe gesture and navigate to the next or previous view as appropriate. For example, the screenshot below demonstrates a `ViewPager` making the transition from one image to the next in response to a user gesture:



Adapter

`ViewPager` pulls its data from an *adapter*. The adapter's job is to create the `View`s displayed by the `ViewPager`, providing them as needed. The diagram below illustrates this concept – the adapter creates and populates `View`s and provides them to the `ViewPager`. As the `ViewPager` detects the user's swipe gestures, it asks the adapter to provide the appropriate `View` to display:

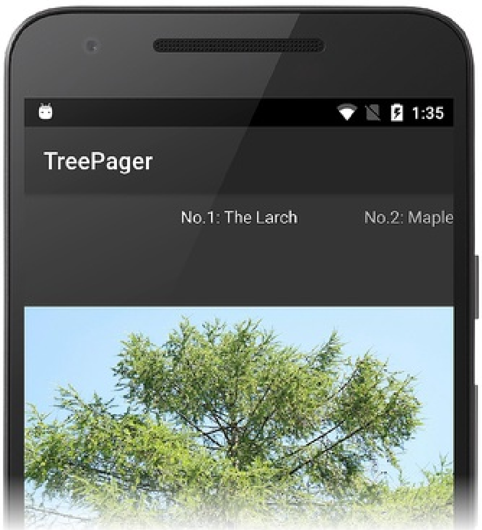


In this particular example, each `View` is constructed from a tree image and a tree name before it is passed to the `ViewPager`.

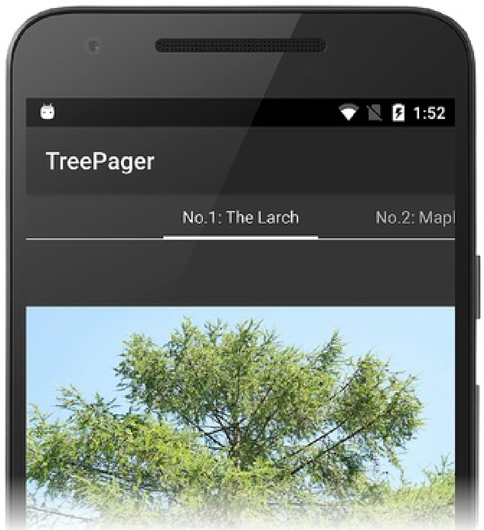
Pager Indicator

`ViewPager` may be used to display a large data set (for example, an image gallery may contain hundreds of images). To help the user navigate large data sets, `ViewPager` is often accompanied by a *pager indicator* that displays a string. The string might be the image title, a caption, or simply the current view's position within the data set.

There are two views that can produce this navigation information for you: `PagerTabStrip` and `PagerTitleStrip.` Each displays a string at the top of a `ViewPager`, and each pulls its data from the `ViewPager`'s adapter so that it always stays in sync with the currently-displayed `View`. The difference between them is that `PagerTabStrip` includes a visual indicator for the "current" string while `PagerTitleStrip` does not (as shown in these screenshots):



PagerTitleStrip



PagerTabStrip

This guide demonstrates how to immplement `ViewPager`, adapter, and indicator app components and integrate them to support gestural navigation.

Related Links

- TreePager (sample)
- FlashCardPager (sample)

ViewPager with Views

7/8/2021 • 10 minutes to read • Edit Online

ViewPager is a layout manager that lets you implement gestural navigation. Gestural navigation allows the user to swipe left and right to step through pages of data. This guide explains how to implement a swipeable UI with ViewPager and PagerTabStrip, using Views as the data pages (a subsequent guide covers how to use Fragments for the pages).

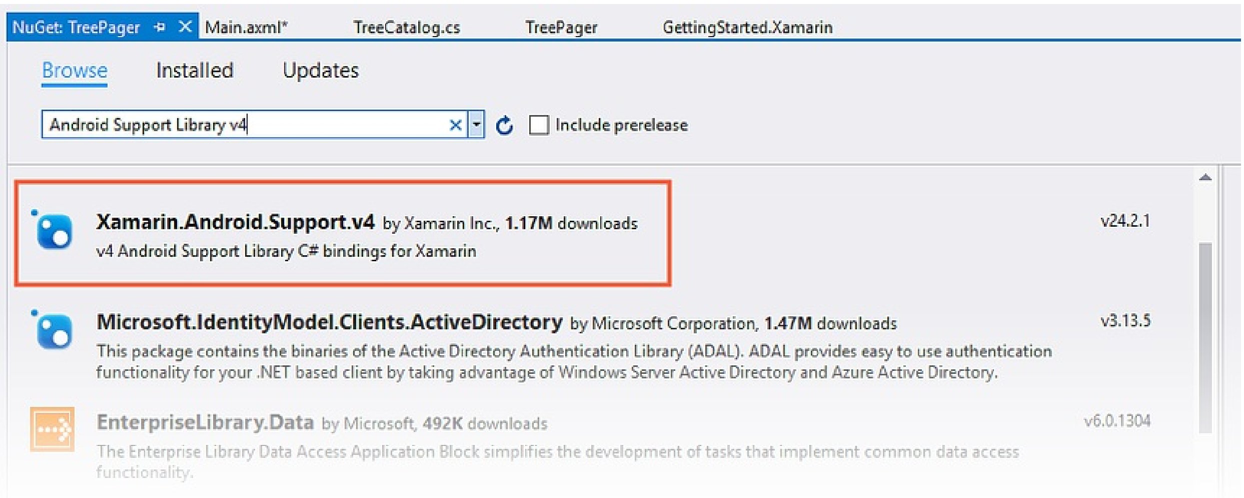
Overview

This guide is a walkthrough that provides a step-by-step demonstration how to use `ViewPager` to implement an image gallery of deciduous and evergreen trees. In this app, the user swipes left and right through a "tree catalog" to view tree images. At the top of each page of the catalog, the name of the tree is listed in a `PagerTabStrip`, and an image of the tree is displayed in an `ImageView`. An adapter is used to interface the `ViewPager` to the underlying data model. This app implements an adapter derived from `PagerAdapter`.

Although `ViewPager`-based apps are often implemented with `Fragment`s, there are some relatively simple use cases where the extra complexity of `Fragment`s is not necessary. For example, the basic image gallery app illustrated in this walkthrough does not require the use of `Fragment`s. Because the content is static and the user only swipes back and forth between different images, the implementation can be kept simpler by using standard Android views and layouts.

Start an App Project

Create a new Android project called **TreePager** (see Hello, Android for more information about creating new Android projects). Next, launch the NuGet Package Manager. (For more information about installing NuGet packages, see Walkthrough: Including a NuGet in your project). Find and install **Android Support Library v4**:



This will also install any additional packages reaquired by **Android Support Library v4**.

Add an Example Data Source

In this example, the tree catalog data source (represented by the `TreeCatalog` class) supplies the `ViewPager` with item content. `TreeCatalog` contains a ready-made collection of tree images and tree titles that the adapter will use for creating `View`s. The `TreeCatalog` constructor requires no arguments:

```
TreeCatalog treeCatalog = new TreeCatalog();
```

The collection of images in `TreeCatalog` is organized such that each image can be accessed by an indexer. For example, the following line of code retrieves the image resource ID for the third image in the collection:

```
int imageId = treeCatalog[2].imageId;
```

Because the implementation details of `TreeCatalog` are not relevant to understanding `ViewPager`, the `TreeCatalog` code is not listed here. The source code to `TreeCatalog` is available at TreeCatalog.cs. Download this source file (or copy and paste the code into a new **TreeCatalog.cs** file) and add it to your project. Also, download and unzip the image files into your **Resources/drawable** folder and include them in the project.

Create a ViewPager Layout

Open **Resources/layout/Main.axml** and replace its contents with the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.view.ViewPager
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/viewpager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

</android.support.v4.view.ViewPager>
```

This XML defines a `ViewPager` that occupies the entire screen. Note that you must use the fully-qualified name **android.support.v4.view.ViewPager** because `ViewPager` is packaged in a support library. `ViewPager` is available only from Android Support Library v4; it is not available in the Android SDK.

Set up ViewPager

Edit **MainActivity.cs** and add the following `using` statement:

```
using Android.Support.V4.View;
```

Replace the `OnCreate` method with the following code:

```
protected override void OnCreate(Bundle bundle)
{
    base.OnCreate(bundle);
    SetContentView(Resource.Layout.Main);
    ViewPager viewPager = FindViewById<ViewPager>(Resource.Id.viewpager);
    TreeCatalog treeCatalog = new TreeCatalog();
}
```

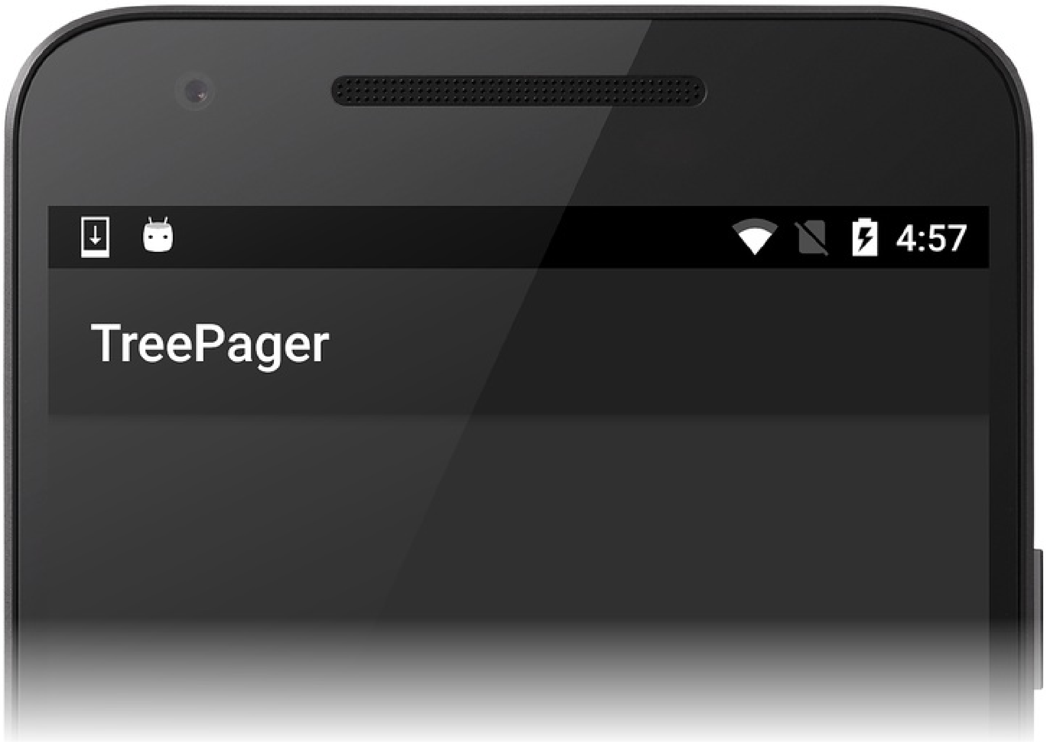
This code does the following:

1. Sets the view from the **Main.axml** layout resource.

2. Retrieves a reference to the `ViewPager` from the layout.

3. Instantiates a new `TreeCatalog` as the data source.

When you build and run this code, you should see a display that resembles the following screenshot:



At this point, the `ViewPager` is empty because it is lacking an adapter for accessing the content in **TreeCatalog**. In the next section, a **PagerAdapter** is created to connect the `ViewPager` to the **TreeCatalog**.

Create the Adapter

`ViewPager` uses an adapter controller object that sits between the `ViewPager` and the data source (see the illustration in Adapter). In order to access this data, `ViewPager` requires that you provide a custom adapter derived from `PagerAdapter`. This adapter populates each `ViewPager` page with content from the data source. Because this data source is app-specific, the custom adapter is the code that understands how to access the data. As the user swipes through pages of the `ViewPager`, the adapter extracts information from the data source and loads it into the pages for the `ViewPager` to display.

When you implement a `PagerAdapter`, you must override the following:

- **InstantiateItem** – Creates the page (`View`) for a given position and adds it to the `ViewPager`'s collection of views.
- **DestroyItem** – Removes a page from a given position.
- **Count** – Read-only property that returns the number of views (pages) available.
- **IsViewFromObject** – Determines whether a page is associated with a specific key object. (This object is created by the `InstantiateItem` method.) In this example, the key object is the `TreeCatalog` data object.

Add a new file called **TreePagerAdapter.cs** and replace its contents with the following code:

```
using System;
using Android.App;
using Android.Runtime;
using Android.Content;
using Android.Views;
using Android.Widget;
using Android.Support.V4.View;
using Java.Lang;

namespace TreePager
{
    class TreePagerAdapter : PagerAdapter
    {
        public override int Count
        {
            get { throw new NotImplementedException(); }
        }

        public override bool IsViewFromObject(View view, Java.Lang.Object obj)
        {
            throw new NotImplementedException();
        }

        public override Java.Lang.Object InstantiateItem (View container, int position)
        {
            throw new NotImplementedException();
        }

        public override void DestroyItem(View container, int position, Java.Lang.Object view)
        {
            throw new NotImplementedException();
        }
    }
}
```

This code stubs out the essential `PagerAdapter` implementation. In the following sections, each of these methods is replaced with working code.

Implement the Constructor

When the app instantiates the `TreePagerAdapter`, it supplies a context (the `MainActivity`) and an instantiated `TreeCatalog`. Add the following member variables and constructor to the top of the `TreePagerAdapter` class in **TreePagerAdapter.cs**:

```
Context context;
TreeCatalog treeCatalog;

public TreePagerAdapter (Context context, TreeCatalog treeCatalog)
{
    this.context = context;
    this.treeCatalog = treeCatalog;
}
```

The purpose of this constructor is to store the context and `TreeCatalog` instance that the `TreePagerAdapter` will use.

Implement Count

The `Count` implementation is relatively simple: it returns the number of trees in the tree catalog. Replace `Count` with the following code:

```
public override int Count
{
    get { return treeCatalog.NumTrees; }
}
```

The `NumTrees` property of `TreeCatalog` returns the number of trees (number of pages) in the data set.

Implement InstantiateItem

The `InstantiateItem` method creates the page for a given position. It must also add the newly-created view to the `ViewPager`'s view collection. To make this possible, the `ViewPager` passes itself as the container parameter.

Replace the `InstantiateItem` method with the following code:

```
public override Java.Lang.Object InstantiateItem (View container, int position)
{
    var imageView = new ImageView (context);
    imageView.SetImageResource (treeCatalog[position].imageId);
    var viewPager = container.JavaCast<ViewPager>();
    viewPager.AddView (imageView);
    return imageView;
}
```

This code does the following:

1. Instantiates a new `ImageView` to display the tree image at the specified position. The app's `MainActivity` is the context that will be passed to the `ImageView` constructor.

2. Sets the `ImageView` resource to the `TreeCatalog` image resource ID at the specified position.

3. Casts the passed container `View` to a `ViewPager` reference. Note that you must use `JavaCast<ViewPager>()` to properly perform this cast (this is needed so that Android performs a runtime-checked type conversion).

4. Adds the instantiated `ImageView` to the `ViewPager` and returns the `ImageView` to the caller.

When the `ViewPager` displays the image at `position`, it displays this `ImageView`. Initially, `InstantiateItem` is called twice to populate the first two pages with views. As the user scrolls, it is called again to maintain views just behind and ahead of the currently displayed item.

Implement DestroyItem

The `DestroyItem` method removes a page from the given position. In apps where the view at any given position can change, `ViewPager` must have some way of removing a stale view at that position before replacing it with a new view. In the `TreeCatalog` example, the view at each position does not change, so a view removed by `DestroyItem` will simply be re-added when `InstantiateItem` is called for that position. (For better efficiency, one could implement a pool to recycle `View`s that will be re-displayed at the same position.)

Replace the `DestroyItem` method with the following code:

```
public override void DestroyItem(View container, int position, Java.Lang.Object view)
{
    var viewPager = container.JavaCast<ViewPager>();
    viewPager.RemoveView(view as View);
}
```

This code does the following:

1. Casts the passed container `View` into a `ViewPager` reference.

2. Casts the passed Java object (`view`) into a C# `View` (`view as View`);

3. Removes the view from the `ViewPager` .

Implement IsViewFromObject

As the user slides left and right through pages of content, `ViewPager` calls `IsViewFromObject` to verify that the child `View` at the given position is associated with the adapter's object for that same position (hence, the adapter's object is called an *object key*). For relatively simple apps, the association is one of identity – the adapter's object key at that instance is the view that was previously returned to the `ViewPager` via `InstantiateItem` . However for other apps, the object key may be some other adapter-specific class instance that is associated with (but not the same as) the child view that `ViewPager` displays at that position. Only the adapter knows whether or not the passed view and object key are associated.

`IsViewFromObject` must be implemented for `PagerAdapter` to function properly. If `IsViewFromObject` returns `false` for a given position, `ViewPager` will not display the view at that position. In the `TreePager` app, the object key returned by `InstantiateItem` *is* the page `View` of a tree, so the code only has to check for identity (i.e, the object key and the view are one and the same). Replace `IsViewFromObject` with the following code:

```
public override bool IsViewFromObject(View view, Java.Lang.Object obj)
{
    return view == obj;
}
```

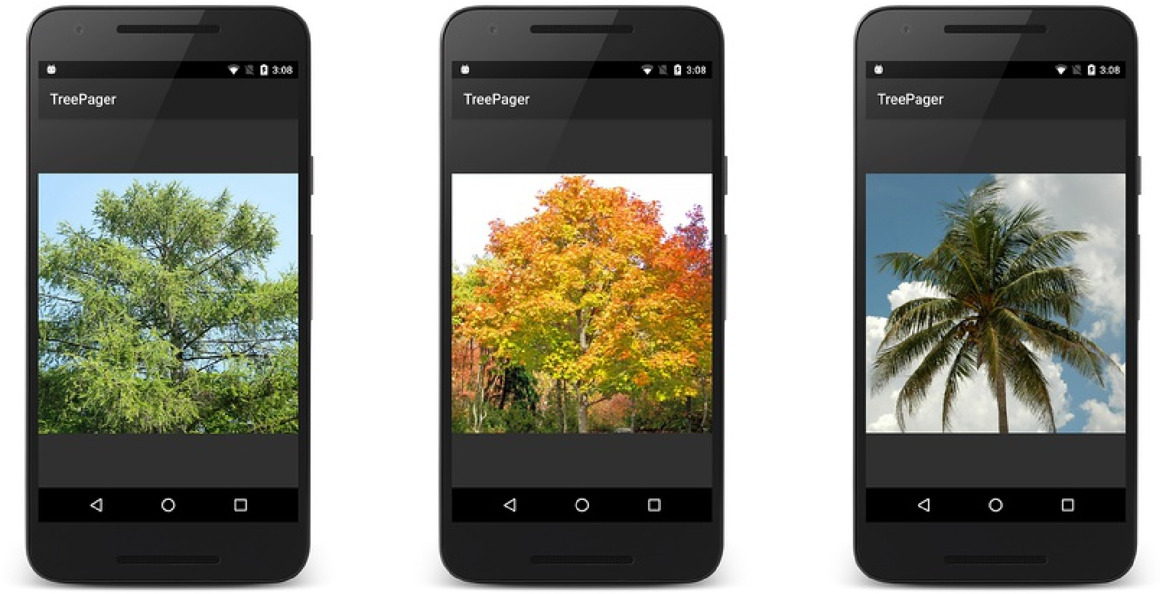
Add the Adapter to the ViewPager

Now that the `TreePagerAdapter` is implemented, it's time to add it to the `ViewPager` . In **MainActivity.cs**, add the following line of code to the end of the `OnCreate` method:

```
viewPager.Adapter = new TreePagerAdapter(this, treeCatalog);
```

This code instantiates the `TreePagerAdapter` , passing in the `MainActivity` as the context (`this`). The instantiated `TreeCatalog` is passed into the constructor's second argument. The `ViewPager` 's `Adapter` property is set to the instantiated `TreePagerAdapter` object; this plugs the `TreePagerAdapter` into the `ViewPager` .

The core implementation is now complete – build and run the app. You should see the first image of the tree catalog appear on the screen as shown on the left in the next screenshot. Swipe left to see more tree views, then swipe right to move back through the tree catalog:



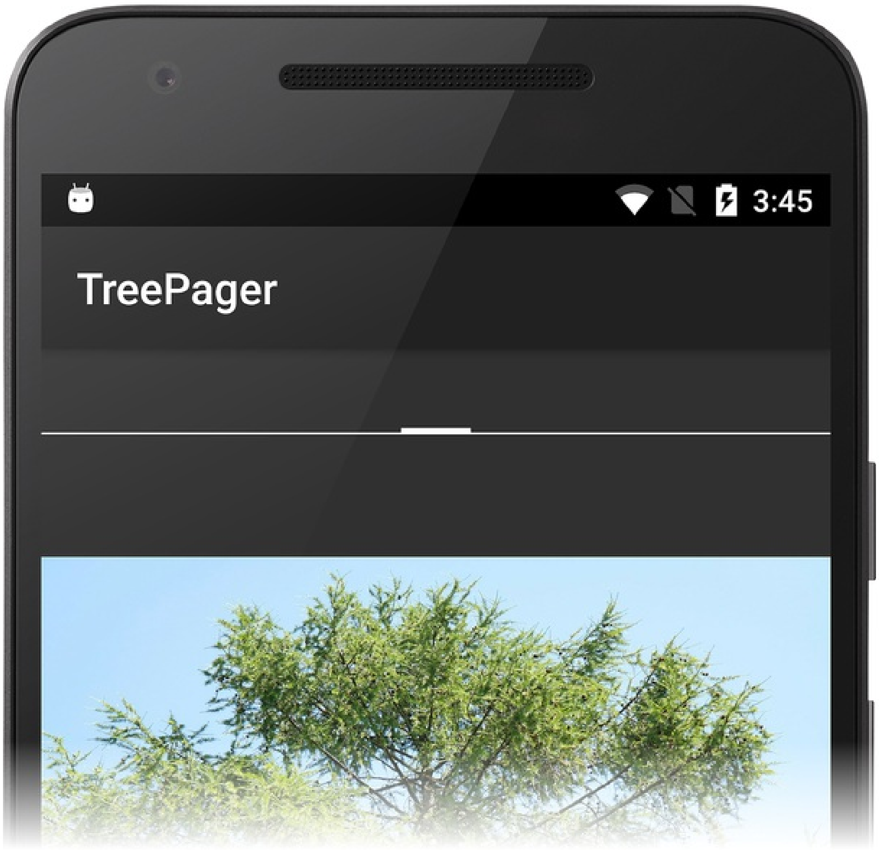
Add a Pager Indicator

This minimal `ViewPager` implementation displays the images of the tree catalog, but it provides no indication as to where the user is within the catalog. The next step is to add a `PagerTabStrip`. The `PagerTabStrip` informs the user as to which page is displayed and provides navigation context by displaying a hint of the previous and next pages. `PagerTabStrip` is intended to be used as an indicator for the current page of a `ViewPager`; it scrolls and updates as the user swipes through each page.

Open **Resources/layout/Main.axml** and add a `PagerTabStrip` to the layout:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.view.ViewPager
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/viewpager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <android.support.v4.view.PagerTabStrip
          android:layout_width="match_parent"
          android:layout_height="wrap_content"
          android:layout_gravity="top"
          android:paddingBottom="10dp"
          android:paddingTop="10dp"
          android:textColor="#fff" />

</android.support.v4.view.ViewPager>
```

`ViewPager` and `PagerTabStrip` are designed to work together. When you declare a `PagerTabStrip` inside a `ViewPager` layout, the `ViewPager` will automatically find the `PagerTabStrip` and connect it to the adapter. When you build and run the app, you should see the empty `PagerTabStrip` displayed at the top of each screen:



Display a Title

To add a title to each page tab, implement the `GetPageTitleFormatted` method in the `PagerAdapter`-derived class. `ViewPager` calls `GetPageTitleFormatted` (if implemented) to obtain the title string that describes the page at the specified position. Add the following method to the `TreePagerAdapter` class in **TreePagerAdapter.cs**:

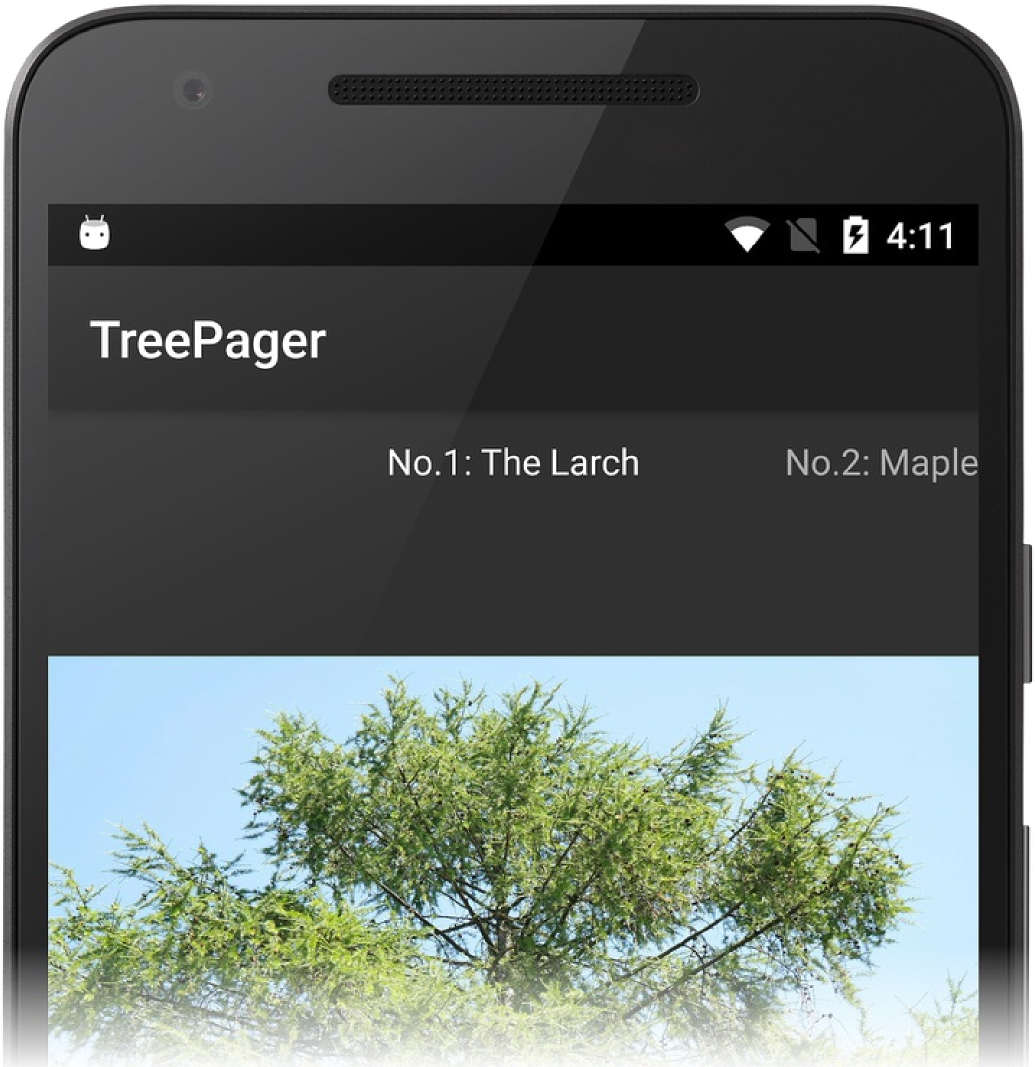
```
public override Java.Lang.ICharSequence GetPageTitleFormatted(int position)
{
    return new Java.Lang.String(treeCatalog[position].caption);
}
```

This code retrieves the tree caption string from the specified page (position) in the tree catalog, converts it into a Java `String`, and returns it to the `ViewPager`. When you run the app with this new method, each page displays the tree caption in the `PagerTabStrip`. You should see the tree name at the top of the screen without an underline:

You can swipe back and forth to view each captioned tree image in the catalog.

PagerTitleStrip Variation

`PagerTitleStrip` is very similar to `PagerTabStrip` except that `PagerTabStrip` adds an underline for the currently selected tab. You can replace `PagerTabStrip` with `PagerTitleStrip` in the above layout and run the app again to see how it looks with `PagerTitleStrip`:



Note that the underline is removed when you convert to `PagerTitleStrip`.

Summary

This walkthrough provided a step-by-step example of how to build a basic `ViewPager`-based app without using `Fragment`s. It presented an example data source containing images and caption strings, a `ViewPager` layout to display the images, and a `PagerAdapter` subclass that connects the `ViewPager` to the data source. To help the user navigate through the data set, instructions were included that explain how to add a `PagerTabStrip` or `PagerTitleStrip` to display the image caption at the top of each page.

Related Links

- TreePager (sample)

ViewPager with Fragments

7/8/2021 • 12 minutes to read • Edit Online

ViewPager is a layout manager that lets you implement gestural navigation. Gestural navigation allows the user to swipe left and right to step through pages of data. This guide explains how to implement a swipeable UI with ViewPager, using Fragments as the data pages.

Overview

`ViewPager` is often used in conjunction with fragments so that it is easier to manage the lifecycle of each page in the `ViewPager`. In this walkthrough, `ViewPager` is used to create an app called **FlashCardPager** that presents a series of math problems on flash cards. Each flash card is implemented as a fragment. The user swipes left and right through the flash cards and taps on a math problem to reveal its answer. This app creates a `Fragment` instance for each flash card and implements an adapter derived from `FragmentPagerAdapter`. In Viewpager and Views, most of the work was done in `MainActivity` lifecycle methods. In **FlashCardPager**, most of the work will be done by a `Fragment` in one of its lifecycle methods.

This guide does not cover the basics of fragments – if you are not yet familiar with fragments in Xamarin.Android, see Fragments to help you get started with fragments.

Start an App Project

Create a new Android project called **FlashCardPager**. Next, launch the NuGet Package Manager (for more information about installing NuGet packages, see Walkthrough: Including a NuGet in your project). Find and install the **Xamarin.Android.Support.v4** package as explained in Viewpager and Views.

Add an Example Data Source

In **FlashCardPager**, the data source is a deck of flash cards represented by the `FlashCardDeck` class; this data source supplies the `ViewPager` with item content. `FlashCardDeck` contains a ready-made collection of math problems and answers. The `FlashCardDeck` constructor requires no arguments:

```
FlashCardDeck flashCards = new FlashCardDeck();
```

The collection of flash cards in `FlashCardDeck` is organized such that each flash card can be accessed by an indexer. For example, the following line of code retrieves the fourth flash card problem in the deck:

```
string problem = flashCardDeck[3].Problem;
```

This line of code retrieves the corresponding answer to the previous problem:

```
string answer = flashCardDeck[3].Answer;
```

Because the implementation details of `FlashCardDeck` are not relevant to understanding `ViewPager`, the `FlashCardDeck` code is not listed here. The source code to `FlashCardDeck` is available at FlashCardDeck.cs. Download this source file (or copy and paste the code into a new **FlashCardDeck.cs** file) and add it to your project.

Create a ViewPager Layout

Open **Resources/layout/Main.axml** and replace its contents with the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.view.ViewPager
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/viewpager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    </android.support.v4.view.ViewPager>
```

This XML defines a `ViewPager` that occupies the entire screen. Note that you must use the fully-qualified name **android.support.v4.view.ViewPager** because `ViewPager` is packaged in a support library. `ViewPager` is available only from the Android Support Library v4; it is not available in the Android SDK.

Set up ViewPager

Edit **MainActivity.cs** and add the following `using` statements:

```
using Android.Support.V4.View;
using Android.Support.V4.App;
```

Change the `MainActivity` class declaration so that it is derived from `FragmentActivity`:

```
public class MainActivity : FragmentActivity
```

`MainActivity` is derived from `FragmentActivity` (rather than `Activity`) because `FragmentActivity` knows how to manage the support of fragments. Replace the `OnCreate` method with the following code:

```
protected override void OnCreate(Bundle bundle)
{
    base.OnCreate(bundle);
    SetContentView(Resource.Layout.Main);
    ViewPager viewPager = FindViewById<ViewPager>(Resource.Id.viewpager);
    FlashCardDeck flashCards = new FlashCardDeck();
}
```

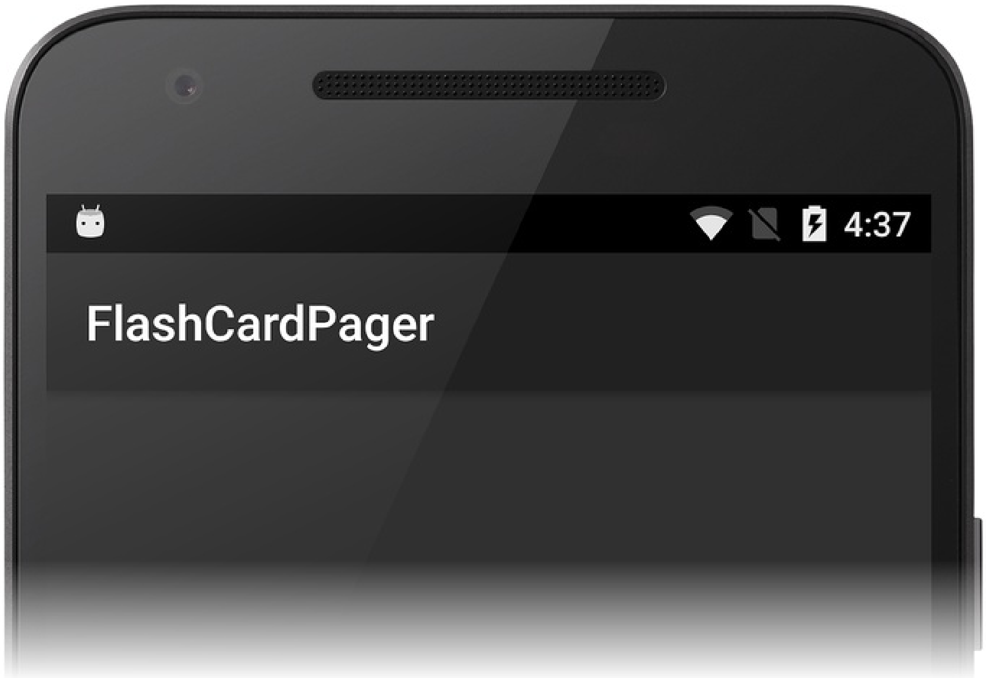
This code does the following:

1. Sets the view from the **Main.axml** layout resource.

2. Retrieves a reference to the `ViewPager` from the layout.

3. Instantiates a new `FlashCardDeck` as the data source.

When you build and run this code, you should see a display that resembles the following screenshot:



At this point, the `ViewPager` is empty because it is lacking the fragments that are used populate the `ViewPager`, and it is lacking an adapter for creating these fragments from the data in **FlashCardDeck**.

In the following sections, a `FlashCardFragment` is create to implement the functionality of each flash card, and a `FragmentPagerAdapter` is created to connect the `ViewPager` to the fragments created from data in the `FlashCardDeck`.

Create the Fragment

Each flash card will be managed by a UI fragment called `FlashCardFragment`. `FlashCardFragment`'s view will display the information contained with a single flash card. Each instance of `FlashCardFragment` will be hosted by the `ViewPager`. `FlashCardFragment`'s view will consist of a `TextView` that displays the flash card problem text. This view will implement an event handler that uses a `Toast` to display the answer when the user taps the flash card question.

Create the FlashCardFragment Layout

Before `FlashCardFragment` can be implemented, its layout must be defined. This layout is a fragment container layout for a single fragment. Add a new Android layout to **Resources/layout** called **flashcard_layout.axml**. Open **Resources/layout/flashcard_layout.axml** and replace its contents with the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/flash_card_question"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:gravity="center"
            android:textAppearance="@android:style/TextAppearance.Large"
            android:textSize="100sp"
            android:layout_centerHorizontal="true"
            android:layout_centerVertical="true"
            android:text="Question goes here" />
    </RelativeLayout>
```

This layout defines a single flash card fragment; each fragment is comprised of a `TextView` that displays a math problem using a large (100sp) font. This text is centered vertically and horizontally on the flash card.

Create the Initial FlashCardFragment Class

Add a new file called **FlashCardFragment.cs** and replace its contents with the following code:

```
using System;
using Android.OS;
using Android.Views;
using Android.Widget;
using Android.Support.V4.App;

namespace FlashCardPager
{
    public class FlashCardFragment : Android.Support.V4.App.Fragment
    {
        public FlashCardFragment() { }

        public static FlashCardFragment newInstance(String question, String answer)
        {
            FlashCardFragment fragment = new FlashCardFragment();
            return fragment;
        }
        public override View OnCreateView (
            LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)
        {
            View view = inflater.Inflate (Resource.Layout.flashcard_layout, container, false);
            TextView questionBox = (TextView)view.FindViewById (Resource.Id.flash_card_question);
            return view;
        }
    }
}
```

This code stubs out the essential `Fragment` definition that will be used to display a flash card. Note that `FlashCardFragment` is derived from the support library version of `Fragment` defined in `Android.Support.V4.App.Fragment`. The constructor is empty so that the `newInstance` factory method is used to create a new `FlashCardFragment` instead of a constructor.

The `OnCreateView` lifecycle method creates and configures the `TextView`. It inflates the layout for the fragment's `TextView` and returns the inflated `TextView` to the caller. `LayoutInflater` and `ViewGroup` are passed to `OnCreateView` so that it can inflate the layout. The `savedInstanceState` bundle contains data that `OnCreateView` uses to recreate the `TextView` from a saved state.

The fragment's view is explicitly inflated by the call to `inflater.Inflate`. The `container` argument is the view's parent, and the `false` flag instructs the inflater to refrain from adding the inflated view to the view's parent (it will be added when `ViewPager` call's the adapter's `GetItem` method later in this walkthrough).

Add State Code to FlashCardFragment

Like an Activity, a fragment has a `Bundle` that it uses to save and retrieve its state. In **FlashCardPager**, this `Bundle` is used to save the question and answer text for the associated flash card. In **FlashCardFragment.cs**, add the following `Bundle` keys to the top of the `FlashCardFragment` class definition:

```
private static string FLASH_CARD_QUESTION = "card_question";
private static string FLASH_CARD_ANSWER = "card_answer";
```

Modify the `newInstance` factory method so that it creates a `Bundle` object and uses the above keys to store the passed question and answer text in the fragment after it is instantiated:

```
public static FlashCardFragment newInstance(String question, String answer)
{
    FlashCardFragment fragment = new FlashCardFragment();

    Bundle args = new Bundle();
    args.PutString(FLASH_CARD_QUESTION, question);
    args.PutString(FLASH_CARD_ANSWER, answer);
    fragment.Arguments = args;

    return fragment;
}
```

Modify the fragment lifecycle method `OnCreateView` to retrieve this information from the passed-in Bundle and load the question text into the `TextBox`:

```
public override View OnCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)
{
    string question = Arguments.GetString(FLASH_CARD_QUESTION, "");
    string answer = Arguments.GetString(FLASH_CARD_ANSWER, "");

    View view = inflater.Inflate(Resource.Layout.flashcard_layout, container, false);
    TextView questionBox = (TextView)view.FindViewById(Resource.Id.flash_card_question);
    questionBox.Text = question;

    return view;
}
```

The `answer` variable is not used here, but it will be used later when event handler code is added to this file.

Create the Adapter

`ViewPager` uses an adapter controller object that sits between the `ViewPager` and the data source (see the illustration in the ViewPager Adapter article). To access this data, `ViewPager` requires that you provide a custom adapter derived from `PagerAdapter`. Because this example uses fragments, it uses a `FragmentPagerAdapter` – `FragmentPagerAdapter` is derived from `PagerAdapter`. `FragmentPagerAdapter` represents each page as a `Fragment` that is persistently kept in the fragment manager for as long as the user can return to the page. As the user swipes through pages of the `ViewPager`, the `FragmentPagerAdapter` extracts information from the data source and uses it to create `Fragment`s for the `ViewPager` to display.

When you implement a `FragmentPagerAdapter`, you must override the following:

- **Count** – Read-only property that returns the number of views (pages) available.

- **GetItem** – Returns the fragment to display for the specified page.

Add a new file called **FlashCardDeckAdapter.cs** and replace its contents with the following code:

```
using System;
using Android.Views;
using Android.Widget;
using Android.Support.V4.App;

namespace FlashCardPager
{
    class FlashCardDeckAdapter : FragmentPagerAdapter
    {
        public FlashCardDeckAdapter (Android.Support.V4.App.FragmentManager fm, FlashCardDeck flashCards)
            : base(fm)
        {
        }

        public override int Count
        {
            get { throw new NotImplementedException(); }
        }

        public override Android.Support.V4.App.Fragment GetItem(int position)
        {
            throw new NotImplementedException();
        }
    }
}
```

This code stubs out the essential `FragmentPagerAdapter` implementation. In the following sections, each of these methods is replaced with working code. The purpose of the constructor is to pass the fragment manager to the `FlashCardDeckAdapter`'s base class constructor.

Implement the Adapter Constructor

When the app instantiates the `FlashCardDeckAdapter`, it supplies a reference to the fragment manager and an instantiated `FlashCardDeck`. Add the following member variable to the top of the `FlashCardDeckAdapter` class in **FlashCardDeckAdapter.cs**:

```
public FlashCardDeck flashCardDeck;
```

Add the following line of code to the `FlashCardDeckAdapter` constructor:

```
this.flashCardDeck = flashCards;
```

This line of code stores the `FlashCardDeck` instance that the `FlashCardDeckAdapter` will use.

Implement Count

The `Count` implementation is relatively simple: it returns the number of flash cards in the flash card deck. Replace `Count` with the following code:

```
public override int Count
{
    get { return flashCardDeck.NumCards; }
}
```

The `NumCards` property of `FlashCardDeck` returns the number of flash cards (number of fragments) in the data set.

Implement GetItem

The `GetItem` method returns the fragment associated with the given position. When `GetItem` is called for a position in the flash card deck, it returns a `FlashCardFragment` configured to display the flash card problem at that position. Replace the `GetItem` method with the following code:

```
public override Android.Support.V4.App.Fragment GetItem(int position)
{
    return (Android.Support.V4.App.Fragment)
        FlashCardFragment.newInstance (
            flashCardDeck[position].Problem, flashCardDeck[position].Answer);
}
```

This code does the following:

1. Looks up the math problem string in the `FlashCardDeck` deck for the specified position.

2. Looks up the answer string in the `FlashCardDeck` deck for the specified position.

3. Calls the `FlashCardFragment` factory method `newInstance`, passing in the flash card problem and answer strings.

4. Creates and returns a new flash card `Fragment` that contains the question and answer text for that position.

When the `ViewPager` renders the `Fragment` at `position`, it displays the `TextBox` containing the math problem string residing at `position` in the flash card deck.

Add the Adapter to the ViewPager

Now that the `FlashCardDeckAdapter` is implemented, it's time to add it to the `ViewPager`. In **MainActivity.cs**, add the following line of code to the end of the `OnCreate` method:

```
FlashCardDeckAdapter adapter =
    new FlashCardDeckAdapter(SupportFragmentManager, flashCards);
viewPager.Adapter = adapter;
```

This code instantiates the `FlashCardDeckAdapter`, passing in the `SupportFragmentManager` in the first argument. (The `SupportFragmentManager` property of FragmentActivity is used to get a reference to the `FragmentManager` – for more information about the `FragmentManager`, see Managing Fragments.)

The core implementation is now complete – build and run the app. You should see the first image of the flash card deck appear on the screen as shown on the left in the next screenshot. Swipe left to see more flash cards, then swipe right to move back through the flash card deck:

Add a Pager Indicator

This minimal `ViewPager` implementation displays each flash card in the deck, but it provides no indication as to where the user is within the deck. The next step is to add a `PagerTabStrip`. The `PagerTabStrip` informs the user as to which problem number is displayed and provides navigation context by displaying a hint of the previous and next flash cards.

Open **Resources/layout/Main.axml** and add a `PagerTabStrip` to the layout:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.view.ViewPager xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

  <android.support.v4.view.PagerTabStrip
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:layout_gravity="top"
      android:paddingBottom="10dp"
      android:paddingTop="10dp"
      android:textColor="#fff" />

</android.support.v4.view.ViewPager>
```

When you build and run the app, you should see the empty `PagerTabStrip` displayed at the top of each flash card:

Display a Title

To add a title to each page tab, implement the `GetPageTitleFormatted` method in the adapter. `ViewPager` calls `GetPageTitleFormatted` (if implemented) to obtain the title string that describes the page at the specified position. Add the following method to the `FlashCardDeckAdapter` class in **FlashCardDeckAdapter.cs**:

```
public override Java.Lang.ICharSequence GetPageTitleFormatted(int position)
{
    return new Java.Lang.String("Problem " + (position + 1));
}
```

This code converts the position in the flash card deck to a problem number. The resulting string is converted into a Java `String` that is returned to the `ViewPager`. When you run the app with this new method, each page displays the problem number in the `PagerTabStrip`:

You can swipe back and forth to see the problem number in the flash card deck that is displayed at the top of each flash card.

Handle User Input

FlashCardPager presents a series of fragment-based flash cards in a `ViewPager`, but it does not yet have a way to reveal the answer for each problem. In this section, an event handler is added to the `FlashCardFragment` to display the answer when the user taps on the flash card problem text.

Open **FlashCardFragment.cs** and add the following code to the end of the `OnCreateView` method just before the view is returned to the caller:

```
questionBox.Click += delegate
{
    Toast.MakeText(Activity.ApplicationContext,
            "Answer: " + answer, ToastLength.Short).Show();
};
```

This `Click` event handler displays the answer in a Toast that appears when the user taps the `TextBox`. The `answer` variable was initialized earlier when state information was read from the Bundle that was passed to `OnCreateView`. Build and run the app, then tap the problem text on each flash card to see the answer:

The **FlashCardPager** presented in this walkthrough uses a `MainActivity` derived from `FragmentActivity`, but you can also derive `MainActivity` from `AppCompatActivity` (which also provides support for managing fragments). To view an `AppCompatActivity` example, see FlashCardPager in the Sample Gallery.

Summary

This walkthrough provided a step-by-step example of how to build a basic `ViewPager`-based app using `Fragment`s. It presented an example data source containing flash card questions and answers, a `ViewPager` layout to display the flash cards, and a `FragmentPagerAdapter` subclass that connects the `ViewPager` to the data source. To help the user navigate through the flash cards, instructions were included that explain how to add a `PagerTabStrip` to display the problem number at the top of each page. Finally, event handling code was added to display the answer when the user taps on a flash card problem.

Related Links

- FlashCardPager (sample)

Xamarin.Android Web View

7/8/2021 • 2 minutes to read • Edit Online

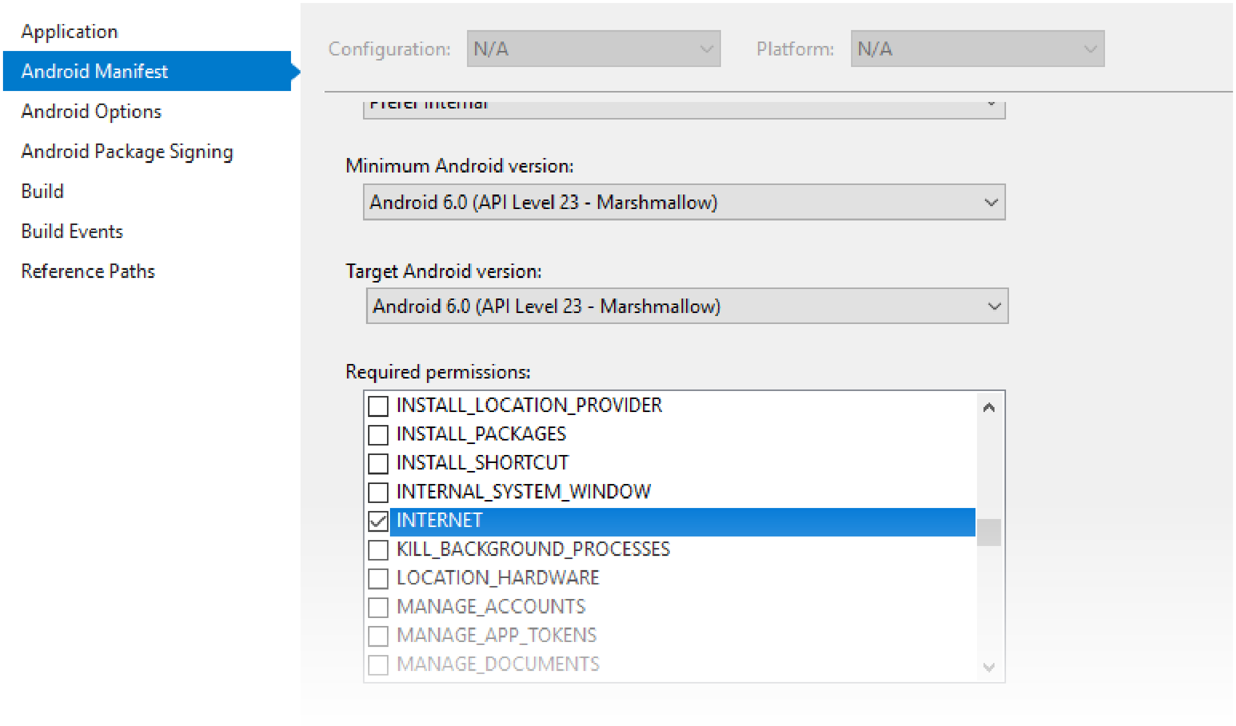
`WebView` allows you to create your own window for viewing web pages (or even develop a complete browser). In this tutorial, you'll create a simple `Activity` that can view and navigate web pages.

Create a new project named **HelloWebView**.

Open **Resources/Layout/Main.axml** and insert the following:

```
<?xml version="1.0" encoding="utf-8"?>
<WebView  xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/webview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" />
```

Because this application will access the Internet, you must add the appropriate permissions to the Android manifest file. Open your project's properties to specify which permissions your application requires to operate. Enable the `INTERNET` permission as shown below:



Now open **MainActivity.cs** and add a using directive for Webkit:

```
using Android.Webkit;
```

At the top of the `MainActivity` class, declare a `WebView` object:

```
WebView web_view;
```

When the **WebView** is asked to load a URL, it will by default delegate the request to the default browser. To have the **WebView** load the URL (rather than the default browser), you must subclass

`Android.Webkit.WebViewClient` and override the `ShouldOverriderUrlLoading` method. An instance of this custom `WebViewClient` is provided to the `WebView`. To do this, add the following nested `HelloWebViewClient` class inside `MainActivity`:

```
public class HelloWebViewClient : WebViewClient
{
    public override bool ShouldOverrideUrlLoading (WebView view, string url)
    {
        view.LoadUrl(url);
        return false;
    }
}
```

When `ShouldOverrideUrlLoading` returns `false`, it signals to Android that the current `WebView` instance handled the request and that no further action is necessary.

If you are targeting API level 24 or later, use the overload of `ShouldOverrideUrlLoading` that takes an `IWebResourceRequest` for the second argument instead of a `string`:

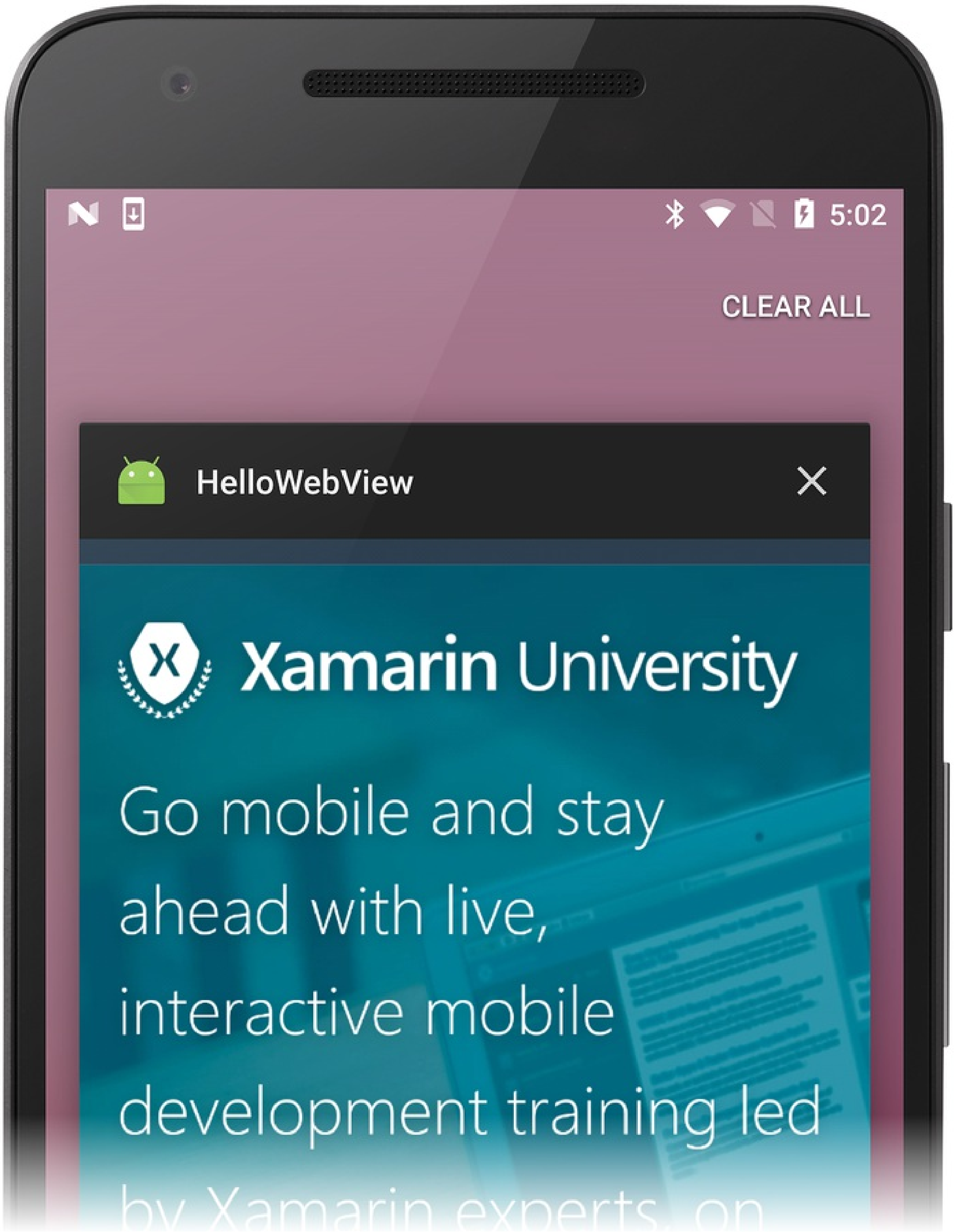
```
public class HelloWebViewClient : WebViewClient
{
    // For API level 24 and later
    public override bool ShouldOverrideUrlLoading (WebView view, IWebResourceRequest request)
    {
        view.LoadUrl(request.Url.ToString());
        return false;
    }
}
```

Next, use the following code for the `OnCreate()`) method:

```
protected override void OnCreate (Bundle bundle)
{
    base.OnCreate (bundle);

    // Set our view from the "main" layout resource
    SetContentView (Resource.Layout.Main);

    web_view = FindViewById<WebView> (Resource.Id.webview);
    web_view.Settings.JavaScriptEnabled = true;
    web_view.SetWebViewClient(new HelloWebViewClient());
    web_view.LoadUrl ("https://www.xamarin.com/university");
}
```

This initializes the member `WebView` with the one from the `Activity` layout and enables JavaScript for the `WebView` with `JavaScriptEnabled` `= true` (see the Call C# from JavaScript recipe for information about how to call C# functions from JavaScript). Finally, an initial web page is loaded with `LoadUrl(String)`.

Build and run the app. You should see a simple web page viewer app as the one seen in the following screenshot:



To handle the **BACK** button key press, add the following using statement:

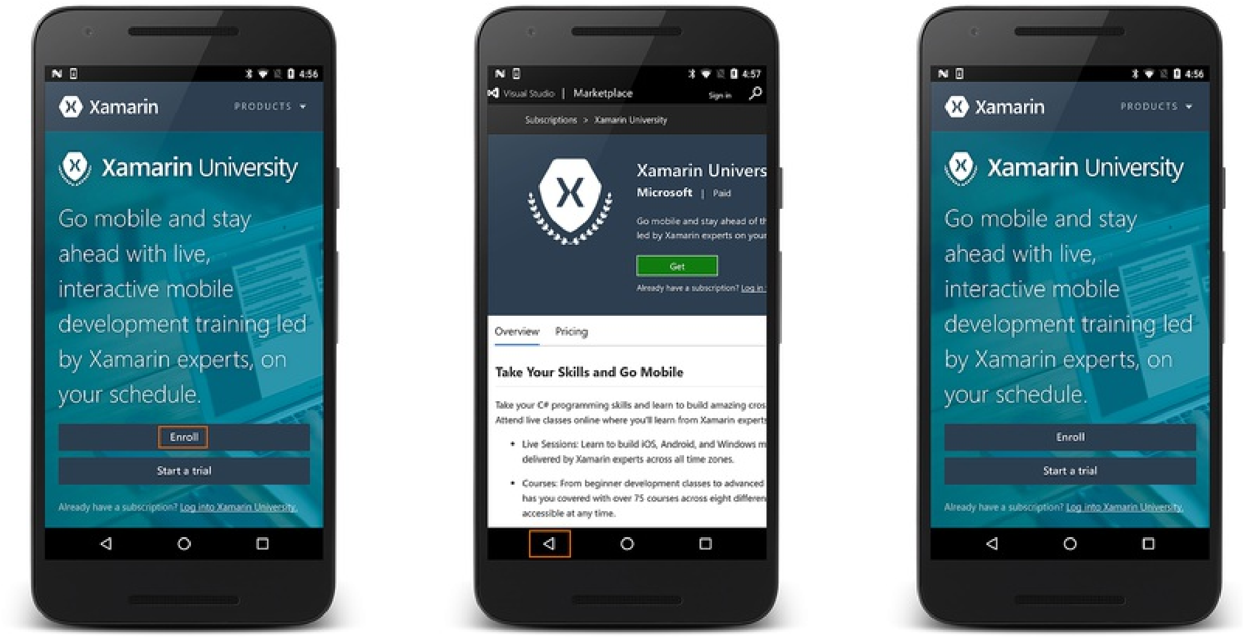
```
using Android.Views;
```

Next, add the following method inside the `HelloWebView` Activity:

```
public override bool OnKeyDown (Android.Views.Keycode keyCode, Android.Views.KeyEvent e)
{
    if (keyCode == Keycode.Back && web_view.CanGoBack ())
    {
        web_view.GoBack ();
        return true;
    }
    return base.OnKeyDown (keyCode, e);
}
```

This `OnKeyDown(int, KeyEvent)` callback method will be called whenever a button is pressed while the Activity is running. The condition inside uses the `KeyEvent` to check whether the key pressed is the **BACK** button and whether the `WebView` is actually capable of navigating back (if it has a history). If both are true, then the `GoBack()` method is called, which will navigate back one step in the `WebView` history. Returning `true` indicates that the event has been handled. If this condition is not met, then the event is sent back to the system.

Run the application again. You should now be able to follow links and navigate back through the page history:



Portions of this page are modifications based on work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

Related Links

- Call C# from JavaScript
- Android.Webkit.WebView
- KeyEvent

Xamarin.Android Platform Features

11/2/2020 • 4 minutes to read • Edit Online

Documents in this section cover features specific to Android. Here you'll find topics such as using Fragments, working with maps, and encapsulating data with Content Providers.

Android Beam

Android Beam is a new Near Field Communication (NFC) technology in Android 4 that allows applications to share information over NFC when in close proximity.

Working with Files

This section discusses how to access files in Xamarin.Android.

Fingerprint Authentication

This section discusses how to use fingerprint authentication, first introduced in Android 6.0, to a Xamarin.Android application.

Firebase Job Dispatcher

This guide discusses the Firebase Job Dispatcher and how to use it to simplify running background jobs in a Xamarin.Android app.

Fragments

Android 3.0 introduced Fragments, showing how to support more flexible designs for the many different screen sizes found on phones and tablets. This article will cover how to use Fragments to develop Xamarin.Android applications, and also how to support Fragments on pre-Android 3.0 (API Level 11) devices.

App-Linking

This guide will discuss how Android 6.0 supports *app-linking*, a technique that allows mobile apps to respond to URLs on websites. It will discuss how to implement app-linking in an Android 6.0 application and how to configure a website to grant permissions to the mobile app to handle app-links for the domain.

AndroidX

This article provides an outline of using AndroidX within your Xamarin.Android projects and provides links to documentation that illustrates how to migrate your application from the Android Support Library to AndroidX.

Android 10

This article provides an outline of the new features in Android 10, explains how to prepare Xamarin.Android for Android 10 development, and provides links to sample applications that illustrate how to use Android Oreo features in Xamarin.Android apps.

Android 9 Pie

This article provides an outline of the new features in Android Pie, explains how to prepare Xamarin.Android for Android Pie development, and provides an example app that illustrates how to use the new Android Pie display cutout and notification features in Xamarin.Android apps.

Android 8 Oreo

This article provides an outline of the new features in Android Oreo, explains how to prepare Xamarin.Android for Android Oreo development, and provides links to sample applications that illustrate how to use Android Oreo features in Xamarin.Android apps.

Android 7 Nougat

This article provides a high-level overview of the new features introduced in Android 7.0 Nougat.

Android 6 Marshmallow

This article provides a high-level overview of the new features introduced in Android 6.0 Marshmallow.

Android 5 Lollipop

This guide provides an overview of new Android 5.0 Lollipop features such as Material Theme, CardView, RecyclerView, and Heads Up Notifications, and it links to in-depth articles that help you use these new features in your app.

Android 4.4 KitKat

Android 4.4 (KitKat) comes loaded with a cornucopia of features for users and developers both. This guide highlights several of these features and provides code examples and implementation details to help you make the most out of KitKat.

Android 4.1 Jelly Bean

This document will provide a high-level overview of the new features for developers that were introduced in Android 4.1. These features include: enhanced notifications, updates to Android Beam to share large files, updates to multimedia, peer-to-peer network discovery, animations, new permissions.

Android 4.0 Ice Cream Sandwich

This article describes several of the new features available to application developers with the *Android 4 API - Ice Cream Sandwich*. It covers several new user interface technologies and then examines a variety of new capabilities that Android 4 offers for sharing data between applications and between devices.

Working with the Android Manifest

This article introduces the AndroidManifest.xml file, and how it maybe be used to control functionality and describe the requirements of a Mono for Android application.

Introduction to Content Providers

A ContentProvider encapsulates a data repository and provides an API to access it. The provider exists as part of an Android application that also provides a UI for displaying/managing the data. The key benefit of using a content provider is enabling other applications to easily access the encapsulated data using a provider client object (called a ContentResolver). Together a content provider and content resolver offer a consistent inter-application API for data access that is simple to build and consume. This document shows how to access and build ContentProviders with Xamarin.Android.

Maps and Location

This section discusses how to use maps and location with Xamarin.Android. It covers everything from leveraging the built-in maps application to using the Google Maps Android API v2 directly. Additionally, it explains how to use a single API to work with location services, which use cellular triangulation to allow an application to obtain location fixes, Wi-Fi location, and GPS.

Android Speech

This section discusses how to use the Android Text to Speech and Speech to Text facilities. It also covers installing language packs and interpretation of the text spoken to the device.

Binding a Java Library

This guide explains how to incorporate Java libraries into Xamarin.Android apps by creating a Bindings Library.

Bind a Kotlin Library

This guide explains how to create C# bindings to Kotlin code, making it possible to consume native libraries in a Xamarin.Android application.

Java Integration

This article provides an overview of the ways that developers can reuse existing Java components in Xamarin.Android apps.

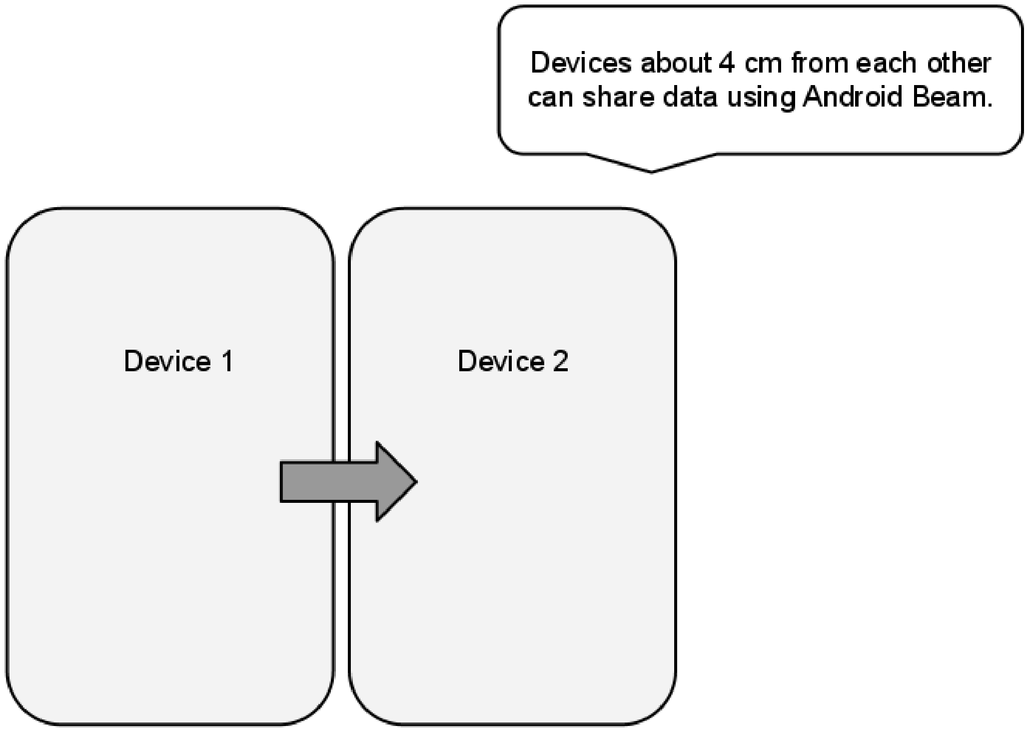
Renderscript

This guide discusses Renderscript.

Android Beam

7/8/2021 • 2 minutes to read • Edit Online

Android Beam is a Near Field Communication (NFC) technology introduced in Android 4.0 that allows applications to share information over NFC when in close proximity.



Android Beam works by pushing messages over NFC when two devices are in range. Devices about 4cm from each other can share data using Android Beam. An Activity on one device creates a message and specifies an Activity (or Activities) that can handle pushing it. When the specified Activity is in the foreground and the devices are in range, Android Beam will push the message to the second device. On the receiving device, an Intent is invoked containing the message data.

Android supports two ways of setting messages with Android Beam:

- `SetNdefPushMessage` - Before Android Beam is initiated, an application can call SetNdefPushMessage to specify an NdefMessage to push over NFC, and the Activity that is pushing it. This mechanism is best used when a message doesn't change while an application is in use.
- `SetNdefPushMessageCallback` - When Android Beam is initiated, an application can handle a callback to create an NdefMessage. This mechanism allows for message creation to be delayed until devices are in range. It supports scenarios where the message may vary based upon what's happening in the application.

In either case, to send data with Android Beam, an application sends an `NdefMessage`, packaging the data in several `NdefRecords`. Let's take a look at the key points that must be addressed before we can trigger Android Beam. First, we'll work with the callback style of creating an `NdefMessage`.

Creating a Message

We can register callbacks with an `NfcAdapter` in the Activity's `OnCreate` method. For example, assuming an `NfcAdapter` named `mNfcAdapter` is declared as a class variable in the Activity, we can write the following code to create the callback that will construct the message:

```
mNfcAdapter = NfcAdapter.GetDefaultAdapter (this);
mNfcAdapter.SetNdefPushMessageCallback (this, this);
```

The Activity, which implements `NfcAdapter.ICreateNdefMessageCallback`, is passed to the `SetNdefPushMessageCallback` method above. When Android Beam is initiated, the system will call `CreateNdefMessage`, from which the Activity can construct an `NdefMessage` as shown below:

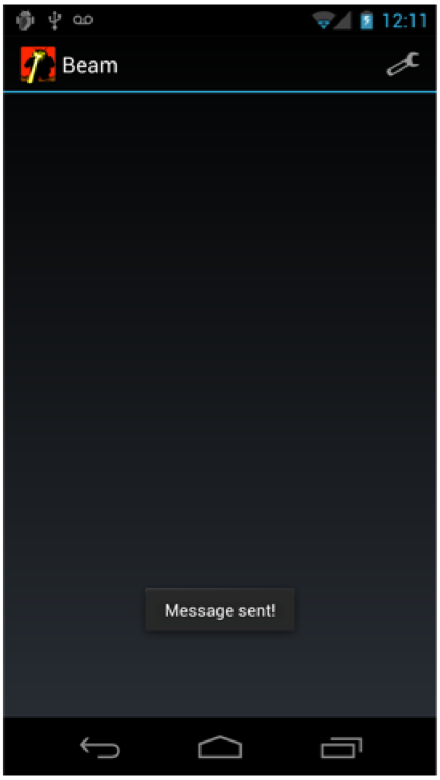
```
public NdefMessage CreateNdefMessage (NfcEvent evt)
{
    DateTime time = DateTime.Now;
    var text = ("Beam me up!\n\n" + "Beam Time: " +
        time.ToString ("HH:mm:ss"));
    NdefMessage msg = new NdefMessage (
        new NdefRecord[]{ CreateMimeRecord (
            "application/com.example.android.beam",
            Encoding.UTF8.GetBytes (text)) });
        } };
    return msg;
}

public NdefRecord CreateMimeRecord (String mimeType, byte [] payload)
{
    byte [] mimeBytes = Encoding.UTF8.GetBytes (mimeType);
    NdefRecord mimeRecord = new NdefRecord (
        NdefRecord.TnfMimeMedia, mimeBytes, new byte [0], payload);
    return mimeRecord;
}
```

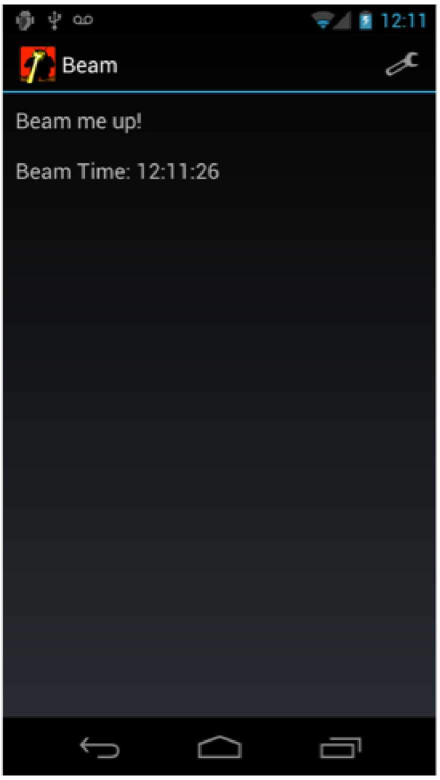
Receiving a Message

On the receiving side, the system invokes an Intent with the `ActionNdefDiscovered` action, from which we can extract the NdefMessage as follows:

```
IParcelable [] rawMsgs = intent.GetParcelableArrayExtra (NfcAdapter.ExtraNdefMessages);
NdefMessage msg = (NdefMessage) rawMsgs [0];
```

For a complete code example that uses Android Beam, shown running in the screenshot below, see the Android Beam demo in the Sample Gallery.





Related Links

- Android Beam Demo (sample)

Working with the Android Manifest

10/28/2019 • 6 minutes to read • Edit Online

AndroidManifest.xml is a powerful file in the Android platform that allows you to describe the functionality and requirements of your application to Android. However, working with it is not easy. Xamarin.Android helps to minimize this difficulty by allowing you to add custom attributes to your classes, which will then be used to automatically generate the manifest for you. Our goal is that 99% of our users should never need to manually modify **AndroidManifest.xml**.

AndroidManifest.xml is generated as part of the build process, and the XML found within **Properties/AndroidManifest.xml** is merged with XML that is generated from custom attributes. The resulting merged **AndroidManifest.xml** resides in the **obj** subdirectory; for example, it resides at **obj/Debug/android/AndroidManifest.xml** for Debug builds. The merging process is trivial: it uses custom attributes within the code to generate XML elements, and *inserts* those elements into **AndroidManifest.xml**.

The Basics

At compile time, assemblies are scanned for non-`abstract` classes that derive from Activity and have the `[Activity]` attribute declared on them. It then uses these classes and attributes to build the manifest. For example, consider the following code:

```
namespace Demo
{
    public class MyActivity : Activity
    {
    }
}
```

This results in nothing being generated in **AndroidManifest.xml**. If you want an `<activity/>` element to be generated, you need to use the `[Activity]` custom attribute:

```
namespace Demo
{
    [Activity]
    public class MyActivity : Activity
    {
    }
}
```

This example causes the following xml fragment to be added to **AndroidManifest.xml**:

```
<activity android:name="md5a7a3c803e481ad8926683588c7e9031b.MainActivity" />
```

The `[Activity]` attribute has no effect on `abstract` types; `abstract` types are ignored.

Activity Name

Beginning with Xamarin.Android 5.1, the type name of an activity is based on the MD5SUM of the assembly-qualified name of the type being exported. This allows the same fully-qualified name to be provided from two different assemblies and not get a packaging error. (Before Xamarin.Android 5.1, the default type name of the activity was created from the lowercased namespace and the class name.)

If you wish to override this default and explicitly specify the name of your activity, use the `Name` property:

```
[Activity (Name="awesome.demo.activity")]
public class MyActivity : Activity
{
}
```

This example produces the following xml fragment:

```
<activity android:name="awesome.demo.activity" />
```

NOTE

You should use the `Name` property only for backward-compatibility reasons, as such renaming can slow down type lookup at runtime. If you have legacy code that expects the default type name of the activity to be based on the lowercased namespace and the class name, see Android Callable Wrapper Naming for tips on maintaining compatibility.

Activity Title Bar

By default, Android gives your application a title bar when it is run. The value used for this is `/manifest/application/activity/@android:label`. In most cases, this value will differ from your class name. To specify your app's label on the title bar, use the `Label` property. For example:

```
[Activity (Label="Awesome Demo App")]
public class MyActivity : Activity
{
}
```

This example produces the following xml fragment:

```
<activity android:label="Awesome Demo App"
          android:name="md5a7a3c803e481ad8926683588c7e9031b.MainActivity" />
```

Launchable from Application Chooser

By default, your activity will not show up in Android's application launcher screen. This is because there will likely be many activities in your application, and you don't want an icon for every one. To specify which one should be launchable from the application launcher, use the `MainLauncher` property. For example:

```
[Activity (Label="Awesome Demo App", MainLauncher=true)]
public class MyActivity : Activity
{
}
```

This example produces the following xml fragment:

```
<activity android:label="Awesome Demo App"
          android:name="md5a7a3c803e481ad8926683588c7e9031b.MainActivity">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

Activity Icon

By default, your activity will be given the default launcher icon provided by the system. To use a custom icon, first add your **.png** to **Resources/drawable**, set its Build Action to **AndroidResource**, then use the `Icon` property to specify the icon to use. For example:

```
[Activity (Label="Awesome Demo App", MainLauncher=true, Icon="@drawable/myicon")]
public class MyActivity : Activity
{
}
```

This example produces the following xml fragment:

```
<activity android:icon="@drawable/myicon" android:label="Awesome Demo App"
          android:name="md5a7a3c803e481ad8926683588c7e9031b.MainActivity">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

Permissions

When you add permissions to the Android Manifest (as described in Add Permissions to Android Manifest), these permissions are recorded in **Properties/AndroidManifest.xml**. For example, if you set the `INTERNET` permission, the following element is added to **Properties/AndroidManifest.xml**:

```
<uses-permission android:name="android.permission.INTERNET" />
```

Debug builds automatically set some permissions to make debug easier (such as `INTERNET` and `READ_EXTERNAL_STORAGE`) – these settings are set only in the generated **obj/Debug/android/AndroidManifest.xml** and are not shown as enabled in the **Required permissions** settings.

For example, if you examine the generated manifest file at **obj/Debug/android/AndroidManifest.xml**, you may see the following added permission elements:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

In the Release build version of the manifest (at **obj/Debug/android/AndroidManifest.xml**), these permissions are *not* automatically configured. If you find that switching to a Release build causes your app to lose a permission that was available in the Debug build, verify that you have explicitly set this permission in the **Required permissions** settings for your app (see **Build > Android Application** in Visual Studio for Mac; see **Properties > Android Manifest** in Visual Studio).

Advanced Features

Intent Actions and Features

The Android manifest provides a way for you to describe the capabilities of your activity. This is done via Intents and the `[IntentFilter]` custom attribute. You can specify which actions are appropriate for your activity with the `IntentFilter` constructor, and which categories are appropriate with the `Categories` property. At least one activity must be provided (which is why activities are provided in the constructor). `[IntentFilter]` can be provided multiple times, and each use results in a separate `<intent-filter/>` element within the `<activity/>`.

For example:

```
[Activity (Label="Awesome Demo App", MainLauncher=true, Icon="@drawable/myicon")]
[IntentFilter (new[]{Intent.ActionView},
        Categories=new[]{Intent.CategorySampleCode, "my.custom.category"})]
public class MyActivity : Activity
{
}
```

This example produces the following xml fragment:

```
<activity android:icon="@drawable/myicon" android:label="Awesome Demo App"
          android:name="md5a7a3c803e481ad8926683588c7e9031b.MainActivity">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
  <intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.SAMPLE_CODE" />
    <category android:name="my.custom.category" />
  </intent-filter>
</activity>
```

Application Element

The Android manifest also provides a way for you to declare properties for your entire application. This is done via the `<application>` element and its counterpart, the Application custom attribute. Note that these are application-wide (assembly-wide) settings rather than per-Activity settings. Typically, you declare `<application>` properties for your entire application and then override these settings (as needed) on a per-Activity basis.

For example, the following `Application` attribute is added to **AssemblyInfo.cs** to indicate that the application can be debugged, that its user-readable name is **My App**, and that it uses the `Theme.Light` style as the default theme for all activities:

```
[assembly: Application (Debuggable=true,
                        Label="My App",
                        Theme="@android:style/Theme.Light")]
```

This declaration causes the following XML fragment to be generated in **obj/Debug/android/AndroidManifest.xml**:

```
<application android:label="My App"
             android:debuggable="true"
             android:theme="@android:style/Theme.Light"
                ... />
```

In this example, all activities in the app will default to the `Theme.Light` style. If you set an Activity's theme to `Theme.Dialog`, only that Activity will use the `Theme.Dialog` style while all other activities in your app will default to the `Theme.Light` style as set in the `<application>` element.

The `Application` element is not the only way to configure `<application>` attributes. Alternately, you can insert attributes directly into the `<application>` element of **Properties/AndroidManifest.xml**. These settings are merged into the final `<application>` element that resides in **obj/Debug/android/AndroidManifest.xml**. Note that the contents of **Properties/AndroidManifest.xml** always override data provided by custom attributes.

There are many application-wide attributes that you can configure in the `<application>` element; for more information about these settings, see the Public Properties section of ApplicationAttribute.

List of Custom Attributes

- Android.App.ActivityAttribute : Generates a /manifest/application/activity XML fragment
- Android.App.ApplicationAttribute : Generates a /manifest/application XML fragment
- Android.App.InstrumentationAttribute : Generates a /manifest/instrumentation XML fragment
- Android.App.IntentFilterAttribute : Generates a //intent-filter XML fragment
- Android.App.MetaDataAttribute : Generates a //meta-data XML fragment
- Android.App.PermissionAttribute : Generates a //permission XML fragment
- Android.App.PermissionGroupAttribute : Generates a //permission-group XML fragment
- Android.App.PermissionTreeAttribute : Generates a //permission-tree XML fragment
- Android.App.ServiceAttribute : Generates a /manifest/application/service XML fragment
- Android.App.UsesLibraryAttribute : Generates a /manifest/application/uses-library XML fragment
- Android.App.UsesPermissionAttribute : Generates a /manifest/uses-permission XML fragment
- Android.Content.BroadcastReceiverAttribute : Generates a /manifest/application/receiver XML fragment
- Android.Content.ContentProviderAttribute : Generates a /manifest/application/provider XML fragment
- Android.Content.GrantUriPermissionAttribute : Generates a /manifest/application/provider/grant-uri-permission XML fragment

File Storage and Access with Xamarin.Android

11/2/2020 • 7 minutes to read • Edit Online

A common requirement for Android apps is to manipulate files – saving pictures, downloading documents, or exporting data to share with other programs. Android (which is based on Linux) supports this by providing space for file storage. Android groups the filesystem into two different types of storage:

- **Internal Storage** – this is a portion of the file system that can be accessed only by the application or the operating system.
- **External Storage** – this is a partition for the storage of files that is accessible by all apps, the user, and possibly other devices. On some devices, external storage may be removable (such as an SD card).

These groupings are conceptual only, and don't necessarily refer to a single partition or directory on the device. An Android device will always provide partition for internal storage and external storage. It is possible that certain devices may have multiple partitions that are considered to be external storage. Regardless of the partition the APIs for reading, writing, or creating files is the same. There are two sets of APIs that a Xamarin.Android application may use for file access:

1. **The .NET APIs (provided by Mono and wrapped by Xamarin.Android)** – These includes the file system helpers provided by Xamarin.Essentials. The .NET APIs provide the best cross-platform compatibility and as such the focus of this guide will be on these APIs.
2. **The native Java file access APIs (provided by Java and wrapped by Xamarin.Android)** – Java provides its own APIs for reading and writing files. These are a completely acceptable alternative to the .NET APIs, but are specific to Android and are not suitable for apps that are intended to be cross-platform.

Reading and writing to files is almost identical in Xamarin.Android as it is to any other .NET application. The Xamarin.Android app determines the path to the file that will be manipulated, then uses standard .NET idioms for file access. Because the actual paths to internal and external storage may vary from device to device or from Android version to Android version, it is not recommended to hard code the path to the files. Instead, use the Xamarin.Android APIs to determine the path to files. That way, the .NET APIs for reading and writing files exposes the native Android APIs that will help with determining the path to files on internal and external storage.

Before discussing the APIs involved with file access, it is important to understand some of the details surrounding internal and external storage. This will be discussed in the next section.

Internal vs external storage

Conceptually, internal storage and external storage are very similar – they are both places at which a Xamarin.Android app may save files. This similarity may be confusing for developers who are not familiar with Android as it is not clear when an app should use internal storage vs external storage.

Internal storage refers to the non-volatile memory that Android allocates to the operating system, APKs, and for individual apps. This space is not accessible except by the operating system or apps. Android will allocate a directory in the internal storage partition for each app. When the app is uninstalled, all the files that are kept on internal storage in that directory will also be deleted. Internal storage is best suited for files that are only accessible to the app and that will not be shared with other apps or will have very little value once the app is uninstalled. On Android 6.0 or higher, files on internal storage may be automatically backed up by Google using the Auto Backup feature in Android 6.0. Internal storage has the following disadvantages:

- Files cannot be shared.
- Files will be deleted when the app is uninstalled.

- The space available on internal storage maybe limited.

External storage refers to file storage that is not internal storage and not exclusively accessible to an app. The primary purpose of external storage is to provide a place to put files that are meant to be shared between apps or that are too large to fit on the internal storage. The advantage of external storage is that it typically has much more space for files than internal storage. However, external storage is not always guaranteed to be present on a device and may require special permission from the user to access it.

> **NOTE**
>
> For devices that support multiple users, Android will provide each user their own directory on both internal and external storage. This directory is inaccessible to other users on the device. This separation is invisible to apps as long as they do not hardcode paths to files on internal or external storage.

As a rule of thumb, Xamarin.Android apps should prefer saving their files on internal storage when it is reasonable, and rely on external storage when files need to be shared with other apps, are very large, or should be retained even if the app is uninstalled. For example, a configuration file is best suited for a internal storage as it has no importance except to the app that creates it. In contrast, photos are a good candidate for external storage. They can be very large and in many cases the user may want to share them or access them even if the app is uninstalled.

This guide will focus on internal storage. Please see the guide External storage for details on using external storage in a Xamarin.Android application.

Working with internal storage

The internal storage directory for an application is determined by the operating system, and is exposed to Android apps by the `Android.Content.Context.FilesDir` property. This will return a `Java.IO.File` object representing the directory that Android has dedicated exclusively for the app. For example, an app with the package name **com.companyname** the internal storage directory might be:

```
/data/user/0/com.companyname/files
```

This document will refer to the internal storage directory as *INTERNAL_STORAGE*.

> **IMPORTANT**
>
> The exact path to the internal storage directory can vary from device to device and between versions of Android. Because of this, apps must not hard code the path to the internal files storage directory, and instead use the Xamarin.Android APIs, such as `System.Environment.GetFolderPath()`.

To maximize code sharing, Xamarin.Android apps (or Xamarin.Forms apps targeting Xamarin.Android) should use the `System.Environment.GetFolderPath()` method. In Xamarin.Android, this method will return a string for a directory that is the same location as `Android.Content.Context.FilesDir`. This method takes an enum, `System.Environment.SpecialFolder`, which is used to identify a set of enumerated constants that represent the paths of special folders used by the operating system. Not all of the `System.Environment.SpecialFolder` values will map to a valid directory on Xamarin.Android. The following table describes what path can be expected for a given value of `System.Environment.SpecialFolder`:

SYSTEM.ENVIRONMENT.SPECIALFOLDER	PATH
`ApplicationData`	*INTERNAL_STORAGE*/.config

SYSTEM.ENVIRONMENT.SPECIALFOLDER	PATH
`Desktop`	*INTERNAL_STORAGE*/Desktop
`LocalApplicationData`	*INTERNAL_STORAGE*/.local/share
`MyDocuments`	*INTERNAL_STORAGE*
`MyMusic`	*INTERNAL_STORAGE*/Music
`MyPictures`	*INTERNAL_STORAGE*/Pictures
`MyVideos`	*INTERNAL_STORAGE*/Videos
`Personal`	*INTERNAL_STORAGE*
`Fonts`	*INTERNAL_STORAGE*/.fonts
`Templates`	*INTERNAL_STORAGE*/Templates
`CommonApplicationData`	/usr/share
`CommonApplicationData`	/usr/share

Reading or Writing to files on internal storage

Any of the C# APIs for writing to a file are sufficient; all that is necessary is to get the path to the file that is in the directory allocated to the application. It is strongly recommended that the async versions of the .NET APIs are used to minimize any issues that may be associate with file access blocking the main thread.

This code snippet is one example of writing an integer to a UTF-8 text file to the internal storage directory of an application:

```
public async Task SaveCountAsync(int count)
{
    var backingFile =
Path.Combine(System.Environment.GetFolderPath(System.Environment.SpecialFolder.Personal), "count.txt");
    using (var writer = File.CreateText(backingFile))
    {
        await writer.WriteLineAsync(count.ToString());
    }
}
```

The next code snippet provides one way to read an integer value that was stored in a text file:

```
public async Task<int> ReadCountAsync()
{
    var backingFile =
Path.Combine(System.Environment.GetFolderPath(System.Environment.SpecialFolder.Personal), "count.txt");

    if (backingFile == null || !File.Exists(backingFile))
    {
        return 0;
    }

    var count = 0;
    using (var reader = new StreamReader(backingFile, true))
    {
        string line;
        while ((line = await reader.ReadLineAsync()) != null)
        {
            if (int.TryParse(line, out var newcount))
            {
                count = newcount;
            }
        }
    }

    return count;
}
```

Using Xamarin.Essentials – File System Helpers

Xamarin.Essentials is a set of APIs for writing cross-platform compatible code. The File System Helpers is a class that contains a series of helpers to simplify locating the application's cache and data directories. This code snippet provides an example of how to find the internal storage directory and the cache directory for an app:

```
// Get the path to a file on internal storage
var backingFile = Path.Combine(Xamarin.Essentials.FileSystem.AppDataDirectory, "count.txt");

// Get the path to a file in the cache directory
var cacheFile = Path.Combine(Xamarin.Essentials.FileSystem.CacheDirectory, "count.txt");
```

Hiding files from the `MediaStore`

The `MediaStore` is an Android component that collects meta data about media files (videos, music, images) on an Android device. Its purpose is simplify the sharing of these files across all Android apps on the device.

Private files will not show up as shareable media. For example, if an app saves a picture to its private external storage, then that file will not be picked up by the media scanner (`MediaStore`).

Public files will be picked up by `MediaStore`. Directories that have a zero byte file name **.NOMEDIA** will not be scanned by `MediaStore`.

Related Links

- External Storage
- Save files on device storage
- Xamarin.Essentials File System Helpers
- Backup user data with Auto Backup
- Adoptable Storage

External storage

7/8/2021 • 9 minutes to read • Edit Online

External storage refers to file storage that is not on internal storage and not exclusively accessible to the app that is responsible for the file. The primary purpose of external storage is to provide a place to put files that are meant to be shared between apps or that are too large to fit on the internal storage.

Historically speaking, external storage referred to a disk partition on removable media such as an SD card (was also known as *portable storage*). This distinction is no longer as relevant as Android devices have evolved and many Android devices no longer support removable storage. Instead some devices will allocate some of their internal non-volatile memory which Android to perform the same function removable media. This is known as *emulated* storage and is still considered to be external storage. Alternately, some Android devices may have multiple external storage partitions. For example, an Android tablet (in addition to its internal storage) might have emulated storage and one or more slots for an SD card. All of these partitions are treated by Android as external storage.

On devices that have multiple users, each user will have a dedicated directory on the primary external storage partition for their external storage. Apps running as one user will not have access to files from another user on the device. The files for all users are still world-readable and world-writeable; however, Android will sandbox each user profile fromthe others.

Reading and writing to files is almost identical in Xamarin.Android as it is to any other .NET application. The Xamarin.Android app determines the path to the file that will be manipulated, then uses standard .NET idioms for file access. Because the actual paths to internal and external storage may vary from device to device or from Android version to Android version, it is not recommended to hard code the path to the files. Instead, Xamarin.Android exposes the native Android APIs that will help with determining the path to files on internal and external storage.

This guide will discuss the concepts and APIs in Android that are specific to external storage.

Public and private files on external storage

There are two different types of files that an app may keep on external storage:

- **Private** files – Private files are files that are specific to your application (but are still world-readable and world-writable). Android expects that private files are stored in a specific directory on external storage. Even though the files are called "private", they are still visible and accessible by other apps on the device, they are not afforded any special protection by Android.
- **Public** files – These are files that are not considered to be specific to the application and are meant to be freely shared.

The differences between these files is primarily conceptual. Private files are private in the sense that they are considered to be a part of the application, while public files are any other files that exist on external storage. Android provides two different APIs for resolving the paths to private and public files, but otherwise the same .NET APIs are used to read and write to these files. These are the same APIs that are discussed in the section on reading and writing.

Private external files

Private external files are considered to be specific to an application (similar to internal files) but are being kept on external storage for any number of reasons (such as being too large for internal storage). Similar to internal files, these files will be deleted when the app is uninstalled by the user.

The primary location for private external files is found by calling the method `Android.Content.Context.GetExternalFilesDir(string type)`. This method will return a `Java.IO.File` object that represents the private external storage directory for the app. Passing `null` to this method will return the path to the user's storage directory for the application. As an example, for an application with the package name `com.companyname.app`, the "root" directory of the private external files would be:

```
/storage/emulated/0/Android/data/com.companyname.app/files/
```

This document will refer to the storage directory for private files on external storage as *PRIVATE_EXTERNAL_STORAGE*.

The parameter for `GetExternalFilesDir()` is a string that specifies an *application directory*. This is a directory intended to provide a standard location for a logical organization of files. The string values are available through constants on the `Android.OS.Environment` class:

ANDROID.OS.ENVIRONMENT	DIRECTORY
DirectoryAlarms	*PRIVATE_EXTERNAL_STORAGE*/Alarms
DirectoryDcim	*PRIVATE_EXTERNAL_STORAGE*/DCIM
DirectoryDownloads	*PRIVATE_EXTERNAL_STORAGE*/Download
DirectoryDocuments	*PRIVATE_EXTERNAL_STORAGE*/Documents
DirectoryMovies	*PRIVATE_EXTERNAL_STORAGE*/Movies
DirectoryMusic	*PRIVATE_EXTERNAL_STORAGE*/Music
DirectoryNotifications	*PRIVATE_EXTERNAL_STORAGE*/Notifications
DirectoryPodcasts	*PRIVATE_EXTERNAL_STORAGE*/Podcasts
DirectoryRingtones	*PRIVATE_EXTERNAL_STORAGE*/Ringtones
DirectoryPictures	*PRIVATE_EXTERNAL_STORAGE*/Pictures

For devices that have multiple external storage partitions, each partition will have a directory that is intended for private files. The method `Android.Content.Context.GetExternalFilesDirs(string type)` will return an array of `Java.IO.Files`. Each object will represent a private application-specific directory on all shared/external storage devices where the application can place the files it owns.

> **IMPORTANT**
>
> The exact path to the private external storage directory can vary from device to device and between versions of Android. Because of this, apps must not hard code the path to this directory, and instead use the Xamarin.Android APIs, such as `Android.Content.Context.GetExternalFilesDir()`.

Public external files

Public files are files that exist on external storage that are not stored in the directory that Android allocates for private files. Public files will not be deleted when the app is uninstalled. Android apps must be granted permission before they can read or write any public files. It is possible for public files to exist anywhere on external storage, but by convention Android expects public files to exist in the directory identified by the property `Android.OS.Environment.ExternalStorageDirectory`. This property will return a `Java.IO.File` object that represents the primary external storage directory. As an example, `Android.OS.Environment.ExternalStorageDirectory` may refer to the following directory:

```
/storage/emulated/0/
```

This document will refer to the storage directory for public files on external storage as *PUBLIC_EXTERNAL_STORAGE*.

Android also supports the concept of application directories on *PUBLIC_EXTERNAL_STORAGE*. These directories are exactly the same as the application directories for `PRIVATE_EXTERNAL_STORAGE` and are described in the table in the previous section. The method `Android.OS.Environment.GetExternalStoragePublicDirectory(string directoryType)` will return a `Java.IO.File` object that correspond to a public application directory. The `directoryType` parameter is a mandatory parameter and cannot be `null`.

For example, calling `Environment.GetExternalStoragePublicDirectory(Environment.DirectoryDocuments).AbsolutePath` will return a string which will resemble:

```
/storage/emulated/0/Documents
```

> **IMPORTANT**
>
> The exact path to the public external storage directory can vary from device to device and between versions of Android. Because of this, apps must not hard code the path to this directory, and instead use the Xamarin.Android APIs, such as `Android.OS.Environment.ExternalStorageDirectory`.

Working with external storage

Once a Xamarin.Android app has obtained the full path to a file, it should utilize any of the standard .NET APIs for creating, reading, writing, or deleting files. This maximizes the amount of cross platform compatible code for an app. However, before attempting to access a file a Xamarin.Android app must ensure that is it possible to access that file.

1. **Verify external storage** – Depending on the nature of the external storage, it is possible that it might not be mounted and usable by the app. All apps should check the state of the external storage before attempting to use it.
2. **Perform a runtime permission check** – An Android app must request permission from the user in order to access external storage. This means that a run time permission request should be performed prior to any file access. The guide Permissions In Xamarin.Android contains more details on Android permissions.

Each of these two tasks will be discussed below.

Verifying that external storage is available

The first step before writing to external storage is to check that it is readable or writeable. The `Android.OS.Environment.ExternalStorageState` property holds a string that identifies the state of the external storage. This property will return a string that represents the state. This table is a list of the `ExternalStorageState` values that might be returned by `Environment.ExternalStorageState`:

EXTERNALSTORAGESTATE	DESCRIPTION
MediaBadRemoval	The media was abruptly removed without being properly unmounted.
MediaChecking	The media is present but undergoing a disk check.
MediaEjecting	Media is in the process of being unmounted and ejected.
MediaMounted	Media is mounted and can be read or written to.
MediaMountedReadOnly	Media is mounted but can only be read from.
MediaNofs	Media is present but does not contain a filesystem suitable for Android.
MediaRemoved	There is no media present.
MediaShared	Media is present, but is not mounted. It is being shared via USB with another device.
MediaUnknown	The state of the media is unrecognized by Android.
MediaUnmountable	The media is present but cannot be mounted by Android.
MediaUnmounted	The media is present but is not mounted.

Most Android apps will only need to check if external storage is mounted. The following code snippet shows how to verify that external storage is mounted for read-only access or read-write access:

```
bool isReadonly = Environment.MediaMountedReadOnly.Equals(Environment.ExternalStorageState);
bool isWriteable = Environment.MediaMounted.Equals(Environment.ExternalStorageState);
```

External storage permissions

Android considers accessing external storage to be a *dangerous permission*, which typically requires the user to grant their permission to access the resource. The user may revoke this permission at any time. This means that a run time permission request should be performed prior to any file access. Apps are automatically granted permissions to read and write their own private files. It is possible for apps to read and write the private files that belong to other apps after being granted permission by the user.

All Android apps must declare one of the two permissions for external storage in the **AndroidManifest.xml** . To identify the permissions, one of the following two `uses-permission` elements must be add to **AndroidManifest.xml**:

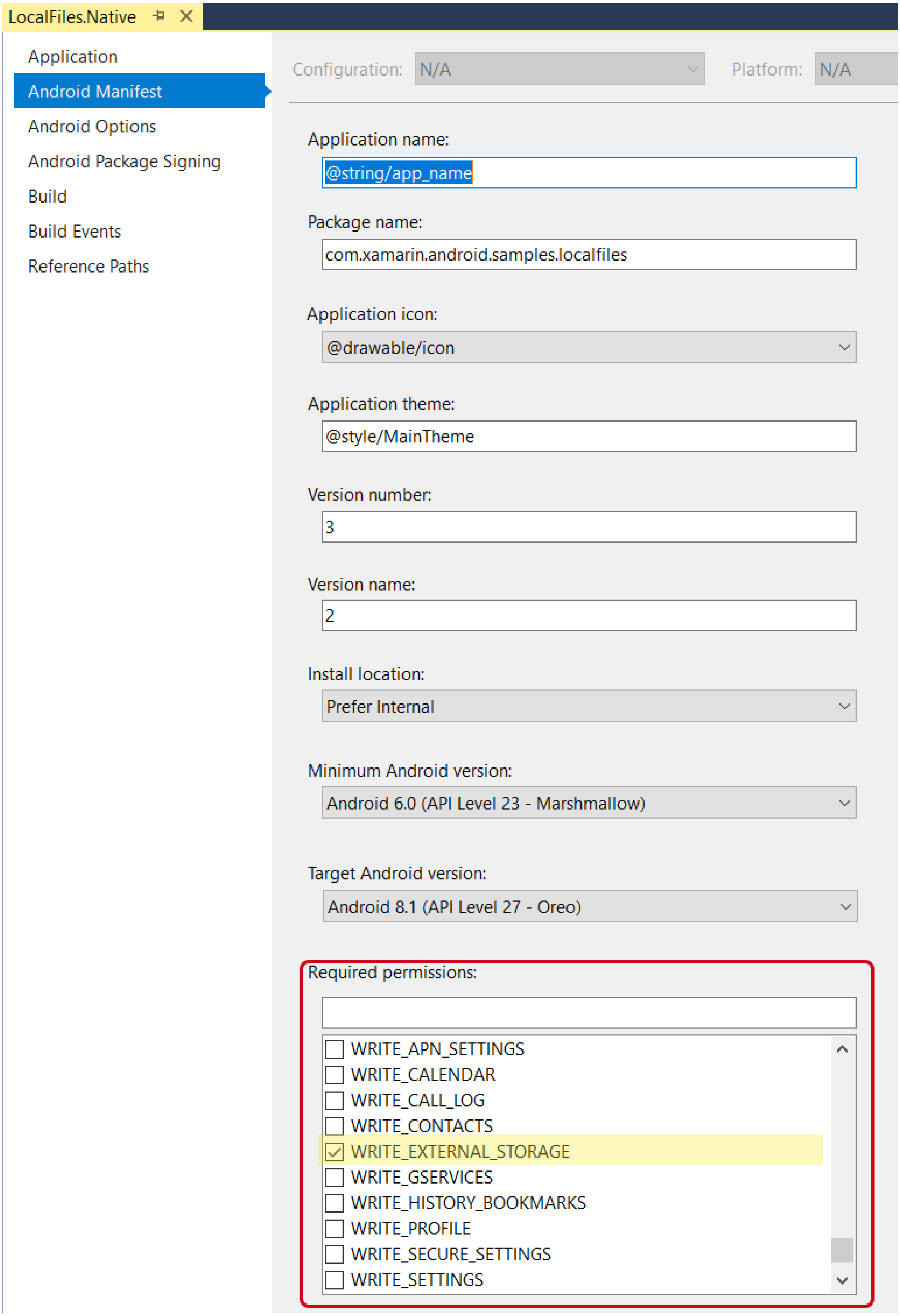
```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

NOTE

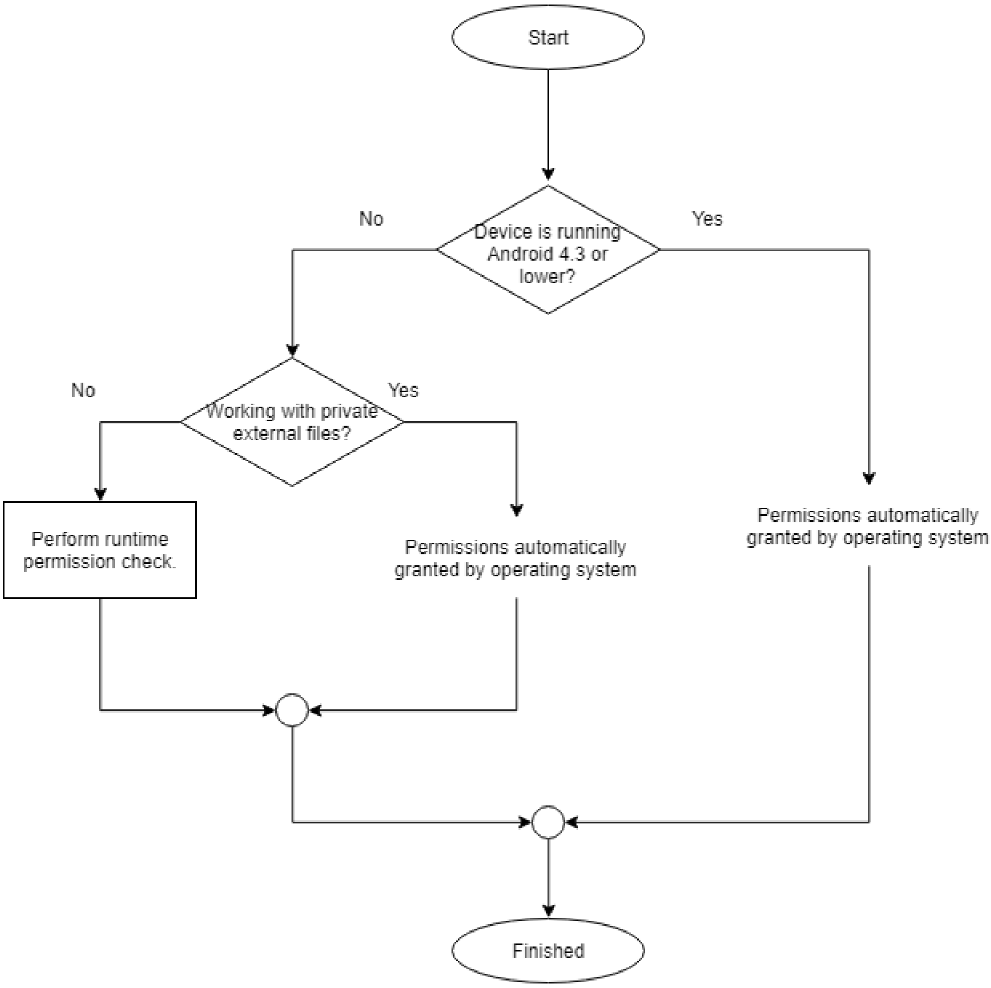
If the user grants `WRITE_EXTERNAL_STORAGE` , then `READ_EXTERNAL_STORAGE` is also implicitly granted. It is not necessary to request both permissions in **AndroidManifest.xml**.

- Visual Studio
- Visual Studio for Mac

The permissions may also be added using the **Android Manifest** tab of the **solution properties**:



Generally speaking, all dangerous permissions must be approved by the user. The permissions for external storage are an anomaly in that there are exceptions to this rule, depending on the version of Android that the app is running:



For more information on performing runtime permission requests, please consult the guide Permissions In Xamarin.Android. The **monodroid-sample** LocalFiles also demonstrates one way of performing runtime permission checks.

Granting and revoking permissions with ADB

In the course of developing an Android app, it may be necessary to grant and revoke permissions to test the various work flows involved with runtime permission checks. It is possible to do this at the command prompt using ADB. The following command line snippets demonstrate how to grant or revoke permissions using ADB for an Android app whose package name is **com.companyname.app**:

```
$ adb shell pm grant com.companyname.app android.permission.WRITE_EXTERNAL_STORAGE

$ adb shell pm revoke com.companyname.app android.permission.WRITE_EXTERNAL_STORAGE
```

Deleting files

Any of the standard C# APIs can be used to delete a file from external storage, such as `System.IO.File.Delete`. It is also possible to use the Java APIs at the expense of code portability. For example:

```
System.IO.File.Delete("/storage/emulated/0/Android/data/com.companyname.app/files/count.txt");
```

Related Links

- Xamarin.Android Local Files sample on **monodroid-samples**
- Permissions In Xamarin.Android

Fingerprint Authentication

11/2/2020 • 2 minutes to read • Edit Online

This guide discusses how to add fingerprint authentication, introduced in Android 6.0, to a Xamarin.Android application.

Fingerprint Authentication Overview

The arrival of fingerprint scanners on Android devices provides applications with an alternative to the traditional username/password method of user authentication. The use of fingerprints to authenticate a user makes it possible for an application to incorporate security that is less intrusive than a username and password.

The FingerprintManager APIs target devices with a fingerprint scanner and are running API level 23 (Android 6.0) or higher. The APIs are found in the `Android.Hardware.Fingerprints` namespace. The Android Support Library v4 provides versions of the fingerprint APIs meant for older versions of Android. The compatibility APIs are found in the `Android.Support.v4.Hardware.Fingerprint` namespace, are distributed through the Xamarin.Android.Support.v4 NuGet package.

The FingerprintManager (and its Support Library counterpart, FingerprintManagerCompat) is the primary class for using the fingerprint scanning hardware. This class is an Android SDK wrapper around the system level service that manages interactions with the hardware itself. It is responsible for starting the fingerprint scanner and for responding to feedback from the scanner. This class has a fairly straightforward interface with only three members:

- `Authenticate` – This method will initialize the hardware scanner and start the service in the background, waiting for the user to scan their fingerprint.
- `EnrolledFingerprints` – This property will return `true` if the user has registered one or more fingerprints with the device.
- `HardwareDetected` – This property is used to determine if the device supports fingerprint scanning.

The `FingerprintManager.Authenticate` method is used by an Android application to start the fingerprint scanner. The following snippet is an example of how to invoke it using the Support Library compatibility APIs:

```
// context is any Android.Content.Context instance, typically the Activity
FingerprintManagerCompat fingerprintManager = FingerprintManagerCompat.From(context);
fingerprintManager.Authenticate(FingerprintManager.CryptoObject crypto,
                                int flags,
                                CancellationSignal cancel,
                                FingerprintManagerCompat.AuthenticationCallback callback,
                                Handler handler
                               );
```

This guide will discuss how to use the `FingerprintManager` APIs to enhance an Android application with fingerprint authentication. It will cover how to instantiate and create a `CryptoObject` to help secure the results from the fingerprint scanner. We'll examine how an application should subclass `FingerprintManager.AuthenticationCallback` and respond to feedback from the fingerprint scanner. Finally, we'll see how to enroll a fingerprint on an Android device or emulator and how to use **adb** to simulate a fingerprint scan.

Requirements

Fingerprint Authentication requires Android 6.0 (API level 23) or higher and a device with a fingerprint scanner.

A fingerprint must already be enrolled with the device for each user that is to be authenticated. This involves setting up a screen lock that uses a password, PIN, swipe pattern, or facial recognition. It is possible to simulate some of the fingerprint authentication functionality in an Android Emulator. For more information on these two topics, please see the Enrolling a Fingerprint section.

Related Links

- Fingerprint Guide Sample App
- Fingerprint Dialog Sample
- Requesting Permissions at Runtime
- android.hardware.fingerprint
- android.support.v4.hardware.fingerprint
- Android.Content.Context
- Fingerprint and payments API (video)

Getting Started with Fingerprint Authentication

7/8/2021 • 3 minutes to read • Edit Online

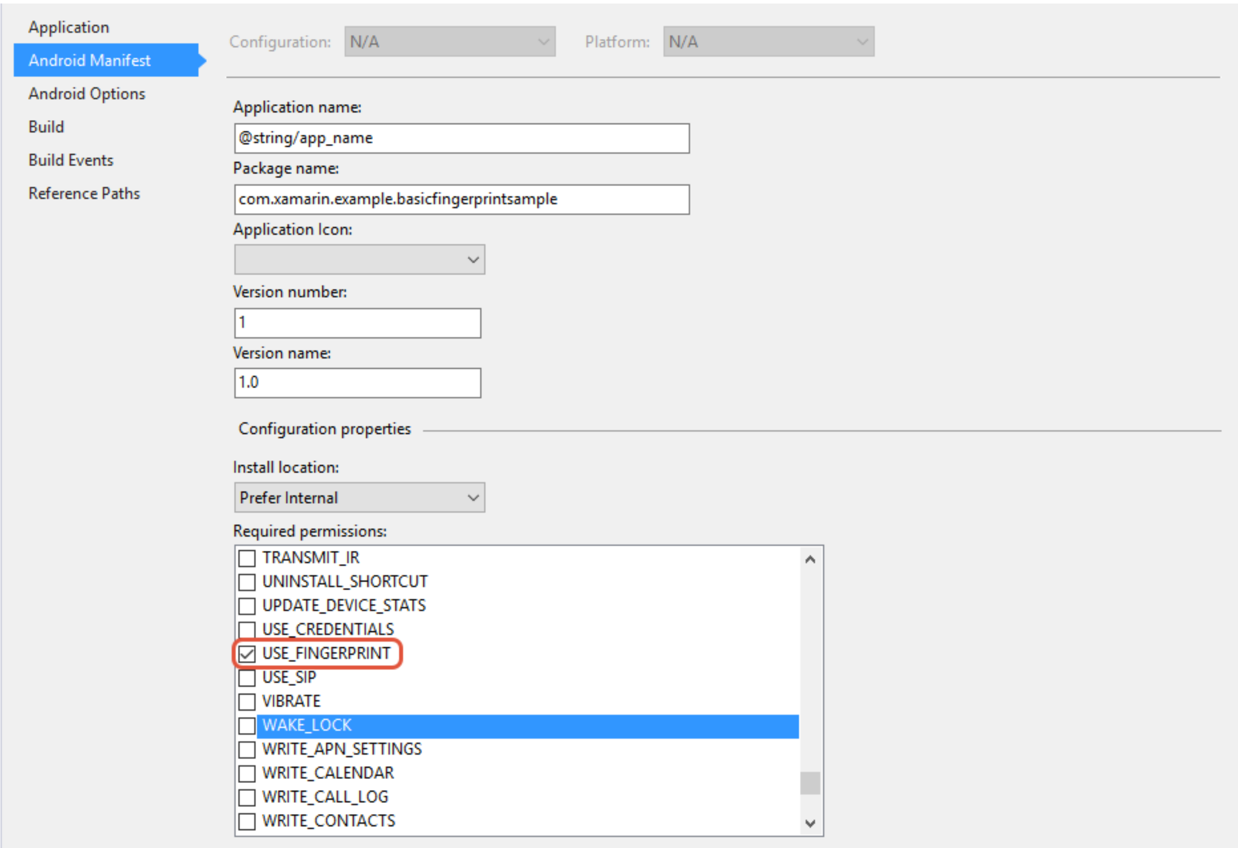
To get started, let's first cover how to configure a Xamarin.Android project so that the application is able to use fingerprint authentication:

1. Update **AndroidManifest.xml** to declare the permissions that the Fingerprint APIs require.
2. Obtain a reference to the `FingerprintManager` .
3. Check that the device is capable of fingerprint scanning.

Requesting Permissions in the Application Manifest

- Visual Studio
- Visual Studio for Mac

An Android application must request the `USE_FINGERPRINT` permission in the manifest. The following screenshot shows how to add this permission to the application in Visual Studio:



Getting an Instance of the FingerprintManager

Next, the application must get an instance of the `FingerprintManager` or the `FingerprintManagerCompat` class. To be compatible with older versions of Android, an Android application should use the compatibility API's found in the Android Support v4 NuGet package. The following snippet demonstrates how to get the appropriate object from the operating system:

```
// Using the Android Support Library v4
FingerprintManagerCompat fingerprintManager = FingerprintManagerCompat.From(context);

// Using API level 23:
FingerprintManager fingerprintManager = context.GetSystemService(Context.FingerprintService) as
FingerprintManager;
```

In the previous snippet, the `context` is any Android `Android.Content.Context`. Typically this is the Activity which is performing the authentication.

Checking for Eligibility

An application must perform several checks to ensure that it is possible to use fingerprint authentication. In total, there are five conditions that the application uses to check for eligibility:

API level 23 – The Fingerprint APIs require API level 23 or higher. The `FingerprintManagerCompat` class will wrap the API level check for you. For this reason it is recommend to use the **Android Support Library v4** and `FingerprintManagerCompat`; this will account for the one of these checks.

Hardware – When the application starts up for the first time, it should check for the presence of a fingerprint scanner:

```
FingerprintManagerCompat fingerprintManager = FingerprintManagerCompat.From(context);
if (!fingerprintManager.IsHardwareDetected)
{
    // Code omitted
}
```

Device Is Secured – The user must have the device secured with a screen lock. If the user has not secured the device with a screen lock and security is important to the application, then the user should be notified that a screen lock must be configured. The following code snippet shows how to check this pre-requiste:

```
KeyguardManager keyguardManager = (KeyguardManager) GetSystemService(KeyguardService);
if (!keyguardManager.IsKeyguardSecure)
{
}
```

Enrolled Fingerprints – The user must have at least one fingerprint registered with the operating system. This permission check should occur prior to each authentication attempt:

```
FingerprintManagerCompat fingerprintManager = FingerprintManagerCompat.From(context);
if (!fingerprintManager.HasEnrolledFingerprints)
{
    // Can't use fingerprint authentication - notify the user that they need to
    // enroll at least one fingerprint with the device.
}
```

Permissions – The application must request permission from the user before using the application. For Android 5.0 and lower, the user grants the permission as a condition of installing the app. Android 6.0 introduced a new permission model that checks permissions at run-time. This code snippet is an example of how to check for permissions on Android 6.0:

```
// The context is typically a reference to the current activity.
Android.Content.PM.Permission permissionResult = ContextCompat.CheckSelfPermission(context,
Manifest.Permission.UseFingerprint);
if (permissionResult == Android.Content.PM.Permission.Granted)
{
    // Permission granted - go ahead and start the fingerprint scanner.
}
else
{
    // No permission. Go and ask for permissions and don't start the scanner. See
    // https://developer.android.com/training/permissions/requesting.html
}
```

Checking all of these conditions each time the application offers authentication options will ensure the user gets the best user experience. Changes or upgrades to their device or operating system might affect the availability of fingerprint authentication. If you choose to cache the results of any of these checks, make sure to cater for upgrade scenarios.

For more information on how to request permissions in Android 6.0, consult the Android guide Requesting Permissions at Run-Time.

Related Links

- Context
- KeyguardManager
- ContextCompat
- FingerprintManager
- FingerprintManagerCompat
- Requesting Permissions at Run-Time

Scanning For Fingerprints

10/28/2019 • 3 minutes to read • Edit Online

Now that we have seen how to prepare a Xamarin.Android application to use fingerprint authentication, let's return to the `FingerprintManager.Authenticate` method, and discuss its place in the Android 6.0 fingerprint authentication. A quick overview of the workflow for fingerprint authentication is described in this list:

1. Invoke `FingerprintManager.Authenticate`, passing a `CryptoObject` and a `FingerprintManager.AuthenticationCallback` instance. The `CryptoObject` is used to ensure that the fingerprint authentication result was not tampered with.
2. Subclass the FingerprintManager.AuthenticationCallback class. An instance of this class will be provided to `FingerprintManager` when fingerprint authentication starts. When the fingerprint scanner is finished, it will invoke one of the callback methods on this class.
3. Write code to update the UI to let the user know that the device has started the fingerprint scanner and is waiting for user interaction.
4. When the fingerprint scanner is done, Android will return results to the application by invoking a method on the `FingerprintManager.AuthenticationCallback` instance that was provided in the previous step.
5. The application will inform the user of the fingerprint authentication results and react to the results as appropriate.

The following code snippet is an example of a method in an Activity that will start scanning for fingerprints:

```
protected void FingerPrintAuthenticationExample()
{
    const int flags = 0; /* always zero (0) */

    // CryptoObjectHelper is described in the previous section.
    CryptoObjectHelper cryptoHelper = new CryptoObjectHelper();

    // cancellationSignal can be used to manually stop the fingerprint scanner.
    cancellationSignal = new Android.Support.V4.OS.CancellationSignal();

    FingerprintManagerCompat fingerprintManager = FingerprintManagerCompat.From(this);

    // AuthenticationCallback is a base class that will be covered later on in this guide.
    FingerprintManagerCompat.AuthenticationCallback authenticationCallback = new MyAuthCallbackSample(this);

    // Start the fingerprint scanner.
    fingerprintManager.Authenticate(cryptoHelper.BuildCryptoObject(), flags, cancellationSignal,
authenticationCallback, null);
}
```

Let's discuss each of these parameters in the `Authenticate` method in a bit more detail:

- The first parameter is a *crypto* object that the fingerprint scanner will use to help authenticate the results of a fingerprint scan. This object may be `null`, in which case the application has to blindly trust that nothing has tampered with the fingerprint results. It is recommended that a `CryptoObject` be instantiated and provided to the `FingerprintManager` rather than null. Creating a CryptObject will explain in detail how to instantiate a `CryptoObject` based on a `Cipher`.
- The second parameter is always zero. The Android documentation identifies this as set of flags and is most likely reserved for future use.
- The third parameter, `cancellationSignal` is an object used to turn off the fingerprint scanner and cancel the

current request. This is an Android CancellationSignal, and not a type from the .NET framework.

- The fourth parameter is mandatory and is a class that subclasses the `AuthenticationCallback` abstract class. Methods on this class will be invoked to signal to clients when the `FingerprintManager` has finished and what the results are. As there is a lot to understand about implementing the `AuthenticationCallback`, it will be covered in it's own section.
- The fifth parameter is an optional `Handler` instance. If a `Handler` object is provided, the `FingerprintManager` will use the `Looper` from that object when processing the messages from the fingerprint hardware. Typically, one does not need to provide a `Handler`, the FingerprintManager will use the `Looper` from the application.

Cancelling a Fingerprint Scan

It might be necessary for the user (or the application) to cancel the fingerprint scan after it has been initiated. In this situation, invoke the `IsCancelled` method on the `CancellationSignal` that was provided to `FingerprintManager.Authenticate` when it was invoked to start the fingerprint scan.

Now that we have seen the `Authenticate` method, let's examine some of the more important parameters in more detail. First, we'll look at Responding to Authentication Callbacks, which will discuss how to subclass the FingerprintManager.AuthenticationCallback, enabling an Android application to react to the results provided by the fingerprint scanner.

Related Links

- CancellationSignal
- FingerprintManager.AuthenticationCallback
- FingerprintManager.CryptoObject
- FingerprintManagerCompat.CryptoObject
- FingerprintManager
- FingerprintManagerCompat

Creating a CryptoObject

12/13/2019 • 4 minutes to read • Edit Online

The integrity of the fingerprint authentication results is important to an application – it is how the application knows the identity of the user. It is theoretically possible for third-party malware to intercept and tamper with the results returned by the fingerprint scanner. This section will discuss one technique for preserving the validity of the fingerprint results.

The `FingerprintManager.CryptoObject` is a wrapper around the Java cryptography APIs and is used by the `FingerprintManager` to protect the integrity of the authentication request. Typically, a `Javax.Crypto.Cipher` object is the mechanism for encrypting the results of the fingerprint scanner. The `Cipher` object itself will use a key that is created by the application using the Android keystore APIs.

To understand how these classes all work together, let's first look at the following code which demonstrates how to create a `CryptoObject`, and then explain in more detail:

```
public class CryptoObjectHelper
{
    // This can be key name you want. Should be unique for the app.
    static readonly string KEY_NAME = "com.xamarin.android.sample.fingerprint_authentication_key";

    // We always use this keystore on Android.
    static readonly string KEYSTORE_NAME = "AndroidKeyStore";

    // Should be no need to change these values.
    static readonly string KEY_ALGORITHM = KeyProperties.KeyAlgorithmAes;
    static readonly string BLOCK_MODE = KeyProperties.BlockModeCbc;
    static readonly string ENCRYPTION_PADDING = KeyProperties.EncryptionPaddingPkcs7;
    static readonly string TRANSFORMATION = KEY_ALGORITHM + "/" +
                                            BLOCK_MODE + "/" +
                                            ENCRYPTION_PADDING;
    readonly KeyStore _keystore;

    public CryptoObjectHelper()
    {
        _keystore = KeyStore.GetInstance(KEYSTORE_NAME);
        _keystore.Load(null);
    }

    public FingerprintManagerCompat.CryptoObject BuildCryptoObject()
    {
        Cipher cipher = CreateCipher();
        return new FingerprintManagerCompat.CryptoObject(cipher);
    }

    Cipher CreateCipher(bool retry = true)
    {
        IKey key = GetKey();
        Cipher cipher = Cipher.GetInstance(TRANSFORMATION);
        try
        {
            cipher.Init(CipherMode.EncryptMode, key);
        } catch(KeyPermanentlyInvalidatedException e)
        {
            _keystore.DeleteEntry(KEY_NAME);
            if(retry)
            {
                CreateCipher(false);
            } else
            {
```

```
            {
                throw new Exception("Could not create the cipher for fingerprint authentication.", e);
            }
        }
        return cipher;
    }

    IKey GetKey()
    {
        IKey secretKey;
        if(!_keystore.IsKeyEntry(KEY_NAME))
        {
            CreateKey();
        }

        secretKey = _keystore.GetKey(KEY_NAME, null);
        return secretKey;
    }

    void CreateKey()
    {
        KeyGenerator keyGen = KeyGenerator.GetInstance(KeyProperties.KeyAlgorithmAes, KEYSTORE_NAME);
        KeyGenParameterSpec keyGenSpec =
            new KeyGenParameterSpec.Builder(KEY_NAME, KeyStorePurpose.Encrypt | KeyStorePurpose.Decrypt)
                .SetBlockModes(BLOCK_MODE)
                .SetEncryptionPaddings(ENCRYPTION_PADDING)
                .SetUserAuthenticationRequired(true)
                .Build();
        keyGen.Init(keyGenSpec);
        keyGen.GenerateKey();
    }
}
```

The sample code will create a new `Cipher` for each `CryptoObject`, using a key that was created by the application. The key is identified by the `KEY_NAME` variable that was set in the beginning of the `CryptoObjectHelper` class. The method `GetKey` will try and retrieve the key using the Android Keystore APIs. If the key does not exist, then the method `CreateKey` will create a new key for the application.

The cipher is instantiated with a call to `Cipher.GetInstance`, taking a *transformation* (a string value that tells the cipher how to encrypt and decrypt data). The call to `Cipher.Init` will complete the initialization of the cipher by providing a key from the application.

It is important to realize that there are some situations where Android may invalidate the key:

- A new fingerprint has been enrolled with the device.
- There are no fingerprints enrolled with the device.
- The user has disabled the screen lock.
- The user has changed the screen lock (the type of the screenlock or the PIN/pattern used).

When this happens, `Cipher.Init` will throw a `KeyPermanentlyInvalidatedException`. The above sample code will trap that exception, delete the key, and then create a new one.

The next section will discuss how to create the key and store it on the device.

Creating a Secret Key

The `CryptoObjectHelper` class uses the Android `KeyGenerator` to create a key and store it on the device. The `KeyGenerator` class can create the key, but needs some meta-data about the type of key to create. This information is provided by an instance of the `KeyGenParameterSpec` class.

A `KeyGenerator` is instantiated using the `GetInstance` factory method. The sample code uses the *Advanced Encryption Standard* (*AES*) as the encryption algorithm. AES will break the data up into blocks of a fixed size and encrypt each of those blocks.

Next, a `KeyGenParameterSpec` is created using the `KeyGenParameterSpec.Builder`. The `KeyGenParameterSpec.Builder` wraps the following information about the key that is to be created:

- The name of the key.
- The key must be valid for both encrypting and decrypting.
- In the sample code the `BLOCK_MODE` is set to *Cipher Block Chaining* (`KeyProperties.BlockModeCbc`), meaning that each block is XORed with the previous block (creating dependencies between each block).
- The `CryptoObjectHelper` uses *Public Key Cryptography Standard #7* (*PKCS7*) to generate the bytes that will pad out the blocks to ensure that they are all of the same size.
- `SetUserAuthenticationRequired(true)` means that user authentication is required before the key can be used.

Once the `KeyGenParameterSpec` is created, it is used to initialize the `KeyGenerator`, which will generate a key and securely store it on the device.

Using the CryptoObjectHelper

Now that the sample code has encapsulated much of the logic for creating a `CryptoWrapper` into the `CryptoObjectHelper` class, let's revisit the code from the start of this guide and use the `CryptoObjectHelper` to create the Cipher and start a fingerprint scanner:

```
protected void FingerPrintAuthenticationExample()
{
    const int flags = 0; /* always zero (0) */

    CryptoObjectHelper cryptoHelper = new CryptoObjectHelper();
    cancellationSignal = new Android.Support.V4.OS.CancellationSignal();

    // Using the Support Library classes for maximum reach
    FingerprintManagerCompat fingerPrintManager = FingerprintManagerCompat.From(this);

    // AuthCallbacks is a C# class defined elsewhere in code.
    FingerprintManagerCompat.AuthenticationCallback authenticationCallback = new MyAuthCallbackSample(this);

    // Here is where the CryptoObjectHelper builds the CryptoObject.
    fingerprintManager.Authenticate(cryptohelper.BuildCryptoObject(), flags, cancellationSignal,
authenticationCallback, null);
}
```

Now that we have seen how to create a `CryptoObject`, lets move on to see how the `FingerprintManager.AuthenticationCallbacks` are used to transfer the results of fingerprint scanner service to an Android application.

Related Links

- Cipher
- FingerprintManager.CryptoObject
- FingerprintManagerCompat.CryptoObject
- KeyGenerator
- KeyGenParameterSpec
- KeyGenParameterSpec.Builder
- KeyPermanentlyInvalidatedException
- KeyProperties
- AES

- RFC 2315 - PCKS #7

Responding to Authentication Callbacks

10/28/2019 • 5 minutes to read • Edit Online

The fingerprint scanner runs in the background on its own thread, and when it is finished it will report the results of the scan by invoking one method of `FingerprintManager.AuthenticationCallback` on the UI thread. An Android application must provide its own handler which extends this abstract class, implementing all the following methods:

- `OnAuthenticationError(int errorCode, ICharSequence errString)` – Called when there is an unrecoverable error. There is nothing more an application or user can do to correct the situation except possibly try again.
- `OnAuthenticationFailed()` – This method is invoked when a fingerprint has been detected but not recognized by the device.
- `OnAuthenticationHelp(int helpMsgId, ICharSequence helpString)` – Called when there is a recoverable error, such as the finger being swiped to fast over the scanner.
- `OnAuthenticationSucceeded(FingerprintManagerCompati.AuthenticationResult result)` – This is called when a fingerprint has been recognized.

If a `CryptoObject` was used when calling `Authenticate`, it is recommended to call `Cipher.DoFinal` in `OnAuthenticationSuccessful`. `DoFinal` will throw an exception if the cipher was tampered with or improperly initialized, indicating that the result of the fingerprint scanner may have been tampered with outside of the application.

> **NOTE**
>
> It is recommended to keep the callback class relatively light weight and free of application specific logic. The callbacks should act as a "traffic cop" between the Android application and the results from the fingerprint scanner.

A Sample Authentication Callback Handler

The following class is an example of a minimal `FingerprintManager.AuthenticationCallback` implementation:

```
class MyAuthCallbackSample : FingerprintManagerCompat.AuthenticationCallback
{
    // Can be any byte array, keep unique to application.
    static readonly byte[] SECRET_BYTES = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    // The TAG can be any string, this one is for demonstration.
    static readonly string TAG = "X:" + typeof (SimpleAuthCallbacks).Name;

    public MyAuthCallbackSample()
    {
    }

    public override void OnAuthenticationSucceeded(FingerprintManagerCompat.AuthenticationResult result)
    {
        if (result.CryptoObject.Cipher != null)
        {
            try
            {
                // Calling DoFinal on the Cipher ensures that the encryption worked.
                byte[] doFinalResult = result.CryptoObject.Cipher.DoFinal(SECRET_BYTES);

                // No errors occurred, trust the results.
            }
            catch (BadPaddingException bpe)
            {
                // Can't really trust the results.
                Log.Error(TAG, "Failed to encrypt the data with the generated key." + bpe);
            }
            catch (IllegalBlockSizeException ibse)
            {
                // Can't really trust the results.
                Log.Error(TAG, "Failed to encrypt the data with the generated key." + ibse);
            }
        }
        else
        {
            // No cipher used, assume that everything went well and trust the results.
        }
    }

    public override void OnAuthenticationError(int errMsgId, ICharSequence errString)
    {
        // Report the error to the user. Note that if the user canceled the scan,
        // this method will be called and the errMsgId will be FingerprintState.ErrorCanceled.
    }

    public override void OnAuthenticationFailed()
    {
        // Tell the user that the fingerprint was not recognized.
    }

    public override void OnAuthenticationHelp(int helpMsgId, ICharSequence helpString)
    {
        // Notify the user that the scan failed and display the provided hint.
    }
}
```

`OnAuthenticationSucceeded` checks to see if a `Cipher` was provided to `FingerprintManager` when `Authentication` was invoked. If so, the `DoFinal` method is called on the cipher. This closes the `Cipher`, restoring it to its original state. If there was a problem with the cipher, then `DoFinal` will throw an exception and the authentication attempt should be considered to have failed.

The `OnAuthenticationError` and `OnAuthenticationHelp` callbacks each receive an integer indicating what the problem was. The following section explains each of the possible help or error codes. The two callbacks serve similar purposes – to inform the application that fingerprint authentication has failed. How they differ is in severity. `OnAuthenticationHelp` is a user recoverable error, such as swiping the fingerprint too fast; `OnAuthenticationError` is more a severe error, such as a damaged fingerprint scanner.

Note that `OnAuthenticationError` will be invoked when the fingerprint scan is cancelled via the `CancellationSignal.Cancel()` message. The `errMsgId` parameter will have the value of 5 (`FingerprintState.ErrorCanceled`). Depending on the requirements, an implementation of the `AuthenticationCallbacks` may treat this situation differently than the other errors.

`OnAuthenticationFailed` is invoked when the fingerprint was successfully scanned but did not match any fingerprint enrolled with the device.

Help Codes and Error Message Ids

A list and description of the error codes and help codes may be found in the Android SDK documentation for the FingerprintManager class. Xamarin.Android represents these values with the `Android.Hardware.Fingerprints.FingerprintState` enum:

- `AcquiredGood` – (value 0) The image acquired was good.
- `AcquiredImagerDirty` – (value 3) The fingerprint image was too noisy due to suspected or detected dirt on the sensor. For example, it's reasonable to return this after multiple `AcquiredInsufficient` or actual detection of dirt on the sensor (stuck pixels, swaths, etc.). The user is expected to take action to clean the sensor when this is returned.
- `AcquiredInsufficient` – (value 2) The fingerprint image was too noisy to process due to a detected condition (i.e. dry skin) or a possibly dirty sensor (See `AcquiredImagerDirty`.
- `AcquiredPartial` – (value 1) Only a partial fingerprint image was detected. During enrollment, the user should be informed on what needs to happen to resolve this problem, e.g., "press firmly on sensor."
- `AcquiredTooFast` – (value 5) The fingerprint image was incomplete due to quick motion. While mostly appropriate for linear array sensors, this could also happen if the finger was moved during acquisition. The user should be asked to move the finger slower (linear) or leave the finger on the sensor longer.
- `AcquiredToSlow` – (value 4) The fingerprint image was unreadable due to lack of motion. This is most appropriate for linear array sensors that require a swipe motion.
- `ErrorCanceled` – (value 5) The operation was canceled because the fingerprint sensor is unavailable. For example, this may happen when the user is switched, the device is locked, or another pending operation prevents or disables it.
- `ErrorHwUnavailable` – (value 1) The hardware is unavailable. Try again later.
- `ErrorLockout` – (value 7) The operation was canceled because the API is locked out due to too many attempts.
- `ErrorNoSpace` – (value 4) Error state returned for operations like enrollment; the operation cannot be completed because there's not enough storage remaining to complete the operation.
- `ErrorTimeout` – (value 3) Error state returned when the current request has been running too long. This is intended to prevent programs from waiting for the fingerprint sensor indefinitely. The timeout is platform and sensor-specific, but is generally about 30 seconds.
- `ErrorUnableToProcess` – (value 2) Error state returned when the sensor was unable to process the current image.

Related Links

- Cipher
- AuthenticationCallback
- AuthenticationCallback

Fingerprint Authentication Guidance

7/8/2021 • 2 minutes to read • Edit Online

Fingerprint Authentication Guidance

Now that we have seen the concepts and APIs surrounding Android 6.0 fingerprint authentication, let's discuss some general advice for the use of the Fingerprint APIs.

1. **Use the Android Support Library v4 Compatibility APIs** – This will simplify the application code by removing the API check from the code and allow an application to target the most devices possible.
2. **Provide Alternatives to Fingerprint Authentication** – Fingerprint authentication is a great, quick way for an application to authenticate a user, however, it cannot be assumed that it will always work or be available. It is possible that the fingerprint scanner may fail, the lens maybe be dirty, the user may not have configured the device to use fingerprint authentication, or the fingerprints have since gone missing. It is also possible that the user may not wish to use fingerprint authentication with your application. For these reasons, an Android application should provide an alternate authentication process such as username and password.
3. **Use Google's fingerprint icon** – All applications should use the same fingerprint icon provided by Google. The use of a standard icon makes it easy for Android users to recognize where in apps fingerprint authentication is used:

4. **Notify the User** – An application should display some kind of notification to the user that the fingerprint scanner is active and awaiting a touch or swipe.

Summary

Fingerprint authentication is a great way to allow a Xamarin.Android application to quickly verify users, making it easier for users to interact with sensitive features such as in-app purchases. This guide discussed the concepts and code that is required to incorporate the Android 6.0 fingerprint API's in your Xamarin.Android application.

First we discussed the fingerprint API's themselves, `FingerprintManager` (and `FingerprintManagerCompat`). We examined how the `FingerprintManager.AuthenticationCallbacks` abstract class must be extended by an application and used as an intermediary between the fingerprint hardware and the application itself. Then we examined how to verify the integrity of the fingerprint scanner results using a Java `Cipher` object. Finally, we touched a bit on testing by describing how to enroll a fingerprint on a device and using **adb** to simulate a fingerprint swipe on an emulator.

If you haven't already done so, you should look at the sample application that accompanies this guide. The Fingerprint Dialog Sample has been ported from Java to Xamarin.Android and provides another example on how to add fingerprint authentication to an Android application.

Related Links

- Fingerprint Guide Sample App
- Fingerprint Dialog Sample
- Fingerprint Icon

Enrolling a Fingerprint

7/8/2021 • 2 minutes to read • Edit Online

Enrolling a Fingerprint Overview

It is only possible for an Android application to leverage fingerprint authentication if the device has already been configured with fingerprint authentication. This guide will discuss how to enroll a fingerprint on an Android device or Emulator. Emulators do not have the actual hardware to perform a fingerprint scan, but it is possible to simulate a fingerprint scan with the help of the Android Debug Bridge (described below). This guide will discuss how to enable screen lock on an Android device and enroll a fingerprint for authentication.

Requirements

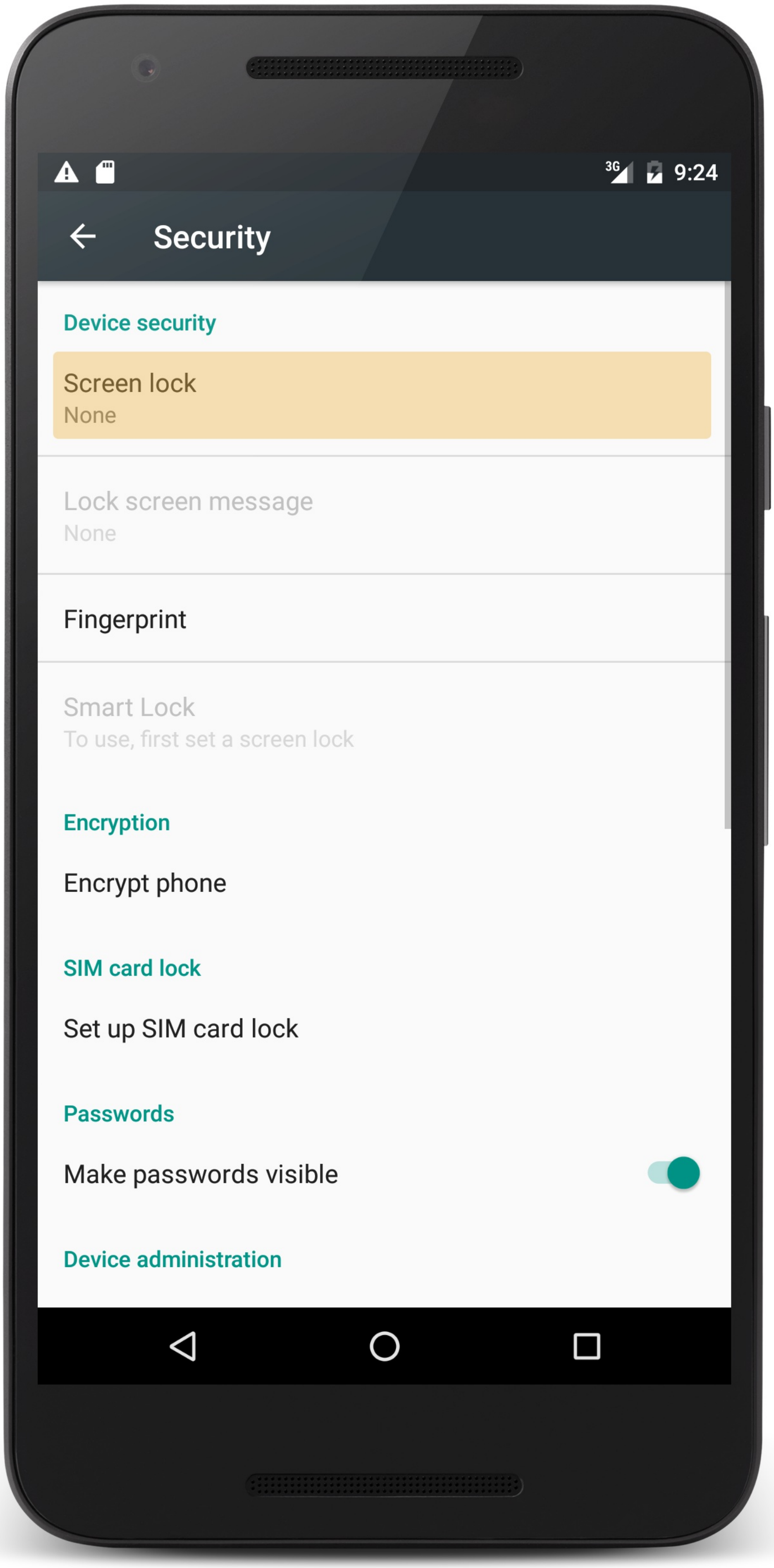
To enroll a fingerprint, you must have an Android device or an emulator running API level 23 (Android 6.0).

The use of the Android Debug Bridge (ADB) requires familiarity with the command prompt, and the **adb** executable must be in the PATH of your Bash, PowerShell, or Command Prompt environment.

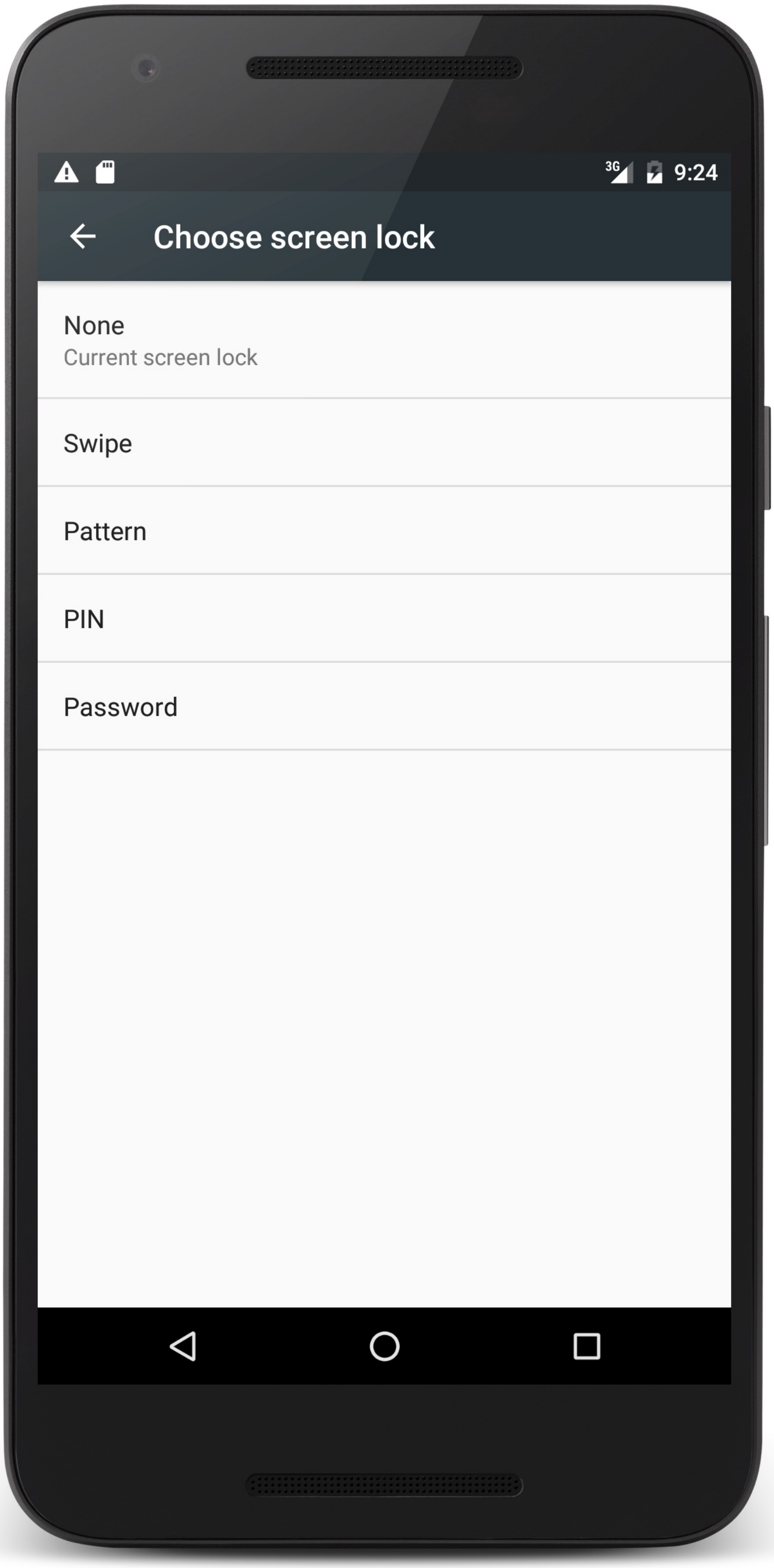
Configuring a Screen Lock and Enrolling a Fingerprint

To setup a screen lock, perform the following steps:

1. Go to **Settings > Security**, and select **Screen lock**:

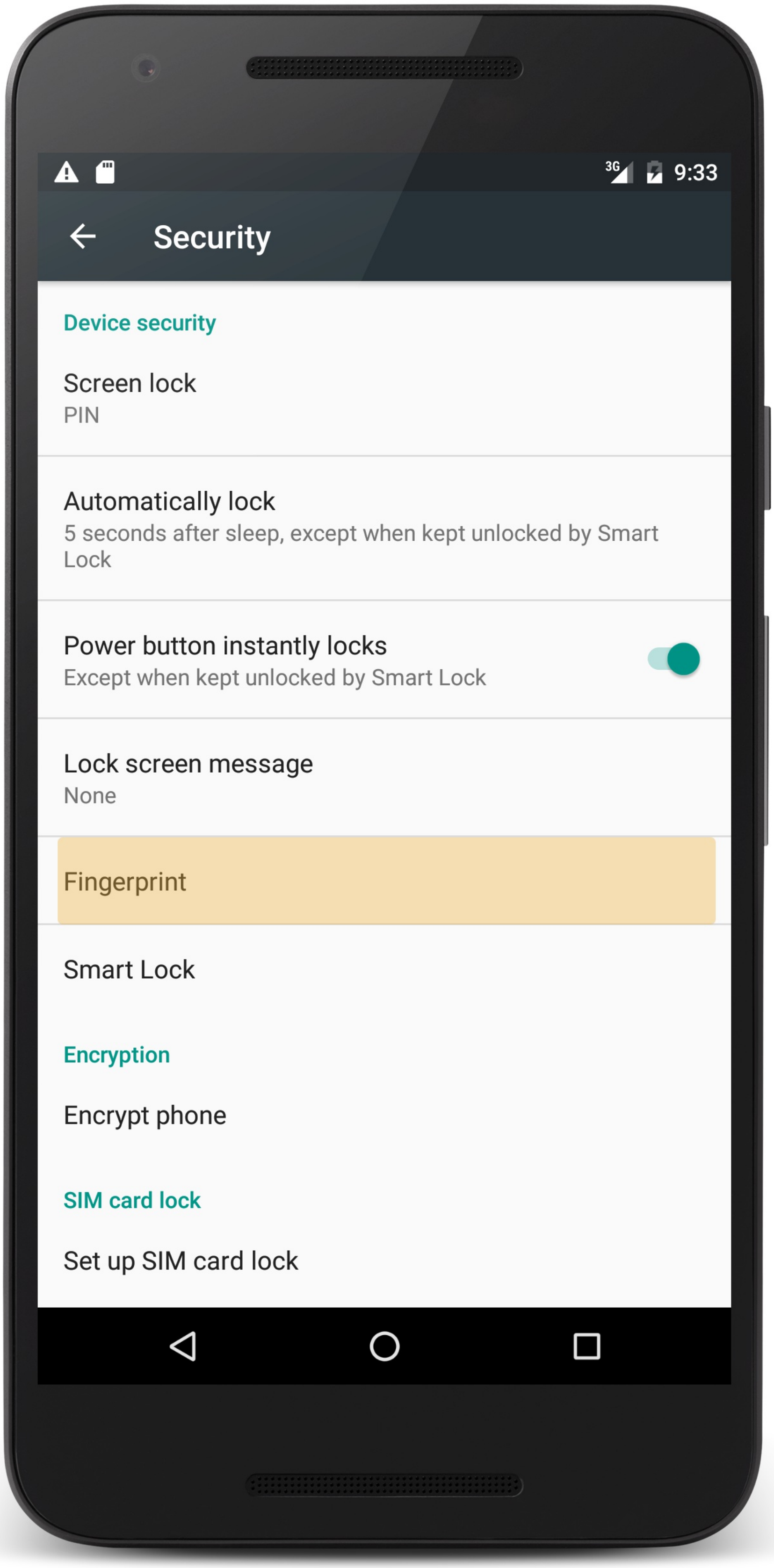


2. The next screen that appears will allow you select and configure one of the screen lock security methods:

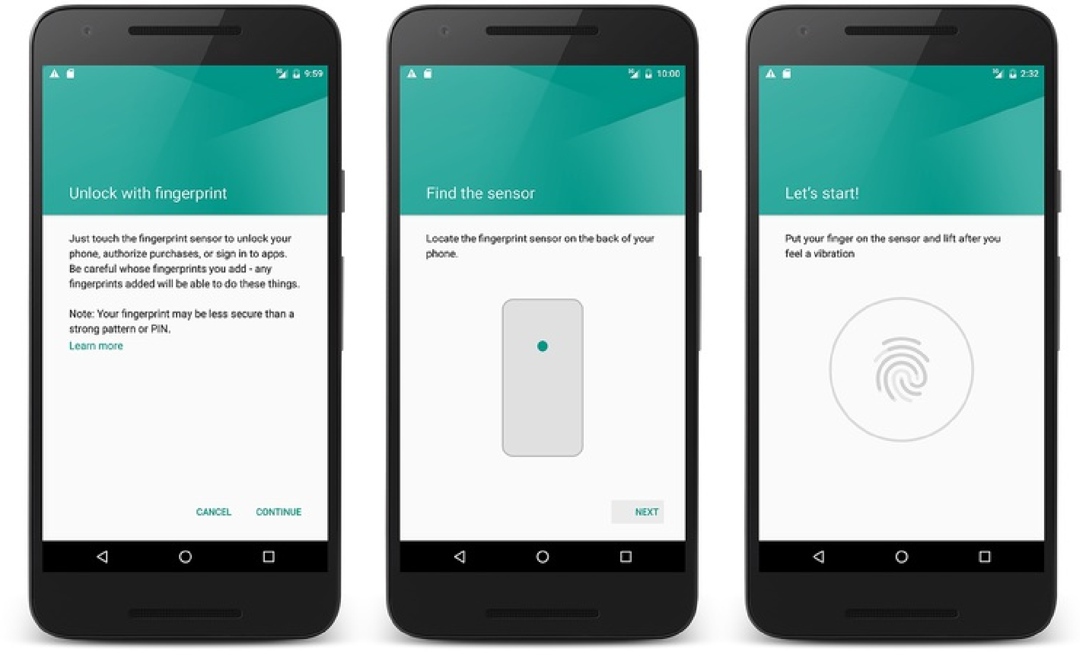


Select and complete one of the available screen lock methods.

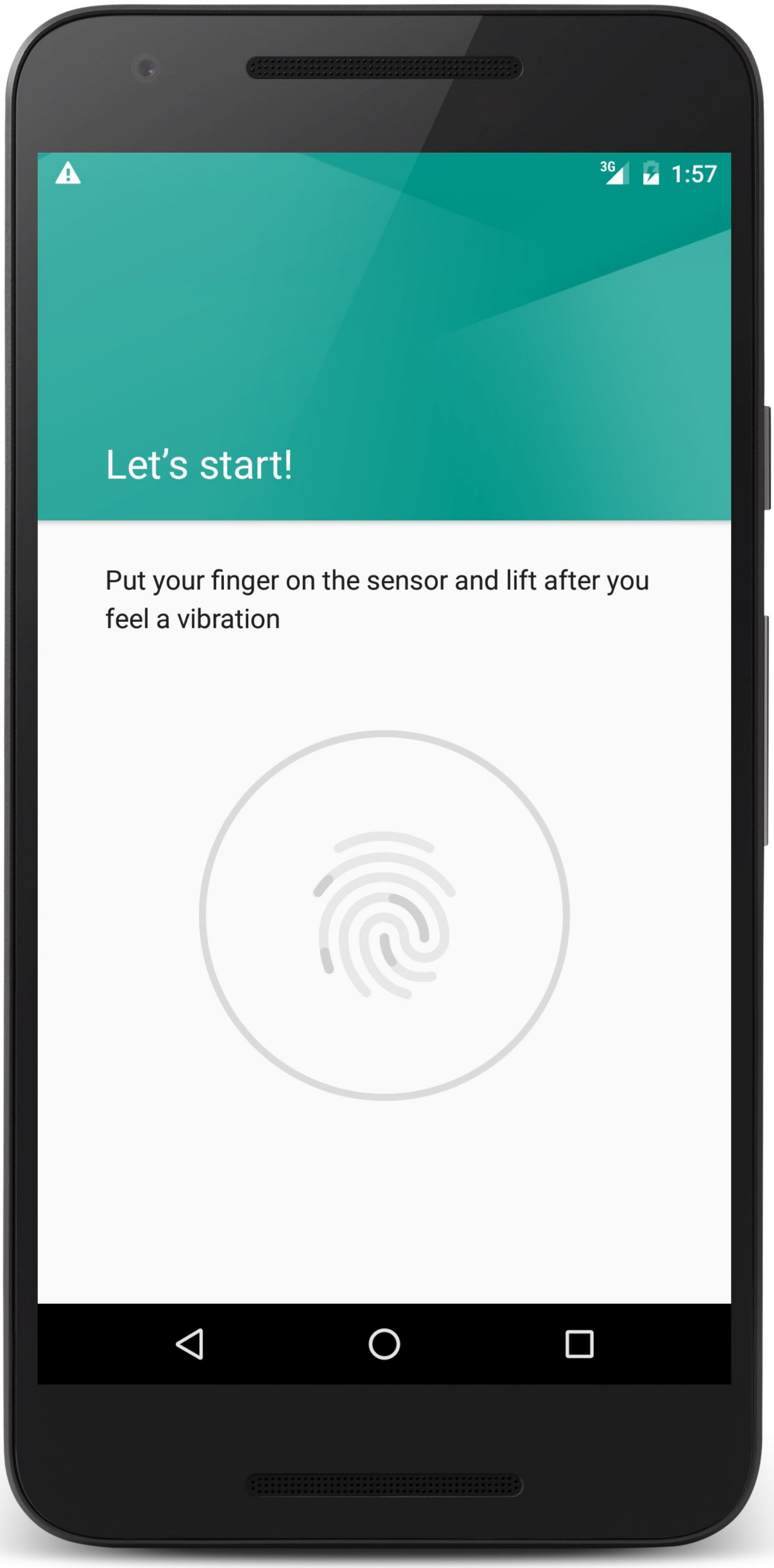
3. Once the screenlock is configured, return to the **Settings > Security** page and select **Fingerprint**:



4. From there, follow the sequence to add a fingerprint to the device:



5. In the final screen you are prompted to place your finger on the fingerprint scanner:



If you are using an Android device, complete the process by touching a finger to the scanner.

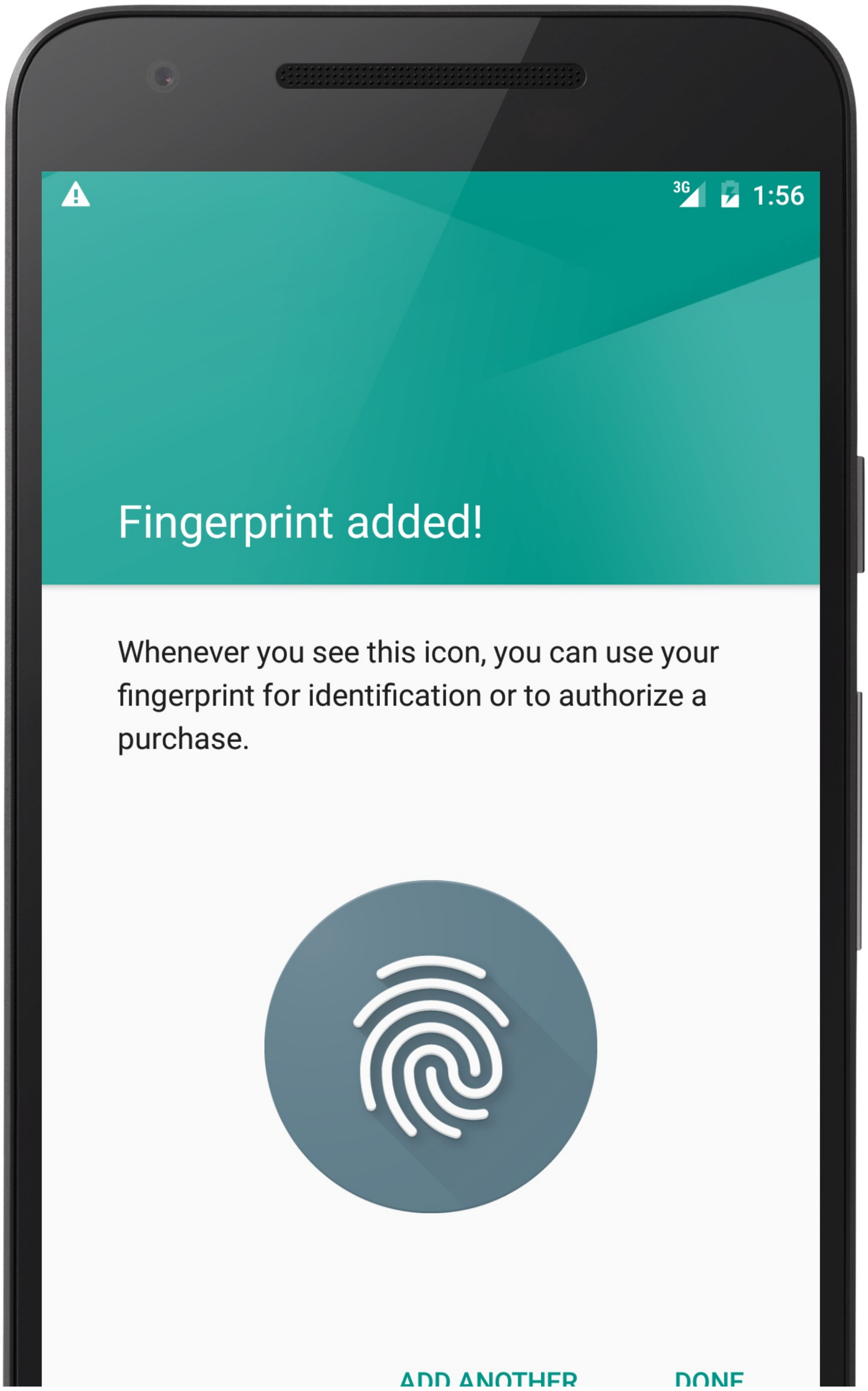
Simulating a Fingerprint Scan on the Emulator

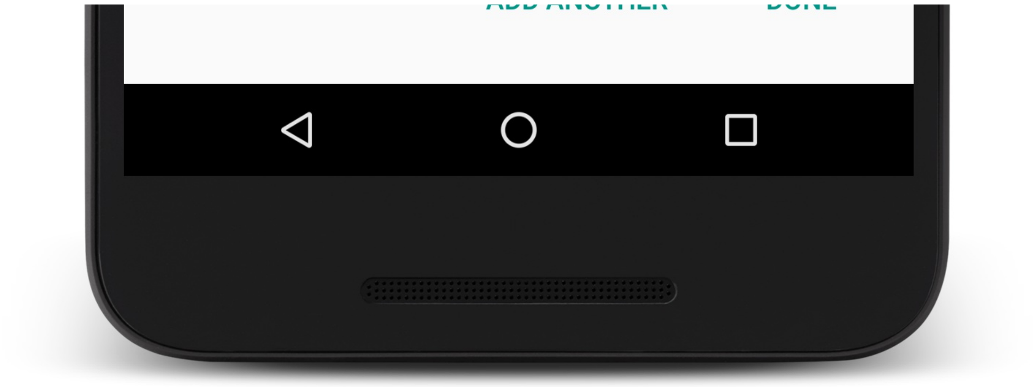
On an Android emulator, it is possible to simulate a fingerprint scan by using the Android Debug Bridge. On OS X start a Terminal session while on Windows start a command prompt or a Powershell session and run `adb`:

```
$ adb -e emu finger touch 1
```

The value of **1** is the *finger_id* for the finger that was "scanned". It is a unique integer that you assign for each virtual fingerprint. In the future when the app is running you can run this same ADB command each time the emulator prompts you for a fingerprint, you can run the `adb` command and pass it the *finger_id* to simulate the fingerprint scan.

After the fingerprint scan is complete, Android will notify you that the fingerprint has been added:





Summary

This guide covered how to setup a screen lock and enroll a fingerprint on an Android device or in an Android emulator.

Android Job Scheduler

1/31/2020 • 10 minutes to read • Edit Online

This guide discusses how to schedule background work using the Android Job Scheduler API, which is available on Android devices running Android 5.0 (API level 21) and higher.

Overview

One of the best ways to keep an Android application responsive to the user is to ensure that complex or long running work is performed in the background. However, it is important that background work will not negatively impact the user's experience with the device.

For example, a background job might poll a website every three or four minutes to query for changes to a particular dataset. This seems benign, however it would have a disastrous impact on battery life. The application will repeatedly wake up the device, elevate the CPU to a higher power state, power up the radios, make the network requests, and then processing the results. It gets worse because the device will not immediately power down and return to the low-power idle state. Poorly scheduled background work may inadvertently keep the device in a state with unnecessary and excessive power requirements. This seemingly innocent activity (polling a website) will render the device unusable in a relatively short period of time.

Android provides the following APIs to help with performing work in the background but by themselves they are not sufficient for intelligent job scheduling.

- Intent Services – Intent Services are great for performing the work, however they provide no way to schedule work.
- AlarmManager – These APIs only allow work to be scheduled but provide no way to actually perform the work. Also, the AlarmManager only allows time based constraints, which means raise an alarm at a certain time or after a certain period of time has elapsed.
- Broadcast Receivers – An Android app can setup broadcast receivers to perform work in response to system-wide events or Intents. However, broadcast receivers don't provide any control over when the job should be run. Also changes in the Android operating system will restrict when broadcast receivers will work, or the kinds of work that they can respond to.

There are two key features to efficiently performing background work (sometimes referred to as a *background job* or a *job*):

1. **Intelligently scheduling the work** – It is important that when an application is doing work in the background that it does so as a good citizen. Ideally, the application should not demand that a job be run. Instead, the application should specify conditions that must be met for when the job can run, and then schedule that job with the operating system that will perform the work when the conditions are met. This allows Android to run the job to ensure maximum efficiency on the device. For example, network requests may be batched to run all at the same time to make maximum use of overhead involved with networking.
2. **Encapsulating the work** – The code to perform the background work should be encapsulated in a discrete component that can be run independently of the user interface and will be relatively easy to reschedule if the work fails to complete for some reason.

The Android Job Scheduler is a framework built in to the Android operating system that provides a fluent API to simplify scheduling background work. The Android Job Scheduler consists of the following types:

- The `Android.App.Job.JobScheduler` is a system service that is used to schedule, execute, and if necessary cancel, jobs on behalf of an Android application.

- An `Android.App.Job.JobService` is an abstract class that must be extended with the logic that will run the job on the main thread of the application. This means that the `JobService` is responsible for how the work is to be performed asynchronously.
- An `Android.App.Job.JobInfo` object holds the criteria to guide Android when the job should run.

To schedule work with the Android Job Scheduler, a Xamarin.Android application must encapsulate the code in a class that extends the `JobService` class. `JobService` has three lifecycle methods that can be called during the lifetime of the job:

- **bool OnStartJob(JobParameters parameters)** – This method is called by the `JobScheduler` to perform work, and runs on the main thread of the application. It is the responsibility of the `JobService` to asynchronously perform the work and return `true` if there is work remaining, or `false` if the work is done.

  When the `JobScheduler` calls this method, it will request and retain a wakelock from Android for the duration of the job. When the job is finished, it is the responsibility of the `JobService` to tell the `JobScheduler` of this fact by call the `JobFinished` method (described next).

- **JobFinished(JobParameters parameters, bool needsReschedule)** – This method must be called by the `JobService` to tell the `JobScheduler` that the work is done. If `JobFinished` is not called, the `JobScheduler` will not remove the wakelock, causing unnecessary battery drain.

- **bool OnStopJob(JobParameters parameters)** – This is called when the job is prematurely stopped by Android. It should return `true` if the job should be rescheduled based on the retry criteria (discussed below in more detail).

It is possible to specify *constraints* or *triggers* that will control when a job can or should run. For example, it is possible to constrain a job so that it will only run when the device is charging or to start a job when a picture is taken.

This guide will discuss in detail how to implement a `JobService` class and schedule it with the `JobScheduler`.

Requirements

The Android Job Scheduler requires Android API level 21 (Android 5.0) or higher.

Using the Android Job Scheduler

There are three steps for using the Android JobScheduler API:

1. Implement a JobService type to encapsulate the work.
2. Use a `JobInfo.Builder` object to create the `JobInfo` object that will hold the criteria for the `JobScheduler` to run the job.
3. Schedule the job using `JobScheduler.Schedule`.

Implement a JobService

All work performed by the Android Job Scheduler library must be done in a type that extends the `Android.App.Job.JobService` abstract class. Creating a `JobService` is very similar to creating a `Service` with the Android framework:

1. Extend the `JobService` class.
2. Decorate the subclass with the `ServiceAttribute` and set the `Name` parameter to a string that is made up of the package name and the name of the class (see the following example).
3. Set the `Permission` property on the `ServiceAttribute` to the string `android.permission.BIND_JOB_SERVICE`.
4. Override the `OnStartJob` method, adding the code to perform the work. Android will invoke this method on

the main thread of the application to run the job. Work that will take longer that a few milliseconds should be performed on a thread to avoid blocking the application.

5. When the work is done, the `JobService` must call the `JobFinished` method. This method is how `JobService` tells the `JobScheduler` that work is done. Failure to call `JobFinished` will result in the `JobService` putting unnecessary demands on the device, shortening the battery life.
6. It is a good idea to also override the `OnStopJob` method. This method is called by Android when the job is being shut down before it is finished and provides the `JobService` with an opportunity to properly dispose of any resources. This method should return `true` if it is necessary to reschedule the job, or `false` if it is not desirable to re-run the job.

The following code is an example of the simplest `JobService` for an application, using the TPL to asynchronously perform some work:

```
[Service(Name = "com.xamarin.samples.downloadscheduler.DownloadJob",
         Permission = "android.permission.BIND_JOB_SERVICE")]
public class DownloadJob : JobService
{
    public override bool OnStartJob(JobParameters jobParams)
    {
        Task.Run(() =>
        {
            // Work is happening asynchronously

            // Have to tell the JobScheduler the work is done.
            JobFinished(jobParams, false);
        });

        // Return true because of the asynchronous work
        return true;
    }

    public override bool OnStopJob(JobParameters jobParams)
    {
        // we don't want to reschedule the job if it is stopped or cancelled.
        return false;
    }
}
```

Creating a JobInfo to schedule a job

Xamarin.Android applications do not instantiate a `JobService` directly, instead they will pass a `JobInfo` object to the `JobScheduler`. The `JobScheduler` will instantiate the requested `JobService` object, scheduling and running the `JobService` according to the metadata in the `JobInfo`. A `JobInfo` object must contain the following information:

- **JobId** – this is an `int` value that is used to identify a job to the `JobScheduler`. Reusing this value will update any existing jobs. The value must be unique for the application.
- **JobService** – this parameter is a `ComponentName` that explicitly identifies the type that the `JobScheduler` should use to run a job.

This extension method demonstrates how to create a `JobInfo.Builder` with an Android `Context`, such as an Activity:

```
public static class JobSchedulerHelpers
{
    public static JobInfo.Builder CreateJobBuilderUsingJobId<T>(this Context context, int jobId) where
T:JobService
    {
        var javaClass = Java.Lang.Class.FromType(typeof(T));
        var componentName = new ComponentName(context, javaClass);
        return new JobInfo.Builder(jobId, componentName);
    }
}

// Sample usage - creates a JobBuilder for a DownloadJob and sets the Job ID to 1.
var jobBuilder = this.CreateJobBuilderUsingJobId<DownloadJob>(1);

var jobInfo = jobBuilder.Build();  // creates a JobInfo object.
```

A powerful feature of the Android Job Scheduler is the ability to control when a job runs or under what conditions a job may run. The following table describes some of the methods on `JobInfo.Builder` that allow an app to influence when a job can run:

METHOD	DESCRIPTION
`SetMinimumLatency`	Specifies that a delay (in milliseconds) that should be observed before a job is run.
`SetOverridingDeadline`	Declares the that the job must run before this time (in milliseconds) has elapsed.
`SetRequiredNetworkType`	Specifies the network requirements for a job.
`SetRequiresBatteryNotLow`	The job may only run when the device is not displaying a "low battery" warning to the user.
`SetRequiresCharging`	The job may only run when the battery is charging.
`SetDeviceIdle`	The job will run when the device is busy.
`SetPeriodic`	Specifies that the job should be regularly run.
`SetPersisted`	The job should perisist across device reboots.

The `SetBackoffCriteria` provides some guidance on how long the `JobScheduler` should wait before trying to run a job again. There are two parts to the backoff criteria: a delay in milliseconds (default value of 30 seconds)and type of back off that should be used (sometimes referred to as the *backoff policy* or the *retry policy*). The two policies are encapsulated in the `Android.App.Job.BackoffPolicy` enum:

- `BackoffPolicy.Exponential` – An exponential backoff policy will increase the initial backoff value exponentially after each failure. The first time a job fails, the library will wait the initial interval that is specified before rescheduling the job – example 30 seconds. The second time the job fails, the library will wait at least 60 seconds before trying to run the job. After the third failed attempt, the library will wait 120 seconds, and so on. This is the default value.
- `BackoffPolicy.Linear` – This strategy is a linear backoff that the job should be rescheduled to run at set intervals (until it succeeds). Linear backoff is best suited for work that must be completed as soon as possible or for problems that will quickly resolve themselves.

For more details on create a `JobInfo` object, please read Google's documentation for the `JobInfo.Builder` class.

Passing parameters to a job via the JobInfo

Parameters are passed to a job by creating a `PersistableBundle` that is passed along with the `Job.Builder.SetExtras` method:

```
var jobParameters = new PersistableBundle();
jobParameters.PutInt("LoopCount", 11);

var jobBuilder = this.CreateJobBuilderUsingJobId<DownloadJob>(1)
                     .SetExtras(jobParameters)
                     .Build();
```

The `PersistableBundle` is accessed from the `Android.App.Job.JobParameters.Extras` property in the `OnStartJob` method of a `JobService`:

```
public override bool OnStartJob(JobParameters jobParameters)
{
    var loopCount = jobParams.Extras.GetInt("LoopCount", 10);

    // rest of code omitted
}
```

Scheduling a job

To schedule a job, a Xamarin.Android application will get a reference to the `JobScheduler` system service and call the `JobScheduler.Schedule` method with the `JobInfo` object that was created in the previous step. `JobScheduler.Schedule` will immediately return with one of two integer values:

- **JobScheduler.ResultSuccess** – The job has been successfully scheduled.
- **JobScheduler.ResultFailure** – The job could not be scheduled. This is typically caused by conflicting `JobInfo` parameters.

This code is an example of scheduling a job and notifying the user of the results of the scheduling attempt:

```
var jobScheduler = (JobScheduler)GetSystemService(JobSchedulerService);
var scheduleResult = jobScheduler.Schedule(jobInfo);

if (JobScheduler.ResultSuccess == scheduleResult)
{
    var snackBar = Snackbar.Make(FindViewById(Android.Resource.Id.Content),
Resource.String.jobscheduled_success, Snackbar.LengthShort);
    snackBar.Show();
}
else
{
    var snackBar = Snackbar.Make(FindViewById(Android.Resource.Id.Content),
Resource.String.jobscheduled_failure, Snackbar.LengthShort);
    snackBar.Show();
}
```

Cancelling a job

It is possible to cancel all the jobs that have been scheduled, or just a single job using the `JobsScheduler.CancelAll()` method or the `JobScheduler.Cancel(jobId)` method:

```
// Cancel all jobs
jobScheduler.CancelAll();

// to cancel a job with jobID = 1
jobScheduler.Cancel(1)
```

Summary

This guide discussed how to use the Android Job Scheduler to intelligently perform work in the background. It discussed how to encapsulate the work to be performed as a `JobService` and how to use the `JobScheduler` to schedule that work, specifying the criteria with a `JobTrigger` and how failures should be handled with a `RetryStrategy`.

Related Links

- Intelligent Job-Scheduling
- JobScheduler API reference
- Scheduling jobs like a pro with JobScheduler
- Android Battery and Memory Optimizations - Google I/O 2016 (video)
- Android JobScheduler - René Ruppert

Firebase Job Dispatcher

11/2/2020 • 13 minutes to read • Edit Online

This guide discusses how to schedule background work using the Firebase Job Dispatcher library from Google.

Overview

One of the best ways to keep an Android application responsive to the user is to ensure that complex or long running work is performed in the background. However, it is important that background work will not negatively impact the user's experience with the device.

For example, a background job might poll a website every three or four minutes to query for changes to a particular dataset. This seems benign, however it would have a disastrous impact on battery life. The application will repeatedly wake up the device, elevate the CPU to a higher power state, power up the radios, make the network requests, and then processing the results. It gets worse because the device will not immediately power down and return to the low-power idle state. Poorly scheduled background work may inadvertently keep the device in a state with unnecessary and excessive power requirements. This seemingly innocent activity (polling a website) will render the device unusable in a relatively short period of time.

Android provides the following APIs to help with performing work in the background but by themselves they are not sufficient for intelligent job scheduling.

- **Intent Services** – Intent Services are great for performing the work, however they provide no way to schedule work.
- **AlarmManager** – These APIs only allow work to be scheduled but provide no way to actually perform the work. Also, the AlarmManager only allows time based constraints, which means raise an alarm at a certain time or after a certain period of time has elapsed.
- **JobScheduler** – The JobSchedule is a great API that works with the operating system to schedule jobs. However, it is only available for those Android apps that target API level 21 or higher.
- **Broadcast Receivers** – An Android app can setup broadcast receivers to perform work in response to system-wide events or Intents. However, broadcast receivers don't provide any control over when the job should be run. Also changes in the Android operating system will restrict when broadcast receivers will work, or the kinds of work that they can respond to.

There are two key features to efficiently performing background work (sometimes referred to as a *background job* or a *job*):

1. **Intelligently scheduling the work** – It is important that when an application is doing work in the background that it does so as a good citizen. Ideally, the application should not demand that a job be run. Instead, the application should specify conditions that must be met for when the job can run, and then schedule that work to run when the conditions are met. This allows Android to intelligently perform work. For example, network requests may be batched to run all at the same time to make maximum use of overhead involved with networking.
2. **Encapsulating the work** – The code to perform the background work should be encapsulated in a discrete component that can be run independently of the user interface and will be relatively easy to reschedule if the work fails to complete for some reason.

The Firebase Job Dispatcher is a library from Google that provides a fluent API to simplify scheduling background work. It is intended to be the replacement for Google Cloud Manager. The Firebase Job Dispatcher consists of the following APIs:

- A `Firebase.JobDispatcher.JobService` is an abstract class that must be extended with the logic that will run in the background job.
- A `Firebase.JobDispatcher.JobTrigger` declares when the job should be started. This is typically expressed as a window of time, for example, wait at least 30 seconds before starting the job, but run the job within 5 minutes.
- A `Firebase.JobDispatcher.RetryStrategy` contains information about what should be done when a job fails to execute properly. The retry strategy specifies how long to wait before trying to run the job again.
- A `Firebase.JobDispatcher.Constraint` is an optional value that describes a condition that must be met before the job can run, such as the device is on an unmetered network or charging.
- The `Firebase.JobDispatcher.Job` is an API that unifies the previous APIs in to a unit-of-work that can be scheduled by the `JobDispatcher`. The `Job.Builder` class is used to instantiate a `Job`.
- A `Firebase.JobDispatcher.JobDispatcher` uses the previous three APIs to schedule the work with the operating system and to provide a way to cancel jobs, if necessary.

To schedule work with the Firebase Job Dispatcher, a Xamarin.Android application must encapsulate the code in a type that extends the `JobService` class. `JobService` has three lifecycle methods that can be called during the lifetime of the job:

- `bool OnStartJob(IJobParameters parameters)` – This method is where the work will occur and should always be implemented. It runs on the main thread. This method will return `true` if there is work remaining, or `false` if the work is done.
- `bool OnStopJob(IJobParameters parameters)` – This is called when the job is stopped for some reason. It should return `true` if the job should be rescheduled for later.
- `JobFinished(IJobParameters parameters, bool needsReschedule)` – This method is called when the `JobService` has finished any asynchronous work.

To schedule a job, the application will instantiate a `JobDispatcher` object. Then, a `Job.Builder` is used to create a `Job` object, which is provided to the `JobDispatcher` which will try and schedule the job to run.

This guide will discuss how to add the Firebase Job Dispatcher to a Xamarin.Android application and use it to schedule background work.

Requirements

The Firebase Job Dispatcher requires Android API level 9 or higher. The Firebase Job Dispatcher library relies on some components provided by Google Play Services; the device must have Google Play Services installed.

Using the Firebase Job Dispatcher Library in Xamarin.Android

To get started with the Firebase Job Dispatcher, first add the Xamarin.Firebase.JobDispatcher NuGet package to the Xamarin.Android project. Search the NuGet Package Manager for the **Xamarin.Firebase.JobDispatcher** package (which is still in pre-release).

After adding the Firebase Job Dispatcher library, create a `JobService` class and then schedule it to run with an instance of the `FirebaseJobDispatcher`.

Creating a JobService

All work performed by the Firebase Job Dispatcher library must be done in a type that extends the `Firebase.JobDispatcher.JobService` abstract class. Creating a `JobService` is very similar to creating a `Service` with the Android framework:

1. Extend the `JobService` class
2. Decorate the subclass with the `ServiceAttribute`. Although not strictly required, it is recommended to

explicitly set the `Name` parameter to help with debugging the `JobService`.

3. Add an `IntentFilter` to declare the `JobService` in the **AndroidManifest.xml**. This will also help the Firebase Job Dispatcher library locate and invoke the `JobService`.

The following code is an example of the simplest `JobService` for an application, using the TPL to asynchronously perform some work:

```
[Service(Name = "com.xamarin.fjdtestapp.DemoJob")]
[IntentFilter(new[] {FirebaseJobServiceIntent.Action})]
public class DemoJob : JobService
{
    static readonly string TAG = "X:DemoService";

    public override bool OnStartJob(IJobParameters jobParameters)
    {
        Task.Run(() =>
        {
            // Work is happening asynchronously (code omitted)

        });

        // Return true because of the asynchronous work
        return true;
    }

    public override bool OnStopJob(IJobParameters jobParameters)
    {
        Log.Debug(TAG, "DemoJob::OnStartJob");
        // nothing to do.
        return false;
    }
}
```

Creating a FirebaseJobDispatcher

Before any work can be scheduled, it is necessary to create a `Firebase.JobDispatcher.FirebaseJobDispatcher` object. The `FirebaseJobDispatcher` is responsible for scheduling a `JobService`. The following code snippet is one way to create an instance of the `FirebaseJobDispatcher`:

```
// This is the "Java" way to create a FirebaseJobDispatcher object
IDriver driver = new GooglePlayDriver(context);
FirebaseJobDispatcher dispatcher = new FirebaseJobDispatcher(driver);
```

In the previous code snippet, the `GooglePlayDriver` is class that helps the `FirebaseJobDispatcher` interact with some of the scheduling APIs in Google Play Services on the device. The parameter `context` is any Android `Context`, such as an Activity. Currently the `GooglePlayDriver` is the only `IDriver` implementation in the Firebase Job Dispatcher library.

The Xamarin.Android binding for the Firebase Job Dispatcher provides an extension method to create a `FirebaseJobDispatcher` from the `Context`:

```
FirebaseJobDispatcher dispatcher = context.CreateJobDispatcher();
```

Once the `FirebaseJobDispatcher` has been instantiated, it is possible to create a `Job` and run the code in the `JobService` class. The `Job` is created by a `Job.Builder` object and will be discussed in the next section.

Creating a Firebase.JobDispatcher.Job with the Job.Builder

The `Firebase.JobDispatcher.Job` class is responsible for encapsulating the meta-data necessary to run a

`JobService` . A `Job` contains information such as any constraint that must be met before the job can run, if the `Job` is recurring, or any triggers that will cause the job to be run. As a bare minimum, a `Job` must have a *tag* (a unique string that identifies the job to the `FirebaseJobDispatcher`) and the type of the `JobService` that should be run. The Firebase Job Dispatcher will instantiate the `JobService` when it is time to run the job. A `Job` is created by using an instance of the `Firebase.JobDispatcher.Job.JobBuilder` class.

The following code snippet is the simplest example of how to create a `Job` using the Xamarin.Android binding:

```
Job myJob = dispatcher.NewJobBuilder()
                      .SetService<DemoJob>("demo-job-tag")
                      .Build();
```

The `Job.Builder` will perform some basic validation checks on the input values for the job. An exception will be thrown if it not possible for the `Job.Builder` to create a `Job` . The `Job.Builder` will create a `Job` with the following defaults:

- A `Job` 's *lifetime* (how long it will be scheduled to run) is only until the device reboots – once the device reboots the `Job` is lost.
- A `Job` is not recurring – it will only run once.
- A `Job` will be scheduled to run as soon as possible.
- The default retry strategy for a `Job` is to use an *exponential backoff* (discussed on more detail below in the section Setting a RetryStrategy)

Scheduling a job

After creating the `Job` , it needs to be scheduled with the `FirebaseJobDispatcher` before it is run. There are two methods for scheduling a `Job` :

```
// This will throw an exception if there was a problem scheduling the job
dispatcher.MustSchedule(myJob);

// This method will not throw an exception; an integer result value is returned
int scheduleResult = dispatcher.Schedule(myJob);
```

The value returned by `FirebaseJobDispatcher.Schedule` will be one of the following integer values:

- `FirebaseJobDispatcher.ScheduleResultSuccess` – The `Job` was successfully scheduled.
- `FirebaseJobDispatcher.ScheduleResultUnknownError` – Some unknown problem occurred which prevented the `Job` from being scheduled.
- `FirebaseJobDispatcher.ScheduleResultNoDriverAvailable` – An invalid `IDriver` was used or the `IDriver` was somehow unavailable.
- `FirebaseJobDispatcher.ScheduleResultUnsupportedTrigger` – The `Trigger` was not supported.
- `FirebaseJobDispatcher.ScheduleResultBadService` – The service is not configured correctly or is unavailable.

Configuring a job

It is possible to customize a job. Examples of how a job may be customized include the following:

- Passing Parameters to a Job – A `Job` may require additional values to perform its work, for example downloading a file.
- Set Constraints – It may be necessary to only run a job when certain conditions are met. For example, only run a `Job` when the device is charging.
- Specify when a `Job` should run – The Firebase Job Dispatcher allows applications to specify a time when the job should run.
- Declare a retry strategy for failed jobs – A *retry strategy* provides guidance to the `FirebaseJobDispatcher` on

what to do with `Jobs` that fail to complete.

Each of these topics will be discussed more in the following sections.

Passing parameters to a job

Parameters are passed to a job by creating a `Bundle` that is passed along with the `Job.Builder.SetExtras` method:

```
Bundle jobParameters = new Bundle();
jobParameters.PutInt(FibonacciCalculatorJob.FibonacciPositionKey, 25);

Job myJob = dispatcher.NewJobBuilder()
                      .SetService<DemoJob>("demo-job-tag")
                      .SetExtras(jobParameters)
                      .Build();
```

The `Bundle` is accessed from the `IJobParameters.Extras` property on the `OnStartJob` method:

```
public override bool OnStartJob(IJobParameters jobParameters)
{
    int position = jobParameters.Extras.GetInt(FibonacciPositionKey, DEFAULT_VALUE);

    // rest of code omitted
}
```

Setting constraints

Constraints can help reduces costs or battery drain on the device. The `Firebase.JobDispatcher.Constraint` class defines these constraints as integer values:

- `Constraint.OnUnmeteredNetwork` – Only run the job when the device is connected to an unmetered network. This is useful to prevent the user from incurring data charges.
- `Constraint.OnAnyNetwork` – Run the job on whatever network the device is connected to. If specified along with `Constraint.OnUnmeteredNetwork`, this value will take priority.
- `Constraint.DeviceCharging` – Run the job only when the device is charging.

Constraints are set with the `Job.Builder.SetConstraint` method:

```
Job myJob = dispatcher.NewJobBuilder()
                      .SetService<DemoJob>("demo-job-tag")
                      .SetConstraint(Constraint.DeviceCharging)
                      .Build();
```

The `JobTrigger` provides guidance to the operating system about when the job should start. A `JobTrigger` has an *executing window* that defines a scheduled time for when the `Job` should run. The execution window has a *start window* value and an *end window* value. The start window is the number of seconds that the device should wait before running the job and the end window value is the maximum number of seconds to wait before running the `Job`.

A `JobTrigger` can be created with the `Firebase.Jobdispatcher.Trigger.ExecutionWindow` method. For example `Trigger.ExecutionWindow(15,60)` means that the job should run between 15 and 60 seconds from when it is scheduled. The `Job.Builder.SetTrigger` method is used to

```
JobTrigger myTrigger = Trigger.ExecutionWindow(15,60);
Job myJob = dispatcher.NewJobBuilder()
                      .SetService<DemoJob>("demo-job-tag")
                      .SetTrigger(myTrigger)
                      .Build();
```

The default `JobTrigger` for a job is represented by the value `Trigger.Now`, which specifies that a job be run as soon as possible after being scheduled..

Setting a RetryStrategy

The `Firebase.JobDispatcher.RetryStrategy` is used to specify how much of a delay a device should use before trying to re-run a failed job. A `RetryStrategy` has a *policy*, which defines what time-base algorithm will be used to re-schedule the failed job, and an execution window that specifies a window in which the job should be scheduled. This *rescheduling window* is defined by two values. The first value is the number of seconds to wait before rescheduling the job (the *initial backoff* value), and the second number is the maximum number of seconds before the job must run (the *maximum backoff* value).

The two types of retry policies are identified by these int values:

- `RetryStrategy.RetryPolicyExponential` – An *exponential backoff* policy will increase the initial backoff value exponentially after each failure. The first time a job fails, the library will wait the _initial interval that is specified before rescheduling the job – example 30 seconds. The second time the job fails, the library will wait at least 60 seconds before trying to run the job. After the third failed attempt, the library will wait 120 seconds, and so on. The default `RetryStrategy` for the Firebase Job Dispatcher library is represented by the `RetryStrategy.DefaultExponential` object. It has an initial backoff of 30 seconds and a maximum backoff of 3600 seconds.
- `RetryStrategy.RetryPolicyLinear` – This strategy is a *linear backoff* that the job should be rescheduled to run at set intervals (until it succeeds). Linear backoff is best suited for work that must be completed as soon as possible or for problems that will quickly resolve themselves. The Firebase Job Dispatcher library defines a `RetryStrategy.DefaultLinear` which has a rescheduling window of at least 30 seconds and up to 3600 seconds.

It is possible to define a custom `RetryStrategy` with the `FirebaseJobDispatcher.NewRetryStrategy` method. It takes three parameters:

1. `int policy` – The *policy* is one of the previous `RetryStrategy` values, `RetryStrategy.RetryPolicyLinear`, or `RetryStrategy.RetryPolicyExponential`.
2. `int initialBackoffSeconds` – The *initial backoff* is a delay, in seconds, that is required before trying to run the job again. The default value for this is 30 seconds.
3. `int maximumBackoffSeconds` – The *maximum backoff* value declares the maximum number of seconds to delay before trying to run the job again. The default value is 3600 seconds.

```
RetryStrategy retry = dispatcher.NewRetryStrategy(RetryStrategy.RetryPolicyLinear, initialBackoffSeconds,
maximumBackoffSet);

// Create a Job and set the RetryStrategy via the Job.Builder
Job myJob = dispatcher.NewJobBuilder()
                      .SetService<DemoJob>("demo-job-tag")
                      .SetRetryStrategy(retry)
                      .Build();
```

Cancelling a job

It is possible to cancel all the jobs that have been scheduled, or just a single job using the `FirebaseJobDispatcher.CancelAll()` method or the `FirebaseJobDispatcher.Cancel(string)` method:

```
int cancelResult = dispatcher.CancelAll();

// to cancel a single job:

int cancelResult = dispatcher.Cancel("unique-tag-for-job");
```

Either method will return an integer value:

- `FirebaseJobDispatcher.CancelResultSuccess` – The job was successfully cancelled.
- `FirebaseJobDispatcher.CancelResultUnknownError` – An error prevented the job from being cancelled.
- `FirebaseJobDispatcher.CancelResult.NoDriverAvailable` – The `FirebaseJobDispatcher` is unable to cancel the job as there is no valid `IDriver` available.

Summary

This guide discussed how to use the Firebase Job Dispatcher to intelligently perform work in the background. It discussed how to encapsulate the work to be performed as a `JobService` and how to use the `FirebaseJobDispatcher` to schedule that work, specifying the criteria with a `JobTrigger` and how failures should be handled with a `RetryStrategy`.

Related Links

- Xamarin.Firebase.JobDispatcher on NuGet
- firebase-job-dispatcher on GitHub
- Xamarin.Firebase.JobDispatcher Binding
- Intelligent Job-Scheduling
- Android Battery and Memory Optimizations - Google I/O 2016 (video)

Fragments

7/8/2021 • 3 minutes to read • Edit Online

Android 3.0 introduced Fragments, showing how to support more flexible designs for the many different screen sizes found on phones and tablets. This article will cover how to use Fragments to develop Xamarin.Android applications, and also how to support Fragments on pre-Android 3.0 (API Level 11) devices.

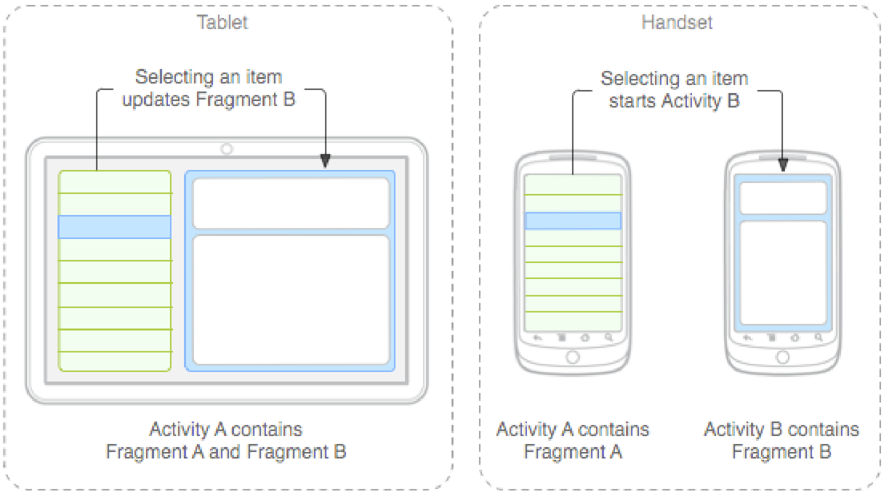
Fragments Overview

The larger screen sizes found on most tablets added an extra layer of complexity to Android development—a layout designed for the small screen does not necessarily work as well for larger screens, and vice-versa. To reduce the number of complications that this introduced, Android 3.0 added two new features, *Fragments* and *Support Packages*.

Fragments can be thought of as user interface modules. They let the developer divide up the user interface into isolated, reusable parts that can be run in separate Activities. At run time, the Activities themselves will decide which Fragments to use.

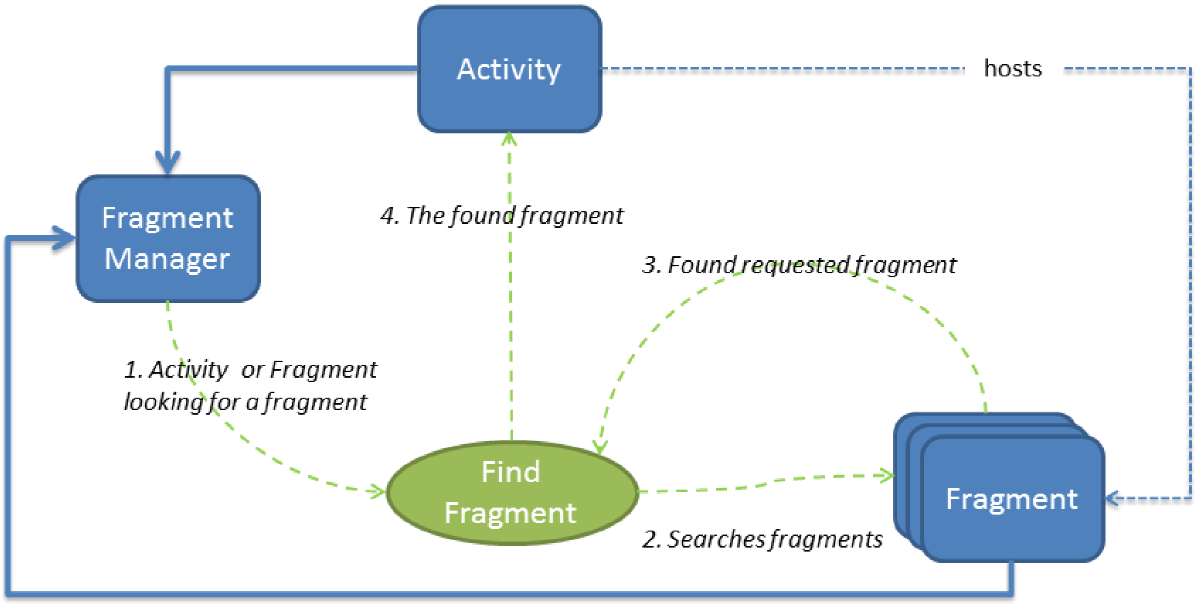
Support Packages were originally called *Compatibility Libraries* and allowed Fragments to be used on devices that run versions of Android prior to Android 3.0 (API Level 11).

For example, the image below illustrates how a single application uses Fragments across varying device form factors.



Fragment A contains a list, while *Fragment B* contains details for an item selected in that list. When the application is run on a tablet, it can display both Fragments on the same Activity. When the same application is run on a handset (with its smaller screen size), the Fragments are hosted in two separate Activities. Fragment A and Fragment B are the same on both form factors, but the Activities that host them are different.

To help an Activity coordinate and manage all these Fragments, Android introduced a new class called the *FragmentManager*. Each Activity has its own instance of a `FragmentManager` for adding, deleting, and finding hosted Fragments. The following diagram illustrates the relationship between Fragments and Activities:



In some regards, Fragments can be thought of as composite controls or as mini-Activities. They bundle up pieces of UI into reusable modules that can then be used independently by developers in Activities. A Fragment does have a view hierarchy—just like an Activity—but, unlike an Activity, it can be shared across screens. Views differ from Fragments in that Fragments have their own lifecycle; views do not.

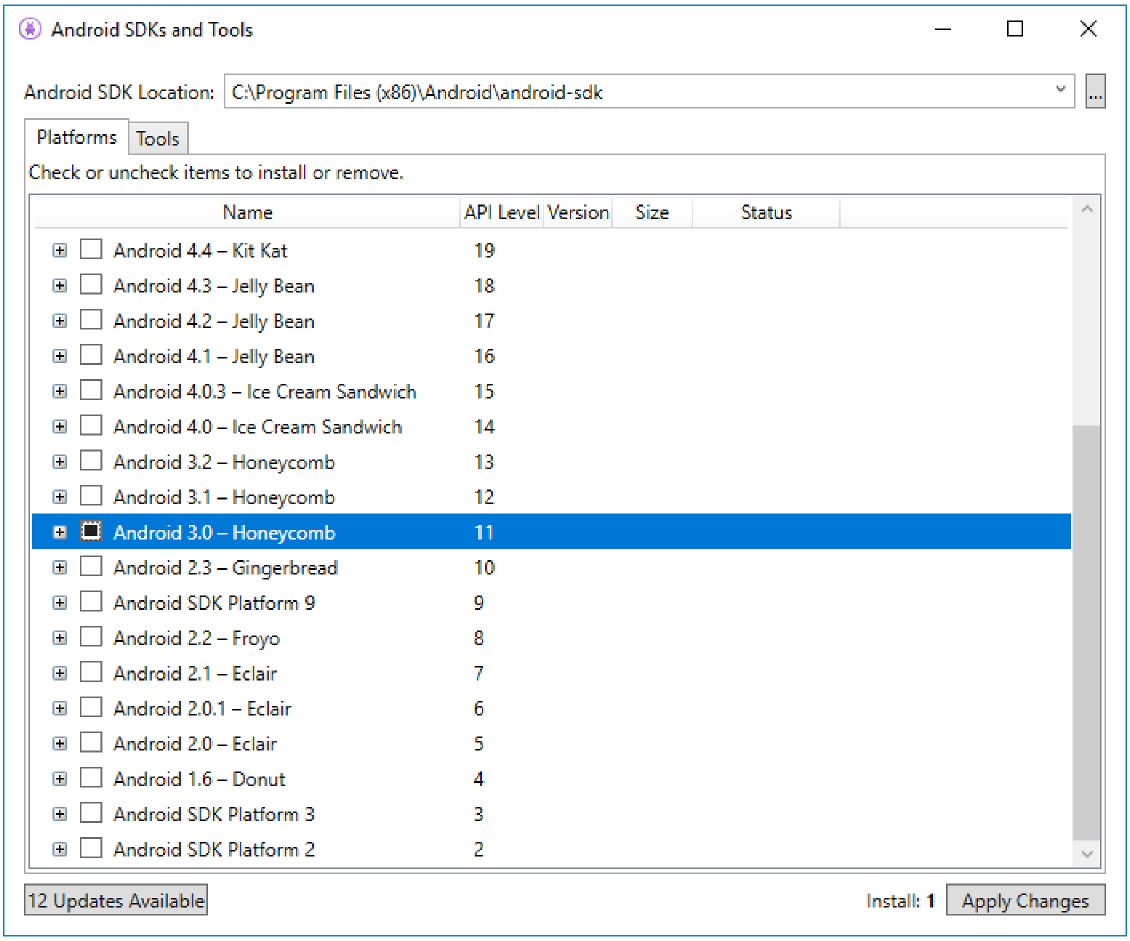
While the Activity is a host to one or more Fragments, it is not directly aware of the Fragments themselves. Likewise, Fragments are not directly aware of other Fragments in the hosting Activity. However, Fragments and Activities are aware of the `FragmentManager` in their Activity. By using the `FragmentManager`, it is possible for an Activity or a Fragment to obtain a reference to a specific instance of a Fragment, and then call methods on that instance. In this way, the Activity or Fragments can communicate and interact with other Fragments.

This guide contains comprehensive coverage about how to use Fragments, including:

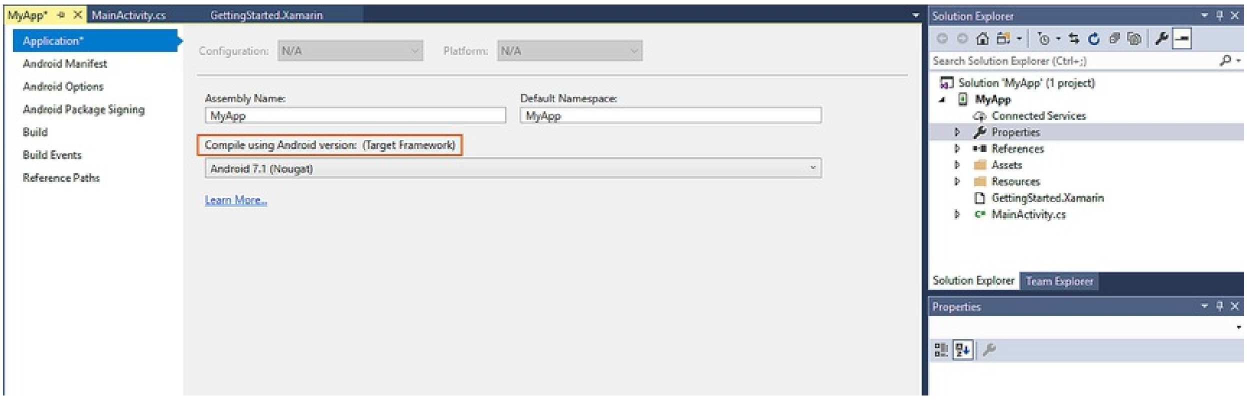
- **Creating Fragments** – How to create a basic Fragment and key methods that must be implemented.
- **Fragment Management and Transactions** – How to manipulate Fragments at run time.
- **Android Support Package** – How to use the libraries that allow Fragments to be used on older versions of Android.

Requirements

Fragments are available in the Android SDK starting with API level 11 (Android 3.0), as shown in the following screenshot:



Fragments are available in Xamarin.Android 4.0 and higher. A Xamarin.Android application must target at least API level 11 (Android 3.0) or higher in order to use Fragments. The Target Framework may be set in the project Properties as shown below:



It is possible to use Fragments in older versions of Android by using the Android Support Package and Xamarin.Android 4.2 or higher. How to do this is covered in more detail in the documents of this section.

Related Links

- Honeycomb Gallery (sample)
- Fragments
- Support Package

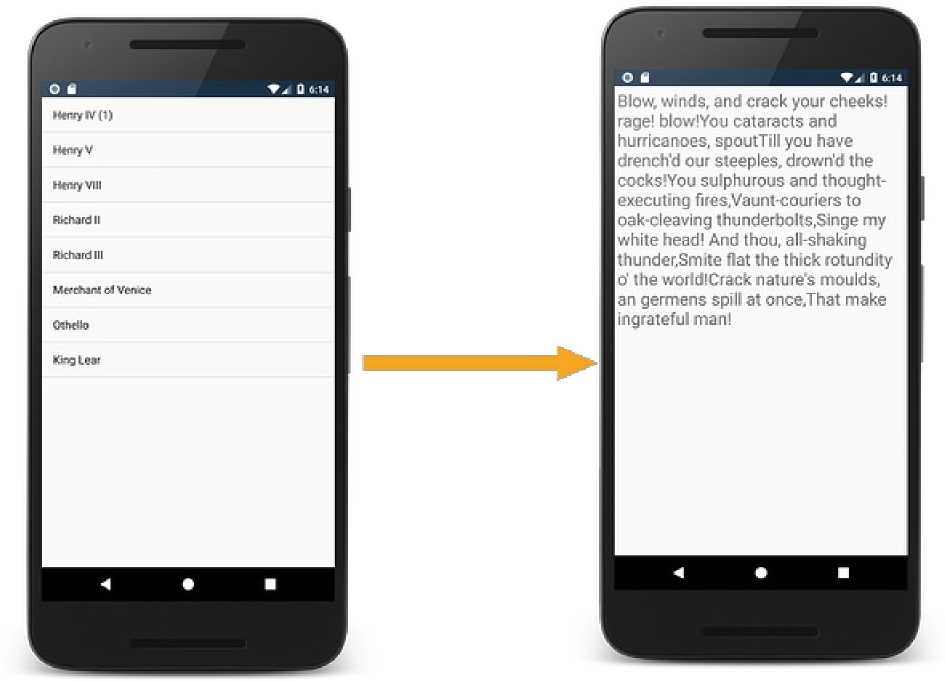
Implementing fragments - walkthrough

7/8/2021 • 2 minutes to read • Edit Online

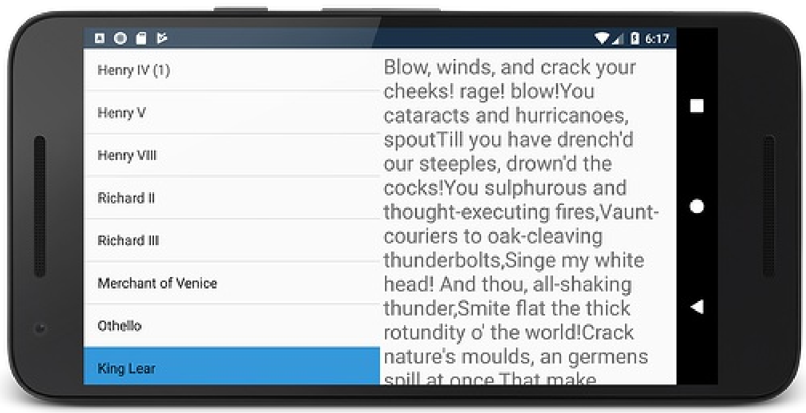
Fragments are self-contained, modular components that can help address the complexity of Android apps that target devices with a variety of screen sizes. This article walks through how to create and use fragments when developing Xamarin.Android applications.

Overview

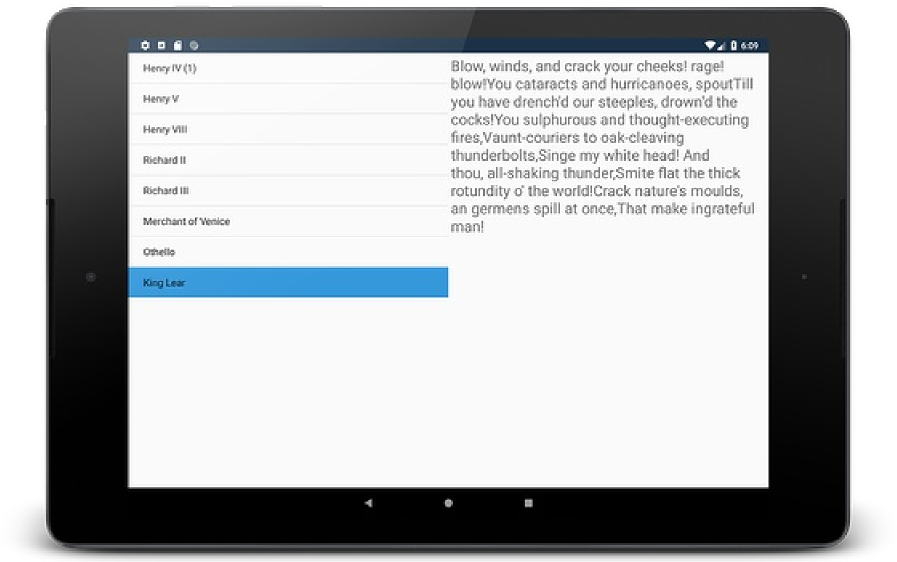
In this section, you'll walk through how to create and use fragments in a Xamarin.Android application. This application will display the titles of several plays by William Shakespeare in a list. When the user taps on the title of a play, then the app will display a quote from that play in a separate activity:



When the phone is rotated to landscape mode, the appearance of the app will change: both the list of plays and quotes will appear in the same activity. When a play is selected, the quote will be display in the same activity:



Finally, if the app is running on a tablet:



This sample application can easily adapt to the different form factors and orientations with minimal code changes by using fragments and Alternate Layouts.

The data for the application will exist in two string arrays that are hardcoded in the app as C# string arrays. Each of the arrays will serve as the data source for one fragment. One array will hold the name of some plays by Shakespeare, and the other array will hold a quote from that play. When the app starts up, it will display the play names in a `ListFragment`. When the user clicks on a play in the `ListFragment`, the app will start up another activity which will display the quote.

The user interface for the app will consist of two layouts, one for portrait and one for landscape mode. At run time, Android will determine what layout to load based on the orientation of the device and will provide that layout to the Activity to render. All of the logic for responding to user clicks and displaying the data will be contained in fragments. The Activities in the app exist only as containers that will host the fragments.

This walkthrough will be broken down into two guides. The first part will focus on the core parts of the application. A single set of layouts (optimized for portrait mode) will be created, along with two fragments and two Activities:

1. `MainActivity` This is the startup Activity for the app.
2. `TitlesFragment` This fragment will display a list of titles of plays that were written by William Shakespeare. It will be hosted by `MainActivity`.
3. `PlayQuoteActivity` `TitlesFragment` will start the `PlayQuoteActivity` in response to the user selecting a play in `TitlesFragment`.
4. `PlayQuoteFragment` This fragment will display a quote from a play by William Shakespeare. It will be hosted by `PlayQuoteActivity`.

The second part of this walkthrough will discuss adding an alternate layout (optimized for landscape mode) which will display both fragments on the screen. Also, some minor code changes will be made to the code so that the app will adapt its behavior to the number of fragments that are concurrently displayed on the screen.

Related Links

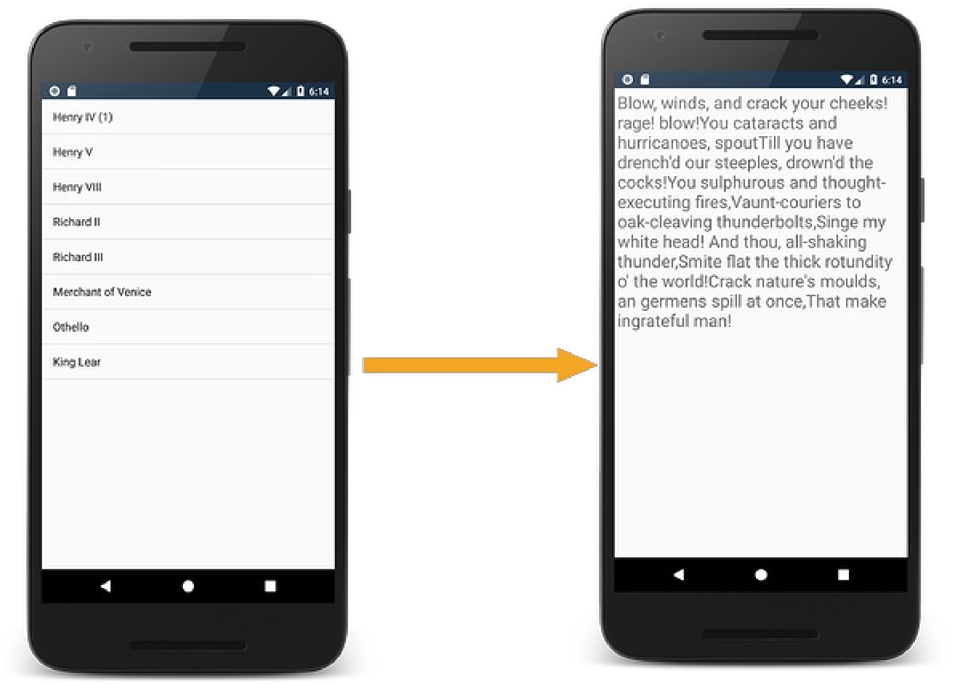
- FragmentsWalkthrough (sample)
- Designer Overview

- Implementing Fragments
- Support Package

Fragments walkthrough – phone

7/8/2021 • 13 minutes to read • Edit Online

This is the first part of a walkthrough that will create a Xamarin.Android app that targets an Android device in portrait orientation. This walkthrough will discuss how to create fragments in Xamarin.Android and how to add them to a sample.



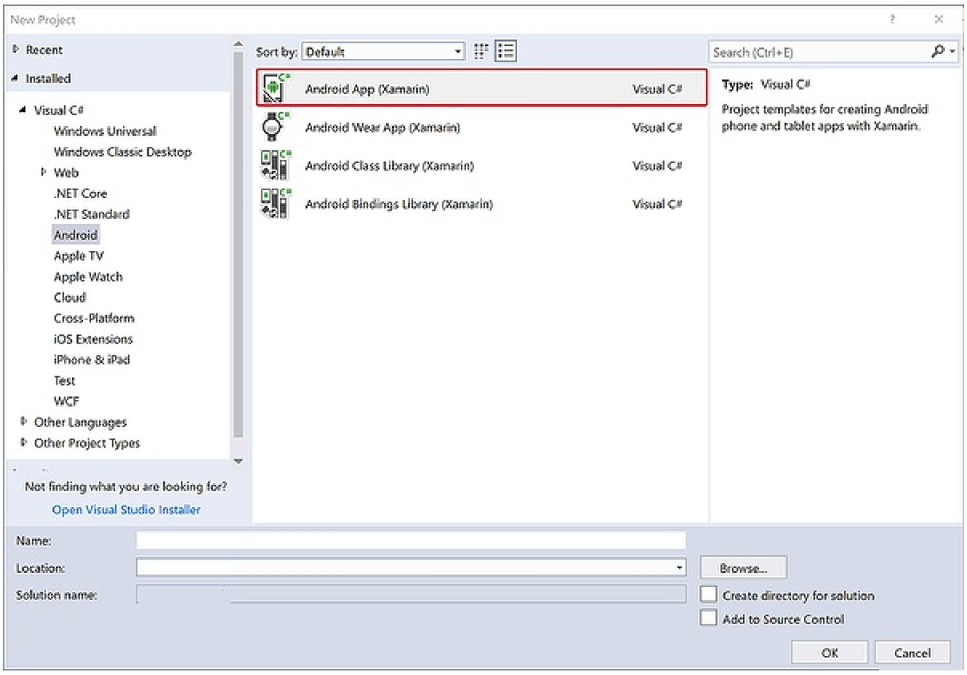
The following classes will be created for this app:

1. `PlayQuoteFragment` This fragment will display a quote from a play by William Shakespeare. It will be hosted by `PlayQuoteActivity`.
2. `Shakespeare` This class will hold two hardcoded arrays as properties.
3. `TitlesFragment` This fragment will display a list of titles of plays that were written by William Shakespeare. It will be hosted by `MainActivity`.
4. `PlayQuoteActivity` `TitlesFragment` will start the `PlayQuoteActivity` in response to the user selecting a play in `TitlesFragment`.

1. Create the Android project

Create a new Xamarin.Android project called **FragmentSample**.

- Visual Studio
- Visual Studio for Mac



2. Add the data

The data for this application will be stored in two hardcoded string arrays that are properties of a class name `Shakespeare`:

- `Shakespeare.Titles` This array will hold a list of plays from William Shakespeare. This is the data source for the `TitlesFragment`.
- `Shakespeare.Dialogue` This array will hold a list of quotes from one of the plays contained in `Shakespeare.Titles`. This is the data source for the `PlayQuoteFragment`.

Add a new C# class to the **FragmentSample** project and name it **Shakespeare.cs**. Inside this file, create a new C# class called `Shakespeare` with the following contents

```
class Shakespeare
{
    public static string[] Titles = {
                                      "Henry IV (1)",
                                      "Henry V",
                                      "Henry VIII",
                                      "Richard II",
                                      "Richard III",
                                      "Merchant of Venice",
                                      "Othello",
                                      "King Lear"
                                    };

    public static string[] Dialogue = {
                                        "So shaken as we are, so wan with care, Find we a time for frighted
peace to pant, And breathe short-winded accents of new broils To be commenced in strands afar remote. No
more the thirsty entrance of this soil Shall daub her lips with her own children's blood; Nor more shall
trenching war channel her fields, Nor bruise her flowerets with the armed hoofs Of hostile paces: those
opposed eyes, Which, like the meteors of a troubled heaven, All of one nature, of one substance bred, Did
lately meet in the intestine shock And furious close of civil butchery Shall now, in mutual well-beseeming
ranks, March all one way and be no more opposed Against acquaintance, kindred and allies: The edge of war,
like an ill-sheathed knife, No more shall cut his master. Therefore, friends, As far as to the sepulchre of
Christ, Whose soldier now, under whose blessed cross We are impressed and engaged to fight, Forthwith a
power of English shall we levy; Whose arms were moulded in their mothers' womb To chase these pagans in
those holy fields Over whose acres walk'd those blessed feet Which fourteen hundred years ago were nail'd
For our advantage on the bitter cross. But this our purpose now is twelve month old, And bootless 'tis to
tell you we will go: Therefore we meet not now. Then let me hear Of you, my gentle cousin Westmoreland, What
yesternight our council did decree In forwarding this dear expedience.",
                                        "Hear him but reason in divinity, And all-admiring with an inward
```

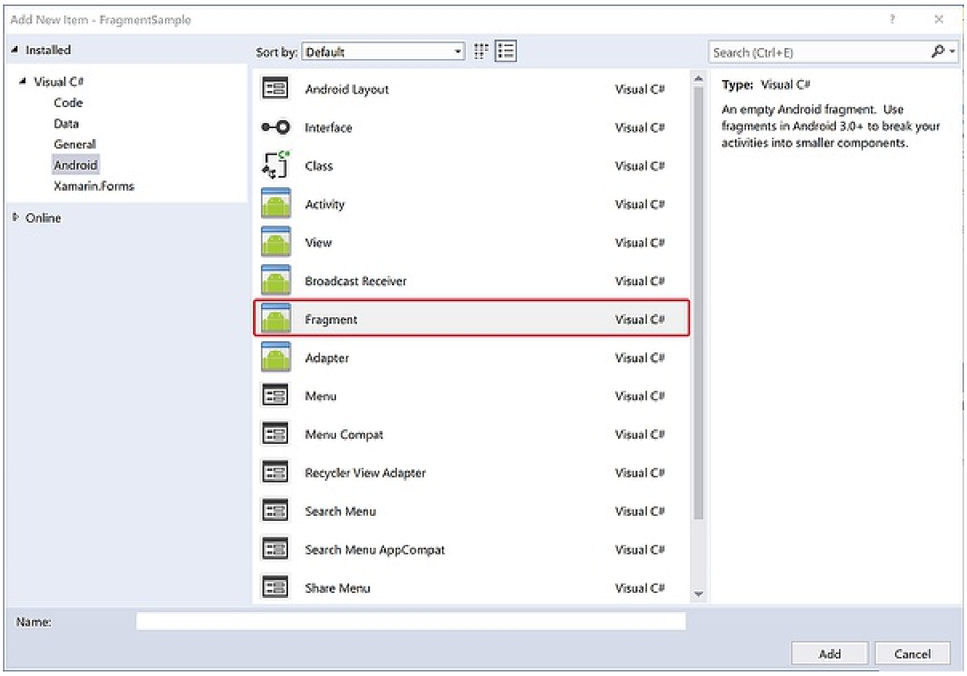
```
                                        Hear him but reason in divinity, And all-admiring with an inward
wish You would desire the king were made a prelate: Hear him debate of commonwealth affairs, You would say
it hath been all in all his study: List his discourse of war, and you shall hear A fearful battle render'd
you in music: Turn him to any cause of policy, The Gordian knot of it he will unloose, Familiar as his
garter: that, when he speaks, The air, a charter'd libertine, is still, And the mute wonder lurketh in men's
ears, To steal his sweet and honey'd sentences; So that the art and practic part of life Must be the
mistress to this theoric: Which is a wonder how his grace should glean it, Since his addiction was to
courses vain, His companies unletter'd, rude and shallow, His hours fill'd up with riots, banquets, sports,
And never noted in him any study, Any retirement, any sequestration From open haunts and popularity.",
                                        "I come no more to make you laugh: things now, That bear a weighty
and a serious brow, Sad, high, and working, full of state and woe, Such noble scenes as draw the eye to
flow, We now present. Those that can pity, here May, if they think it well, let fall a tear; The subject
will deserve it. Such as give Their money out of hope they may believe, May here find truth too. Those that
come to see Only a show or two, and so agree The play may pass, if they be still and willing, I'll undertake
may see away their shilling Richly in two short hours. Only they That come to hear a merry bawdy play, A
noise of targets, or to see a fellow In a long motley coat guarded with yellow, Will be deceived; for,
gentle hearers, know, To rank our chosen truth with such a show As fool and fight is, beside forfeiting Our
own brains, and the opinion that we bring, To make that only true we now intend, Will leave us never an
understanding friend. Therefore, for goodness' sake, and as you are known The first and happiest hearers of
the town, Be sad, as we would make ye: think ye see The very persons of our noble story As they were living;
think you see them great, And follow'd with the general throng and sweat Of thousand friends; then in a
moment, see How soon this mightiness meets misery: And, if you can be merry then, I'll say A man may weep
upon his wedding-day.",
                                        "First, heaven be the record to my speech! In the devotion of a
subject's love, Tendering the precious safety of my prince, And free from other misbegotten hate, Come I
appellant to this princely presence. Now, Thomas Mowbray, do I turn to thee, And mark my greeting well; for
what I speak My body shall make good upon this earth, Or my divine soul answer it in heaven. Thou art a
traitor and a miscreant, Too good to be so and too bad to live, Since the more fair and crystal is the sky,
The uglier seem the clouds that in it fly. Once more, the more to aggravate the note, With a foul traitor's
name stuff I thy throat; And wish, so please my sovereign, ere I move, What my tongue speaks my right drawn
sword may prove.",
                                        "Now is the winter of our discontent Made glorious summer by this
sun of York; And all the clouds that lour'd upon our house In the deep bosom of the ocean buried. Now are
our brows bound with victorious wreaths; Our bruised arms hung up for monuments; Our stern alarums changed
to merry meetings, Our dreadful marches to delightful measures. Grim-visaged war hath smooth'd his wrinkled
front; And now, instead of mounting barded steeds To fright the souls of fearful adversaries, He capers
nimbly in a lady's chamber To the lascivious pleasing of a lute. But I, that am not shaped for sportive
tricks, Nor made to court an amorous looking-glass; I, that am rudely stamp'd, and want love's majesty To
strut before a wanton ambling nymph; I, that am curtail'd of this fair proportion, Cheated of feature by
dissembling nature, Deformed, unfinish'd, sent before my time Into this breathing world, scarce half made
up, And that so lamely and unfashionable That dogs bark at me as I halt by them; Why, I, in this weak piping
time of peace, Have no delight to pass away the time, Unless to spy my shadow in the sun And descant on mine
own deformity: And therefore, since I cannot prove a lover, To entertain these fair well-spoken days, I am
determined to prove a villain And hate the idle pleasures of these days. Plots have I laid, inductions
dangerous, By drunken prophecies, libels and dreams, To set my brother Clarence and the king In deadly hate
the one against the other: And if King Edward be as true and just As I am subtle, false and treacherous,
This day should Clarence closely be mew'd up, About a prophecy, which says that 'G' Of Edward's heirs the
murderer shall be. Dive, thoughts, down to my soul: here Clarence comes.",
                                        "To bait fish withal: if it will feed nothing else, it will feed my
revenge. He hath disgraced me, and hindered me half a million; laughed at my losses, mocked at my gains,
scorned my nation, thwarted my bargains, cooled my friends, heated mine enemies; and what's his reason? I am
a Jew. Hath not a Jew eyes? hath not a Jew hands, organs, dimensions, senses, affections, passions? fed with
the same food, hurt with the same weapons, subject to the same diseases, healed by the same means, warmed
and cooled by the same winter and summer, as a Christian is? If you prick us, do we not bleed? if you tickle
us, do we not laugh? if you poison us, do we not die? and if you wrong us, shall we not revenge? If we are
like you in the rest, we will resemble you in that. If a Jew wrong a Christian, what is his humility?
Revenge. If a Christian wrong a Jew, what should his sufferance be by Christian example? Why, revenge. The
villany you teach me, I will execute, and it shall go hard but I will better the instruction.",
                                        "Virtue! a fig! 'tis in ourselves that we are thus or thus. Our
bodies are our gardens, to the which our wills are gardeners: so that if we will plant nettles, or sow
lettuce, set hyssop and weed up thyme, supply it with one gender of herbs, or distract it with many, either
to have it sterile with idleness, or manured with industry, why, the power and corrigible authority of this
lies in our wills. If the balance of our lives had not one scale of reason to poise another of sensuality,
the blood and baseness of our natures would conduct us to most preposterous conclusions: but we have reason
to cool our raging motions, our carnal stings, our unbitted lusts, whereof I take this that you call love to
be a sect or scion.",
                                        "Blow, winds, and crack your cheeks! rage! blow! You cataracts and
hurricanoes, spout Till you have drench'd our steeples, drown'd the cocks! You sulphurous and thought-
executing fires, Vaunt-couriers to oak-cleaving thunderbolts, Singe my white head! And thou, all-shaking
thunder, Smite flat the thick rotundity o' the world! Crack nature's moulds, an germens spill at once, That
```

```
thunder, Smite flat the thick rotundity o' the world! Crack nature's moulds, an germens spill at once, That
make ingrateful man!"
                                    };
}
```

3. Create the PlayQuoteFragment

The `PlayQuoteFragment` is an Android fragment that will display a quote for a Shakespeare play that was selected by the user earlier on in the application, This fragment will not use an Android layout file; instead, it will dynamically create its user interface. Add a new `Fragment` class named `PlayQuoteFragment` to the project:

- Visual Studio
- Visual Studio for Mac



Then, change the code for the fragment to resemble this snippet:

```
public class PlayQuoteFragment : Fragment
{
    public int PlayId => Arguments.GetInt("current_play_id", 0);

    public static PlayQuoteFragment NewInstance(int playId)
    {
        var bundle = new Bundle();
        bundle.PutInt("current_play_id", playId);
        return new PlayQuoteFragment {Arguments = bundle};
    }

    public override View OnCreateView(LayoutInflater inflater, ViewGroup container, Bundle
savedInstanceState)
    {
        if (container == null)
        {
            return null;
        }

        var textView = new TextView(Activity);
        var padding = Convert.ToInt32(TypedValue.ApplyDimension(ComplexUnitType.Dip, 4,
Activity.Resources.DisplayMetrics));
        textView.SetPadding(padding, padding, padding, padding);
        textView.TextSize = 24;
        textView.Text = Shakespeare.Dialogue[PlayId];

        var scroller = new ScrollView(Activity);
        scroller.AddView(textView);

        return scroller;
    }
}
```

It is a common pattern in Android apps to provide a factory method that will instantiate a fragment. This ensures that the fragment will be created with the necessary parameters for proper functioning. In this walkthrough, the app is expected to use the `PlayQuoteFragment.NewInstance` method to create a new fragment each time a quote is selected. The `NewInstance` method will take a single parameter – the index of the quote to display.

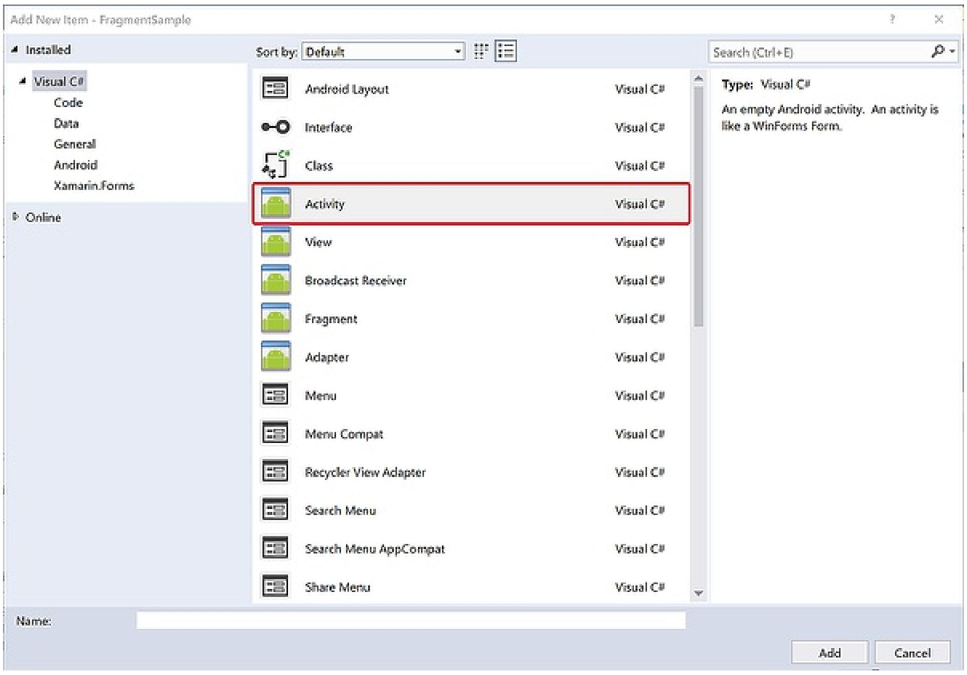
The `OnCreateView` method will be invoked by Android when it is time to render the fragment on the screen. It will return an Android `View` object that is the fragment. This fragment does not use a layout file to create a view. Instead, it will programmatically create the view by instantiating a **TextView** to hold the quote, and will display that widget in a **ScrollView**.

> **NOTE**
>
> Fragment sub-classes must have a public default constructor that has no parameters.

4. Create the PlayQuoteActivity

Fragments must be hosted inside an Activity, so this app requires an Activity that will host the `PlayQuoteFragment`. The Activity will dynamically add the fragment to its layout at run-time. Add a new Activity to the application and name it `PlayQuoteActivity`:

- Visual Studio
- Visual Studio for Mac



Edit the code in `PlayQuoteActivity`:

```
using AndroidX.Fragment.App;

[Activity(Label = "PlayQuoteActivity")]
public class PlayQuoteActivity : FragmentActivity
{
    protected override void OnCreate(Bundle savedInstanceState)
    {
        base.OnCreate(savedInstanceState);

        var playId = Intent.Extras.GetInt("current_play_id", 0);

        var detailsFrag = PlayQuoteFragment.NewInstance(playId);
        SupportFragmentManager.BeginTransaction()
                        .Add(Android.Resource.Id.Content, detailsFrag)
                        .Commit();
    }
}
```

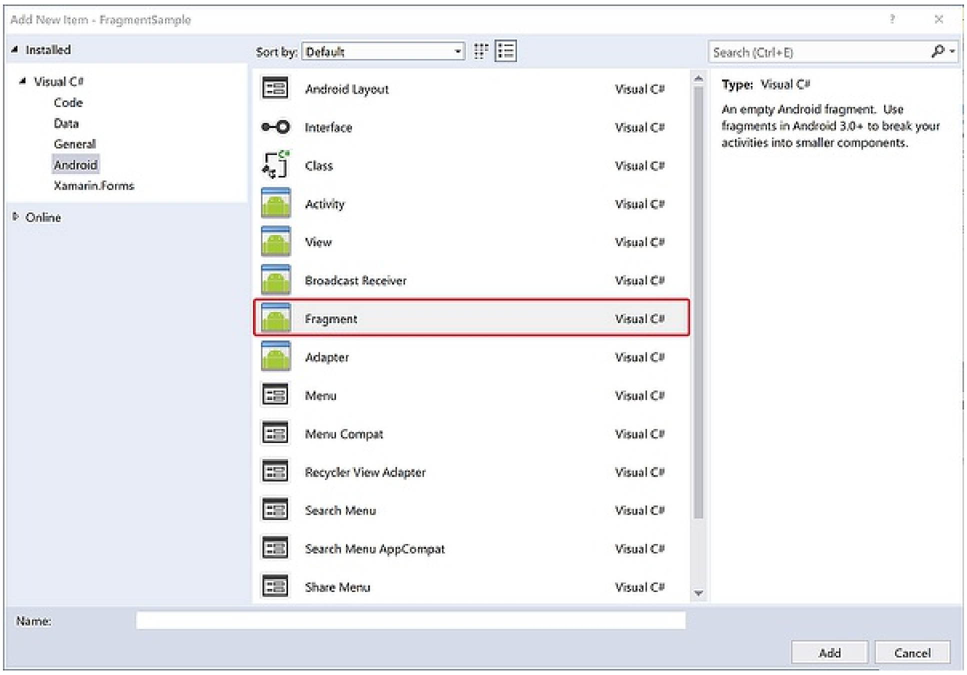
When `PlayQuoteActivity` is created, it will instantiate a new `PlayQuoteFragment` and load that fragment in its root view in the context of a `FragmentTransaction`. Notice that this activity does not load an Android layout file for its user interface. Instead, a new `PlayQuoteFragment` is added to the root view of the application. The resource identifier `Android.Resource.Id.Content` is used to refer to the root view of an Activity without knowing its specific identifier.

5. Create TitlesFragment

The `TitlesFragment` will subclass a specialized fragment known as a `ListFragment` which encapsulates the logic for displaying a `ListView` in a fragment. A `ListFragment` exposes a `ListAdapter` property (used by the `ListView` to display its contents) and an event handler named `OnListItemClick` which allows the fragment to respond to clicks on a row that is displayed by the `ListView`.

To get started, add a new fragment to the project and name it **TitlesFragment**:

- Visual Studio
- Visual Studio for Mac



Edit the code inside the fragment:

```
using AndroidX.Fragment.App;

public class TitlesFragment : ListFragment
{
    int selectedPlayId;

    public TitlesFragment()
    {
        // Being explicit about the requirement for a default constructor.
    }

    public override void OnCreate(Bundle savedInstanceState)
    {
        base.OnCreate(savedInstanceState);
        ListAdapter = new ArrayAdapter<String>(Activity, Android.Resource.Layout.SimpleListItemActivated1,
Shakespeare.Titles);

        if (savedInstanceState != null)
        {
            selectedPlayId = savedInstanceState.GetInt("current_play_id", 0);
        }
    }

    public override void OnSaveInstanceState(Bundle outState)
    {
        base.OnSaveInstanceState(outState);
        outState.PutInt("current_play_id", selectedPlayId);
    }

    public override void OnListItemClick(ListView l, View v, int position, long id)
    {
        ShowPlayQuote(position);
    }

    void ShowPlayQuote(int playId)
    {
        var intent = new Intent(Activity, typeof(PlayQuoteActivity));
        intent.PutExtra("current_play_id", playId);
        StartActivity(intent);
    }
}
```

When the Activity is created Android will invoke the `OnCreate` method of the fragment; this is where the list adapter for the `ListView` is created. The `ShowQuoteFromPlay` method will start an instance of the `PlayQuoteActivity` to display the quote for the selected play.

Display TitlesFragment in MainActivity

The final step is to display `TitlesFragment` within `MainActivity`. The Activity does not dynamically load the fragment. Instead the fragment will be statically loaded by declaring it in the layout file of the activity using a `fragment` element. The fragment to load is identified by setting the `android:name` attribute to the fragment class (including the namespace of the type). For example, to use the `TitlesFragment`, then `android:name` would be set to `FragmentSample.TitlesFragment`.

Edit the layout file **activity_main.axml**, replacing the existing XML with the following:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <fragment
        android:name="FragmentSample.TitlesFragment"
        android:id="@+id/titles"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

NOTE

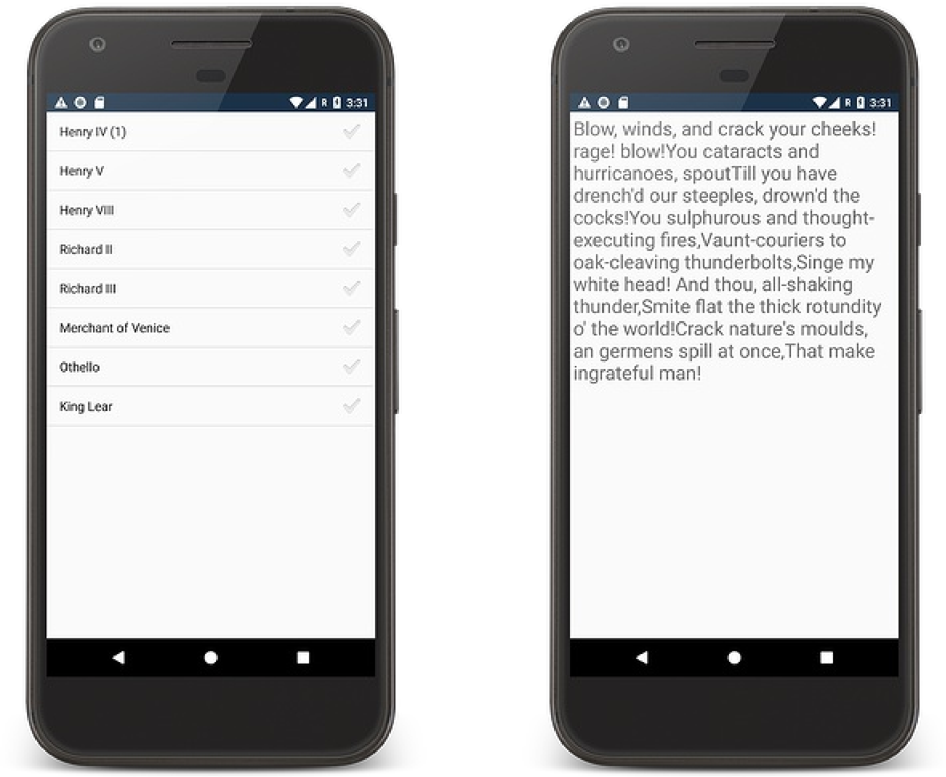
The `class` attribute is a valid substitute for `android:name`. There is no formal guidance on which form is preferred, there are many examples of code bases that will use `class` interchangeably with `android:name`.

There are no code changes required for MainActivity. The code in that class should be very similar to this snippet:

```
using AndroidX.Fragment.App;

[Activity(Label = "@string/app_name", Theme = "@style/AppTheme", MainLauncher = true)]
public class MainActivity : FragmentActivity
{
    protected override void OnCreate(Bundle savedInstanceState)
    {
        base.OnCreate(savedInstanceState);
        SetContentView(Resource.Layout.activity_main);
    }
}
```

Run the app

Now that the code is complete, run the app on a device to see it in action.

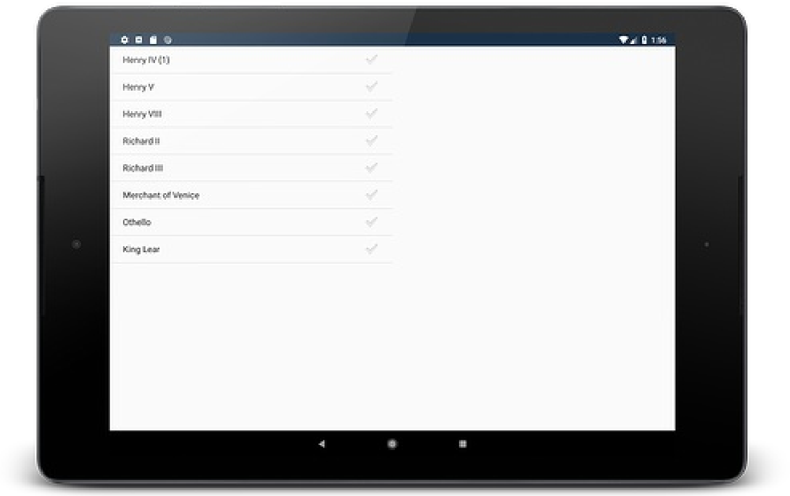


Part 2 of this walkthrough will optimtize this application for devices running in landscape mode.

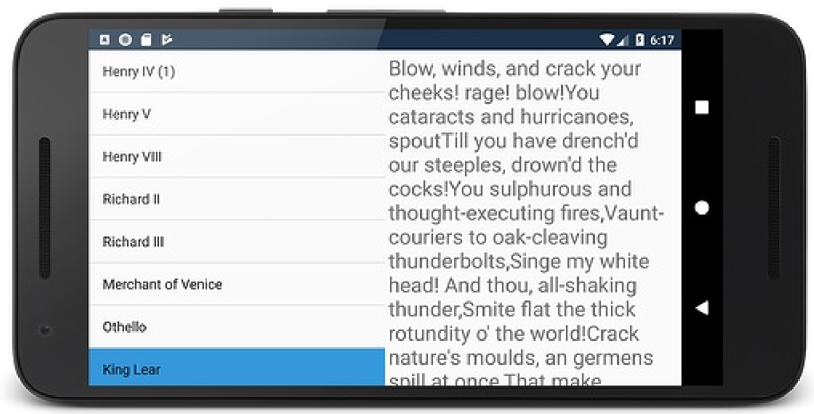
Fragments walkthrough – landscape

7/8/2021 • 5 minutes to read • Edit Online

The Fragments Walkthrough – Part 1 demonstrated how to create and use fragments in an Android app that targets the smaller screens on a phone. The next step in this walkthrough is to modify the application to take advantage of the extra horizontal space on tablet – there will be one activity that will always be the list of plays (the `TitlesFragment`) and `PlayQuoteFragment` will be dynamically added to the Activity in response to a selection made by the user:



Phones that are running in landscape mode will also benefit from this enhancement:



Updating the app to handle landscape orientation

The following modifications will build upon the work that was done in the Fragments Walkthrough - Phone

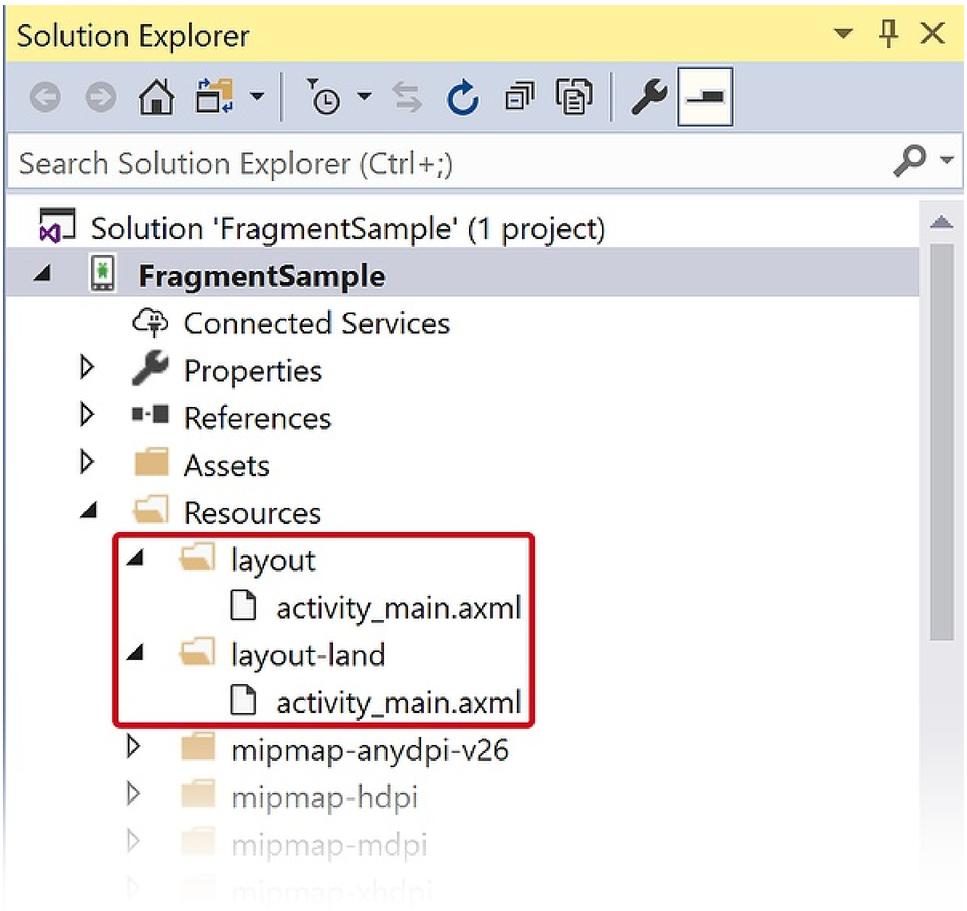
1. Create an alternate layout to display both the `TitlesFragment` and `PlayQuoteFragment`.
2. Update `TitlesFragment` to detect if the device is displaying both fragments simultaneously and change behavior accordingly.
3. Update `PlayQuoteActivity` to close when the device is in landscape mode.

1. Create an alternate layout

When Main Activity is created on an Android device, Android will decide which layout to load based on the orientation of the device. By default, Android will provide the **Resources/layout/activity_main.axml** layout file. For devices that load in landscape mode Android will provide the **Resources/layout-land/activity_main.axml** layout file. The guide on Android Resources contains more details on how Android decides what resource files to load for an application.

Create an alternate layout that targets **Landscape** orientation by following the steps described in the Alternate Layouts guide. This should add a new layout resource file to the project, **Resources/layout/activity_main.axml**:

- Visual Studio
- Visual Studio for Mac



After creating the alternate layout, edit the source of the file **Resources/layout-land/activity_main.axml** so that it matches this XML:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/two_fragments_layout"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <fragment android:name="FragmentSample.TitlesFragment"
        android:id="@+id/titles"
        android:layout_weight="1"
        android:layout_width="0px"
        android:layout_height="match_parent" />

    <FrameLayout android:id="@+id/playquote_container"
            android:layout_weight="1"
            android:layout_width="0px"
            android:layout_height="match_parent"
             />
</LinearLayout>
```

The root view of the activity is given the resource ID `two_fragments_layout` and has two sub-views, a `fragment` and a `FrameLayout`. While the `fragment` is statically loaded, the `FrameLayout` acts as a "placeholder" that will be replaced at run-time by the `PlayQuoteFragment`. Each time a new play is selected in the `TitlesFragment`, the `playquote_container` will be updated with a new instance of the `PlayQuoteFragment`.

Each of the sub-views will occupy the full height of their parent. The width of each subview is controlled by the `android:layout_weight` and `android:layout_width` attributes. In this example, each subview will occupy 50% of width provide by the parent. See Google's document on the LinearLayout for details about *Layout Weight*.

2. Changes to TitlesFragment

Once the alternate layout has been created, it is necessary to update `TitlesFragment`. When the app is displaying the two fragments on one activity, then `TitlesFragment` should load the `PlayQuoteFragment` in the parent Activity. Otherwise, `TitlesFragment` should launch the `PlayQuoteActivity` which host the `PlayQuoteFragment`. A boolean flag will help `TitlesFragment` determine which behavior it should use. This flag will be initialized in the `OnActivityCreated` method.

First, add an instance variable at the top of the `TitlesFragment` class:

```
bool showingTwoFragments;
```

Then, add the following code snippet to `OnActivityCreated` to initialize the variable:

```
var quoteContainer = Activity.FindViewById(Resource.Id.playquote_container);
showingTwoFragments = quoteContainer != null &&
                      quoteContainer.Visibility == ViewStates.Visible;
if (showingTwoFragments)
{
    ListView.ChoiceMode = ChoiceMode.Single;
    ShowPlayQuote(selectedPlayId);
}
```

If the device is running in landscape mode, then the `FrameLayout` with the resource ID `playquote_container` will be visible on the screen, so `showingTwoFragments` will be initialized to `true`. If the device is running in portrait mode, then `playquote_container` will not be on the screen, so `showingTwoFragments` will be `false`.

The `ShowPlayQuote` method will need to change how it displays a quote – either in a fragment or launch a new activity. Update the `ShowPlayQuote` method to load a fragment when showing two fragments, otherwise it should launch an Activity:

```
void ShowPlayQuote(int playId)
{
    selectedPlayId = playId;
    if (showingTwoFragments)
    {
        ListView.SetItemChecked(selectedPlayId, true);

        var playQuoteFragment = FragmentManager.FindFragmentById(Resource.Id.playquote_container) as
PlayQuoteFragment;

        if (playQuoteFragment == null || playQuoteFragment.PlayId != playId)
        {
            var container = Activity.FindViewById(Resource.Id.playquote_container);
            var quoteFrag = PlayQuoteFragment.NewInstance(selectedPlayId);

            FragmentTransaction ft = FragmentManager.BeginTransaction();
            ft.Replace(Resource.Id.playquote_container, quoteFrag);
            ft.Commit();
        }
    }
    else
    {
        var intent = new Intent(Activity, typeof(PlayQuoteActivity));
        intent.PutExtra("current_play_id", playId);
        StartActivity(intent);
    }
}
```

If the user has selected a play that is different from the one that is currently being displayed in `PlayQuoteFragment`, then a new `PlayQuoteFragment` is created and will replace the contents of the `playquote_container` within the context of a `FragmentTransaction`.

Complete code for TitlesFragment

After completing all the previous changes to `TitlesFragment`, the complete class should match this code:

```
public class TitlesFragment : ListFragment
{
    int selectedPlayId;
    bool showingTwoFragments;

    public override void OnActivityCreated(Bundle savedInstanceState)
    {
        base.OnActivityCreated(savedInstanceState);
        ListAdapter = new ArrayAdapter<string>(Activity, Android.Resource.Layout.SimpleListItemActivated1,
Shakespeare.Titles);

        if (savedInstanceState != null)
        {
            selectedPlayId = savedInstanceState.GetInt("current_play_id", 0);
        }

        var quoteContainer = Activity.FindViewById(Resource.Id.playquote_container);
        showingTwoFragments = quoteContainer != null &&
                                quoteContainer.Visibility == ViewStates.Visible;
        if (showingTwoFragments)
        {
            ListView.ChoiceMode = ChoiceMode.Single;
            ShowPlayQuote(selectedPlayId);
        }
    }
```

```
    public override void OnSaveInstanceState(Bundle outState)
    {
        base.OnSaveInstanceState(outState);
        outState.PutInt("current_play_id", selectedPlayId);
    }

    public override void OnListItemClick(ListView l, View v, int position, long id)
    {
        ShowPlayQuote(position);
    }

    void ShowPlayQuote(int playId)
    {
        selectedPlayId = playId;
        if (showingTwoFragments)
        {
            ListView.SetItemChecked(selectedPlayId, true);

            var playQuoteFragment = FragmentManager.FindFragmentById(Resource.Id.playquote_container) as
PlayQuoteFragment;

            if (playQuoteFragment == null || playQuoteFragment.PlayId != playId)
            {
                var container = Activity.FindViewById(Resource.Id.playquote_container);
                var quoteFrag = PlayQuoteFragment.NewInstance(selectedPlayId);

                FragmentTransaction ft = FragmentManager.BeginTransaction();
                ft.Replace(Resource.Id.playquote_container, quoteFrag);
                ft.AddToBackStack(null);
                ft.SetTransition(FragmentTransit.FragmentFade);
                ft.Commit();
            }
        }
        else
        {
            var intent = new Intent(Activity, typeof(PlayQuoteActivity));
            intent.PutExtra("current_play_id", playId);
            StartActivity(intent);
        }
    }
}
```

3. Changes to PlayQuoteActivity

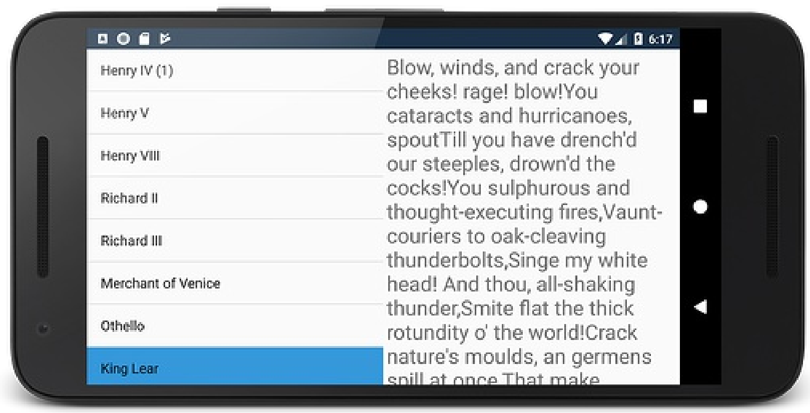
There is one final detail to take care of: `PlayQuoteActivity` is not necessary when the device is in landscape mode. If the device is in landscape mode the `PlayQuoteActivity` should not be visible. Update the `OnCreate` method of `PlayQuoteActivity` so that it will close itself. This code is the final version of `PlayQuoteActivity.OnCreate`:

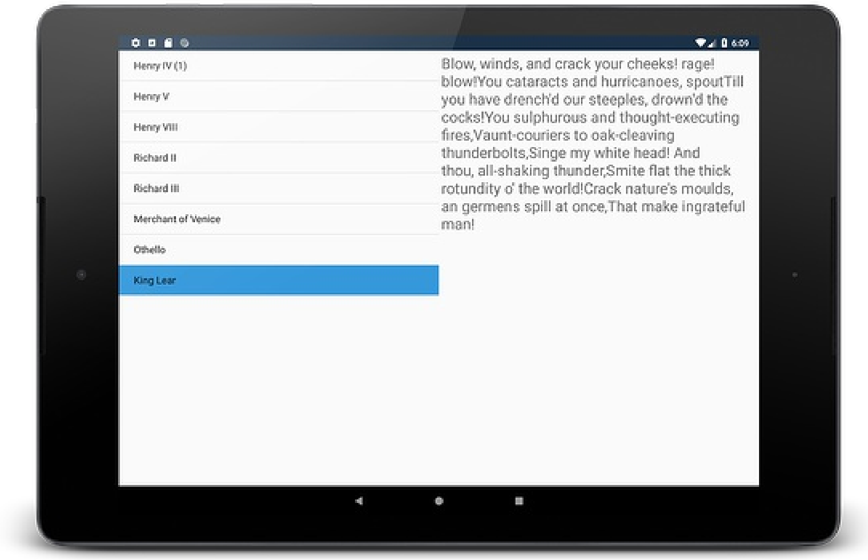
```
protected override void OnCreate(Bundle savedInstanceState)
{
    base.OnCreate(savedInstanceState);

    if (Resources.Configuration.Orientation == Android.Content.Res.Orientation.Landscape)
    {
        Finish();
    }

    var playId = Intent.Extras.GetInt("current_play_id", 0);
    var playQuoteFrag = PlayQuoteFragment.NewInstance(playId);
    FragmentManager.BeginTransaction()
                    .Add(Android.Resource.Id.Content, playQuoteFrag)
                    .Commit();
}
```

This modification adds a check for the device orientation. If it is in landscape mode, then `PlayQuoteActivity` will close itself.

4. Run the application

Once these changes are complete, run the app, rotate the device to landscape mode (if necessary), and then select a play. The quote should be displayed on the same screen as the list of plays:




Henry IV (1)
Henry V
Henry VIII
Richard II
Richard III
Merchant of Venice
Othello
King Lear
Blow, winds, and crack your cheeks! rage! blow!You cataracts and hurricanoes, spoutTill you have drench'd our steeples, drown'd the cocks!You sulphurous and thought-executing fires,Vaunt-couriers to oak-cleaving thunderbolts,Singe my white head! And thou, all-shaking thunder,Smite flat the thick rotundity o' the world!Crack nature's moulds, an germens spill at once,That make ingrateful man!

Creating A Fragment

7/8/2021 • 9 minutes to read • Edit Online

To create a Fragment, a class must inherit from `Android.App.Fragment` and then override the `OnCreateView` method. `OnCreateView` will be called by the hosting Activity when it is time to put the Fragment on the screen, and will return a `View`. A typical `OnCreateView` will create this `View` by inflating a layout file and then attaching it to a parent container. The container's characteristics are important as Android will apply the layout parameters of the parent to the UI of the Fragment. The following example illustrates this:

```
public override View OnCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)
{
    return inflater.Inflate(Resource.Layout.Example_Fragment, container, false);
}
```

The code above will inflate the view `Resource.Layout.Example_Fragment`, and add it as a child view to the `ViewGroup` container.

> **NOTE**
>
> Fragment sub-classes must have a public default no argument constructor.

Adding a Fragment to an Activity

There are two ways that a Fragment may be hosted inside an Activity:

- **Declaratively** – Fragments can be used declaratively within `.axml` layout files by using the `<Fragment>` tag.
- **Programmatically** – Fragments can also be instantiated dynamically by using the `FragmentManager` class's API.

Programmatic usage via the `FragmentManager` class will be discussed later in this guide.

Using a Fragment Declaratively

Adding a Fragment inside the layout requires using the `<fragment>` tag and then identifying the Fragment by providing either the `class` attribute or the `android:name` attribute. The following snippet shows how to use the `class` attribute to declare a `fragment`:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment class="com.xamarin.sample.fragments.TitlesFragment"
            android:id="@+id/titles_fragment"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent" />
```

This next snippet shows how to declare a `fragment` by using the `android:name` attribute to identify the Fragment class :

```xml
<?xml version="1.0" encoding="utf-8"?>
<fragment android:name="com.xamarin.sample.fragments.TitlesFragment"
            android:id="@+id/titles_fragment"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent" />
```

When the Activity is being created, Android will instantiate each Fragment specified in the layout file and insert the view that is created from `OnCreateView` in place of the `Fragment` element. Fragments that are declaratively added to an Activity are static and will remain on the Activity until it is destroyed; it is not possible to dynamically replace or remove such a Fragment during the lifetime of the Activity to which it is attached.

Each Fragment must be assigned a unique identifier:

- **android:id** – As with other UI elements in a layout file, this is a unique ID.
- **android:tag** – This attribute is a unique string.

If neither of the previous two methods is used, then the Fragment will assume the ID of the container view. In the following example where neither `android:id` nor `android:tag` is provided, Android will assign the ID `fragment_container` to the Fragment:

```xml
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
                android:id="+@id/fragment_container"
                android:orientation="horizontal"
                android:layout_width="match_parent"
                android:layout_height="match_parent">

        <fragment class="com.example.android.apis.app.TitlesFragment"
                android:layout_width="match_parent"
                android:layout_height="match_parent" />
</LinearLayout>
```

Package Name Case

Android does not allow for uppercase characters in package names; it will throw an exception when trying to inflate the view if a package name contains an uppercase character. However, Xamarin.Android is more forgiving, and will tolerate uppercase characters in the namespace.

For example, both of the following snippets will work with Xamarin.Android. However, the second snippet will cause an `android.view.InflateException` to be thrown by a pure Java-based Android application.

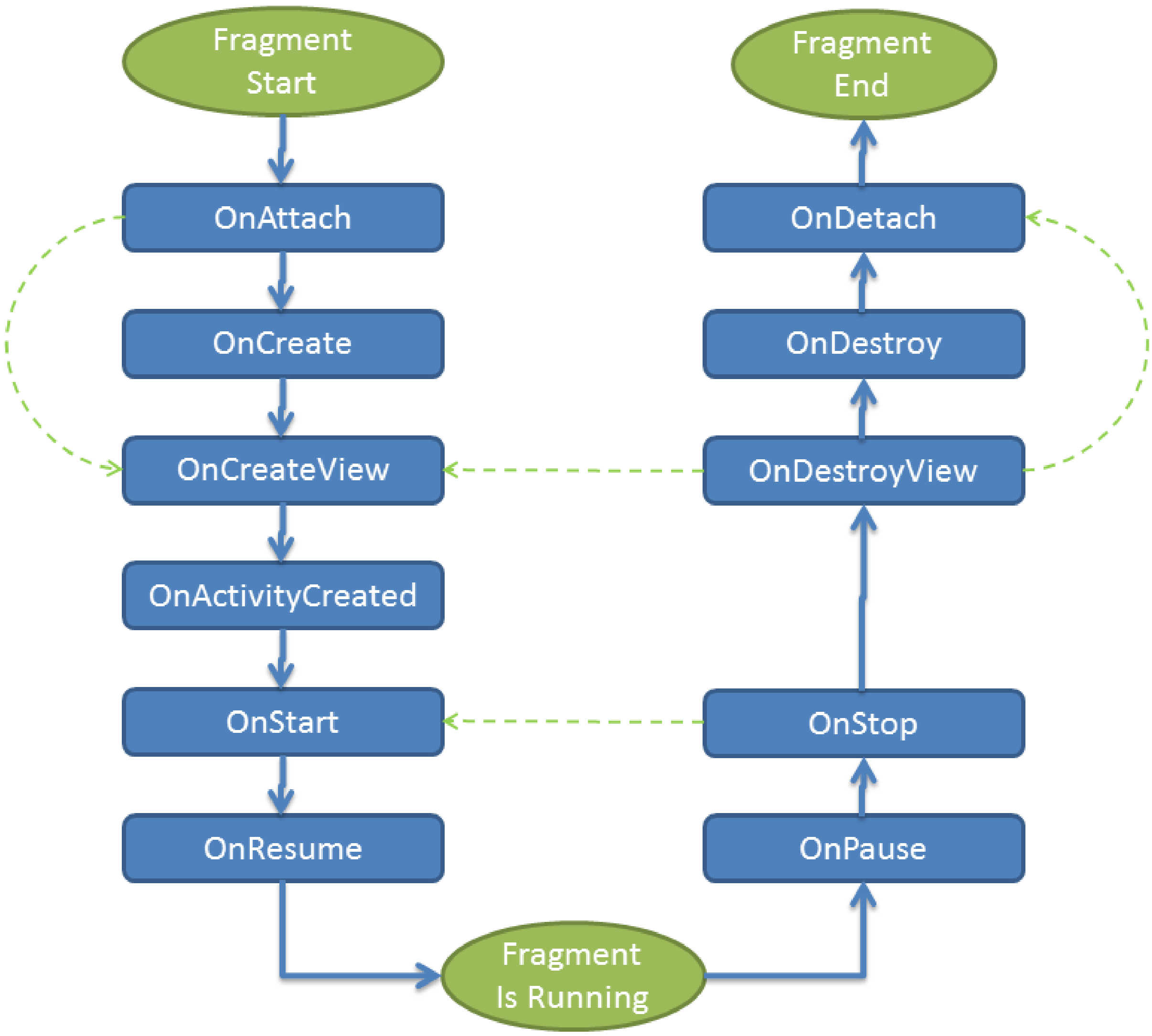
```xml
<fragment class="com.example.DetailsFragment" android:id="@+id/fragment_content"
android:layout_width="match_parent" android:layout_height="match_parent" />
```

OR

```xml
<fragment class="Com.Example.DetailsFragment" android:id="@+id/fragment_content"
android:layout_width="match_parent" android:layout_height="match_parent" />
```

Fragment Lifecycle

Fragments have their own lifecycle that is somewhat independent of, but still affected by, the lifecycle of the hosting Activity. For example, when an Activity pauses, all of its associated Fragments are paused. The following diagram outlines the lifecycle of the Fragment.



Fragment Creation Lifecycle Methods

The list below shows the flow of the various callbacks in the lifecycle of a Fragment as it is being created:

- `OnInflate()` – Called when the Fragment is being created as part of a view layout. This may be called immediately after the Fragment is created declaratively from an XML layout file. The Fragment is not associated with its Activity yet, but the **Activity**, **Bundle**, and **AttributeSet** from the view hierarchy are passed in as parameters. This method is best used for parsing the **AttributeSet** and for saving the attributes that might be used later by the Fragment.

- `OnAttach()` – Called after the Fragment is associated with the Activity. This is the first method to be run when the Fragment is ready to be used. In general, Fragments should not implement a constructor or override the default constructor. Any components that are required for the Fragment should be initialized in this method.

- `OnCreate()` – Called by the Activity to create the Fragment. When this method is called, the view hierarchy of the hosting Activity may not be completely instantiated, so the Fragment should not rely on any parts of the Activity's view hierarchy until later on in the Fragment's lifecycle. For example, do not use this method to perform any tweaks or adjustments to the UI of the application. This is the earliest time at which the Fragment may begin gathering the data that it needs. The Fragment is running in the UI thread at this point, so avoid any lengthy processing, or perform that processing on a background thread. This method may be skipped if **SetRetainInstance(true)** is called. This alternative will be described in more detail below.

- `OnCreateView()` – Creates the view for the Fragment. This method is called once the Activity's **OnCreate()** method is complete. At this point, it is safe to interact with the view hierarchy of the Activity.

This method should return the view that will be used by the Fragment.

- `OnActivityCreated()` – Called after **Activity.OnCreate** has been completed by the hosting Activity. Final tweaks to the user interface should be performed at this time.

- `OnStart()` – Called after the containing Activity has been resumed. This makes the Fragment visible to the user. In many cases, the Fragment will contain code that would otherwise be in the **OnStart()** method of an Activity.

- `OnResume()` – This is the last method called before the user can interact with the Fragment. An example of the kind of code that should be performed in this method would be enabling features of a device that the user may interact with, such as the camera that the location services. Services such as these can cause excessive battery drain, though, and an application should minimize their use to preserve battery life.

Fragment Destruction Lifecycle Methods

The next list explains the lifecycle methods that are called as a Fragment is being destroyed:

- `OnPause()` – The user is no longer able to interact with the Fragment. This situation exists because some other Fragment operation is modifying this Fragment, or the hosting Activity is paused. It is possible that the Activity hosting this Fragment might still be visible, that is, the Activity in focus is partially transparent or does not occupy the full screen. When this method becomes active, it's the first indication that the user is leaving the Fragment. The Fragment should save any changes.

- `OnStop()` – The Fragment is no longer visible. The host Activity may be stopped, or a Fragment operation is modifying it in the Activity. This callback serves the same purpose as **Activity.OnStop**.

- `OnDestroyView()` – This method is called to clean up resources associated with the view. This is called when the view associated with the Fragment has been destroyed.

- `OnDestroy()` – This method is called when the Fragment is no longer in use. It is still associated with the Activity, but the Fragment is no longer functional. This method should release any resources that are in use by the Fragment, such as a SurfaceView that might be used for a camera. This method may be skipped if **SetRetainInstance(true)** is called. This alternative will be described in more detail below.

- `OnDetach()` – This method is called just before the Fragment is no longer associated with the Activity. The view hierarchy of the Fragment no longer exists, and all resources that are used by the Fragment should be released at this point.

Using SetRetainInstance

It is possible for a Fragment to specify that it should not be completely destroyed if the Activity is being re-created. The `Fragment` class provides the method `SetRetainInstance` for this purpose. If `true` is passed to this method, then when the Activity is restarted, the same instance of the Fragment will be used. If this happens, then all callback methods will be invoked except the `OnCreate` and `OnDestroy` lifecycle callbacks. This process is illustrated in the lifecycle diagram shown above (by the green dotted lines).

Fragment State Management

Fragments may save and restore their state during the Fragment lifecycle by using an instance of a `Bundle`. The Bundle allows a Fragment to save data as key/value pairs and is useful for simple data that doesn't require much memory. A Fragment can save its state with a call to `OnSaveInstanceState`:

```
public override void OnSaveInstanceState(Bundle outState)
{
    base.OnSaveInstanceState(outState);
    outState.PutInt("current_choice", _currentCheckPosition);
}
```

When a new instance of a Fragment is created, the state saved in the `Bundle` will become available to the new instance via the `OnCreate`, `OnCreateView`, and `OnActivityCreated` methods of the new instance. The following sample demonstrates how to retrieve the value `current_choice` from the `Bundle`:

```
public override void OnActivityCreated(Bundle savedInstanceState)
{
    base.OnActivityCreated(savedInstanceState);
    if (savedInstanceState != null)
    {
        _currentCheckPosition = savedInstanceState.GetInt("current_choice", 0);
    }
}
```

Overriding `OnSaveInstanceState` is an appropriate mechanism for saving transient data in a Fragment across orientation changes, such as the `current_choice` value in the above example. However, the default implementation of `OnSaveInstanceState` takes care of saving transient data in the UI for every view that has an ID assigned. For example, look at an application that has an `EditText` element defined in XML as follows:

```
<EditText android:id="@+id/myText"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"/>
```

Since the `EditText` control has an `id` assigned, the Fragment automatically saves the data in the widget when `OnSaveInstanceState` is called.

Bundle Limitations

Although using `OnSaveInstanceState` makes it easy to save transient data, use of this method has some limitations:

- If the Fragment is not added to the back stack, then its state will not be restored when the user presses the **Back** button.
- When the Bundle is used to save data, that data is serialized. This can lead to processing delays.

Contributing to the Menu

Fragments may contribute items to the menu of their hosting Activity. An Activity handles menu items first. If the Activity does not have a handler, then the event will be passed on to the Fragment, which will then handle it.

To add items to the Activity's menu, a Fragment must do two things. First, the Fragment must implement the method `OnCreateOptionsMenu` and place its items into the menu, as shown in the following code:

```
public override void OnCreateOptionsMenu(IMenu menu, MenuInflater menuInflater)
{
    menuInflater.Inflate(Resource.Menu.menu_fragment_vehicle_list, menu);
    base.OnCreateOptionsMenu(menu, menuInflater);
}
```

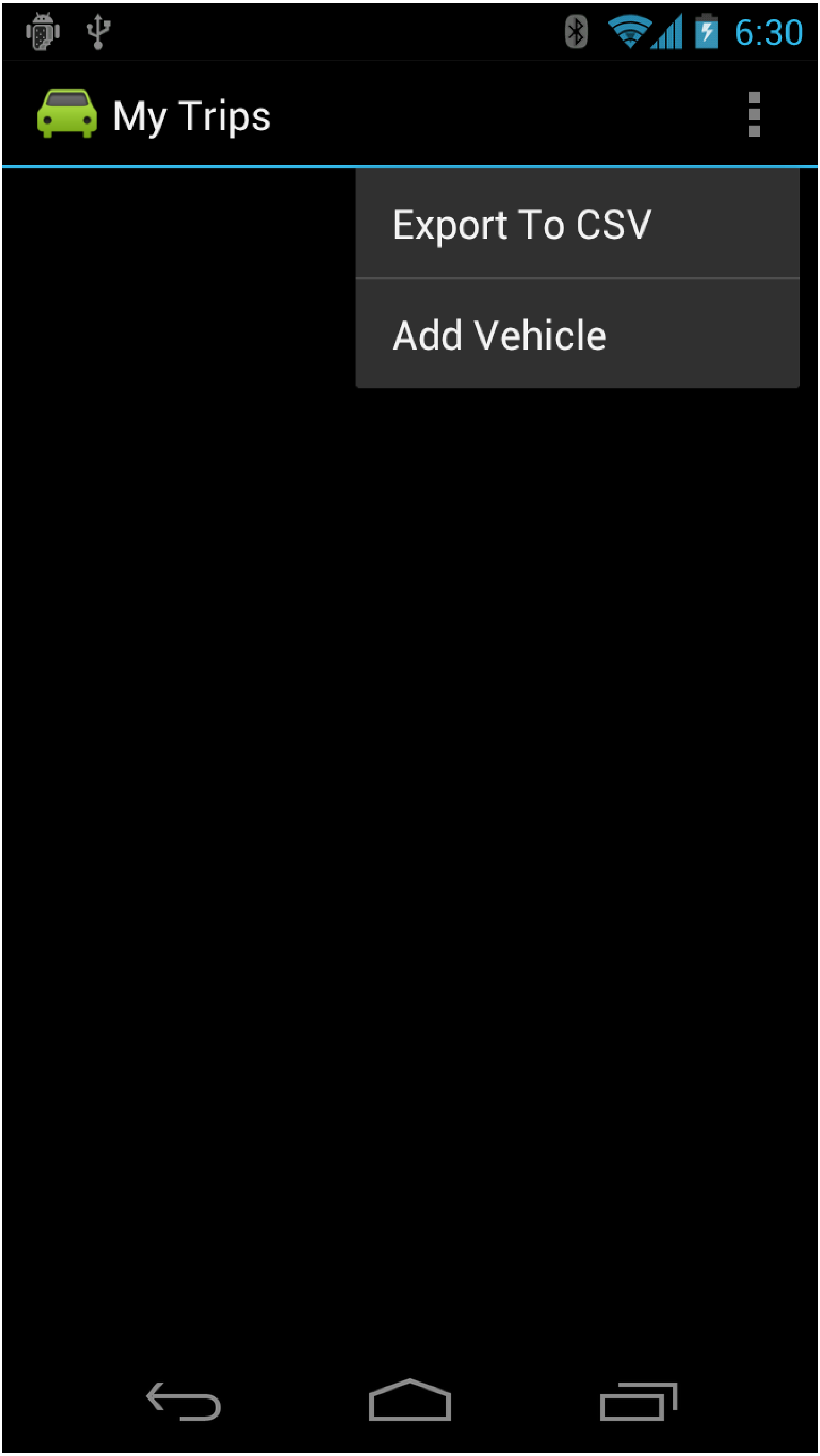
The menu in the previous code snippet is inflated from the following XML, located in the file `menu_fragment_vehicle_list.xml`:

```xml
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/add_vehicle"
        android:icon="@drawable/ic_menu_add_data"
        android:title="@string/add_vehicle" />
</menu>
```

Next, the Fragment must call `SetHasOptionsMenu(true)`. The call to this method announces to Android that the Fragment has menu items to contribute to the option menu. Unless the call to this method is made, the menu items for the Fragment will not be added to the Activity's option menu. This is typically done in the lifecycle method `OnCreate()`, as shown in the next code snippet:

```
public override void OnCreate(Bundle savedState)
{
    base.OnCreate(savedState);
    SetHasOptionsMenu(true);
}
```

The following screen shows how this menu would look:


6:30
My Trips
Export To CSV
Add Vehicle

Managing Fragments

10/28/2019 • 2 minutes to read • Edit Online

To help with managing Fragments, Android provides the `FragmentManager` class. Each Activity has an instance of `Android.App.FragmentManager` that will find or dynamically change its Fragments. Each set of these changes is known as a *transaction*, and is performed by using one of the APIs contained in the class `Android.App.FragmentTransation`, which is managed by the `FragmentManager`. An Activity may start a transaction like this:

```
FragmentTransaction fragmentTx = this.FragmentManager.BeginTransaction();
```

These changes to the Fragments are performed in the `FragmentTransaction` instance by using methods such as `Add()`, `Remove(),` and `Replace().` The changes are then applied by using `Commit()`. The changes in a transaction are not performed immediately. Instead, they are scheduled to run on the Activity's UI thread as soon as possible.

The following example shows how to add a Fragment to an existing container:

```
// Create a new fragment and a transaction.
FragmentTransaction fragmentTx = this.FragmentManager.BeginTransaction();
DetailsFragment aDifferentDetailsFrag = new DetailsFragment();

// The fragment will have the ID of Resource.Id.fragment_container.
fragmentTx.Add(Resource.Id.fragment_container, aDifferentDetailsFrag);

// Commit the transaction.
fragmentTx.Commit();
```

If a transaction is committed after `Activity.OnSaveInstanceState()` is called, an exception will be thrown. This happens because when the Activity saves its state, Android also saves the state of any hosted Fragments. If any Fragment transactions are committed after this point, the state of these transactions will be lost when the Activity is restored.

It's possible to save the Fragment transactions to the Activity's back stack by making a call to `FragmentTransaction.AddToBackStack()`. This allows the user to navigate backwards through Fragment changes when the **Back** button is pressed. Without a call to this method, Fragments that are removed will be destroyed and will be unavailable if the user navigates back through the Activity.

The following example shows how to use the `AddToBackStack` method of a `FragmentTransaction` to replace one Fragment, while preserving the state of the first Fragment on the back stack:

```
// Create a new fragment and a transaction.
FragmentTransaction fragmentTx = this.FragmentManager.BeginTransaction();
DetailsFragment aDifferentDetailsFrag = new DetailsFragment();

// Replace the fragment that is in the View fragment_container (if applicable).
fragmentTx.Replace(Resource.Id.fragment_container, aDifferentDetailsFrag);

// Add the transaction to the back stack.
fragmentTx.AddToBackStack(null);

// Commit the transaction.
fragmentTx.Commit();
```

Communicating with Fragments

The *FragmentManager* knows about all of the Fragments that are attached to an Activity and provides two methods to help find these Fragments:

- **FindFragmentById** – This method will find a Fragment by using the ID that was specified in the layout file or the container ID when the Fragment was added as part of a transaction.
- **FindFragmentByTag** – This method is used to find a Fragment that has a tag that was provided in the layout file or that was added in a transaction.

Both Fragments and Activities reference the `FragmentManager`, so the same techniques are used to communicate back and forth between them. An application may find a reference Fragment by using one of these two methods, cast that reference to the appropriate type, and then directly call methods on the Fragment. The following snippet provides an example:

It is also possible for the Activity to use the `FragmentManager` to find Fragments:

```
var emailList = FragmentManager.FindFragmentById<EmailListFragment>(Resource.Id.email_list_fragment);
emailList.SomeCustomMethod(parameter1, parameter2);
```

Communicating with the Activity

It is possible for a Fragment to use the `Fragment.Activity` property to reference its host. By casting the Activity to a more specific type, it is possible for an Activity to call methods and properties on its host, as shown in the following example:

```
var myActivity = (MyActivity) this.Activity;
myActivity.SomeCustomMethod();
```

Specialized Fragment Classes

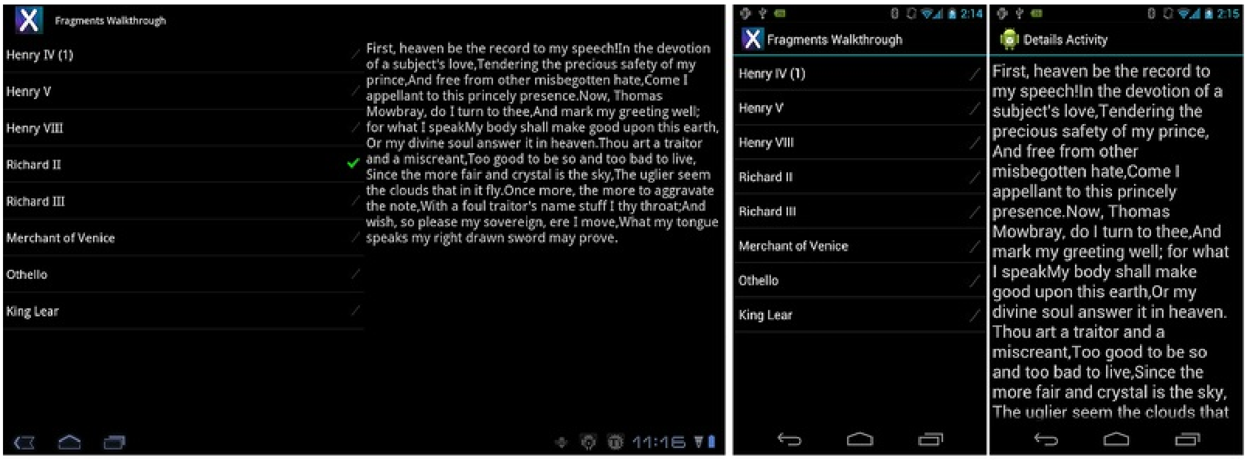
7/8/2021 • 6 minutes to read • Edit Online

The Fragments API provides other subclasses that encapsulate some of the more common functionality found in applications. These subclasses are:

- **ListFragment** – This Fragment is used to display a list of items bound to a datasource such as an array or a cursor.
- **DialogFragment** – This Fragment is used as a wrapper around a dialog. The Fragment will display the dialog on top of its Activity.
- **PreferenceFragment** – This Fragment is used to show Preference objects as lists.

The ListFragment

The `ListFragment` is very similar in concept and functionality to the `ListActivity`; it is a wrapper that hosts a `ListView` in a Fragment. The image below shows a `ListFragment` running on a tablet and a phone:



Tablet Phone

Binding Data With The ListAdapter

The `ListFragment` class already provides a default layout, so it is not necessary to override `OnCreateView` to display the contents of the `ListFragment`. The `ListView` is bound to data by using a `ListAdapter` implementation. The following example shows how this could be done by using a simple array of strings:

```
public override void OnActivityCreated(Bundle savedInstanceState)
{
    base.OnActivityCreated(savedInstanceState);
    string[] values = new[] { "Android", "iPhone", "WindowsMobile",
                "Blackberry", "WebOS", "Ubuntu", "Windows7", "Max OS X",
                "Linux", "OS/2" };
    this.ListAdapter = new ArrayAdapter<string>(Activity, Android.Resource.Layout.SimpleExpandableListItem1,
values);
}
```

When setting the `ListAdapter`, it is important to use the `ListFragment.ListAdapter` property, and not the `ListView.ListAdapter` property. Using `ListView.ListAdapter` will cause important initialization code to be skipped.

Responding to User Selection

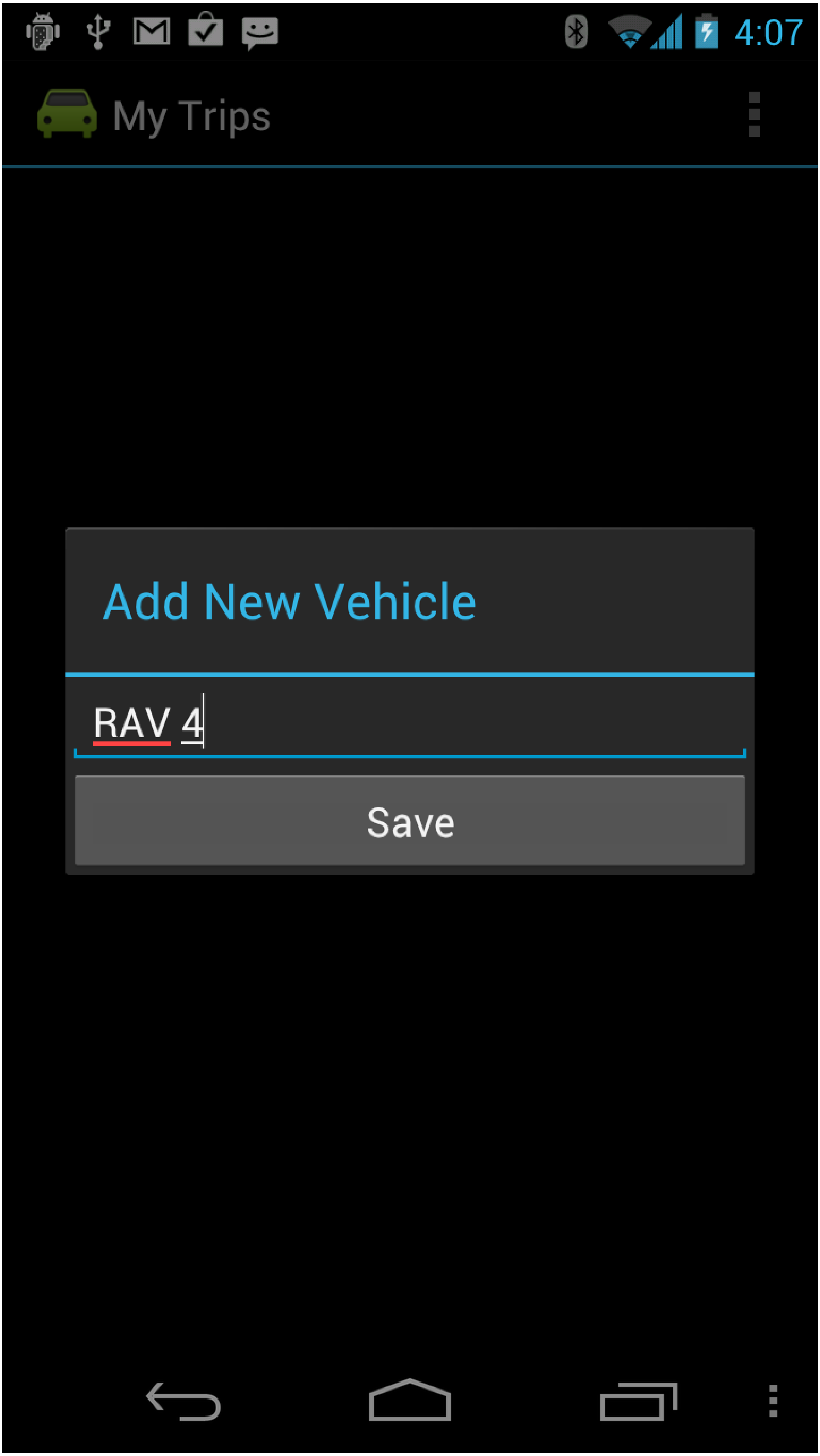
To respond to user selections, an application must override the `OnListItemClick` method. The following example shows one such possibility:

```
public override void OnListItemClick(ListView l, View v, int index, long id)
{
    // We can display everything in place with fragments.
    // Have the list highlight this item and show the data.
    ListView.SetItemChecked(index, true);

    // Check what fragment is shown, replace if needed.
    var details = FragmentManager.FindFragmentById<DetailsFragment>(Resource.Id.details);
    if (details == null || details.ShownIndex != index)
    {
        // Make new fragment to show this selection.
        details = DetailsFragment.NewInstance(index);

        // Execute a transaction, replacing any existing
        // fragment with this one inside the frame.
        var ft = FragmentManager.BeginTransaction();
        ft.Replace(Resource.Id.details, details);
        ft.SetTransition(FragmentTransit.FragmentFade);
        ft.Commit();
    }
}
```

In the code above, when the user selects an item in the `ListFragment`, a new Fragment is displayed in the hosting Activity, showing more details about the item that was selected.

DialogFragment

The *DialogFragment* is a Fragment that is used to display a dialog object inside of a Fragment that will float on top of the Activity's window. It is meant to replace the managed dialog APIs (starting in Android 3.0). The following screenshot shows an example of a `DialogFragment`:


4:07
My Trips
Add New Vehicle
RAV 4
Save

A `DialogFragment` ensures that the state between the Fragment and the dialog remain consistent. All interactions and control of the dialog object should happen through the `DialogFragment` API, and not be made with direct calls on the dialog object. The `DialogFragment` API provides each instance with a `Show()` method that is used to display a Fragment. There are two ways to get rid of a Fragment:

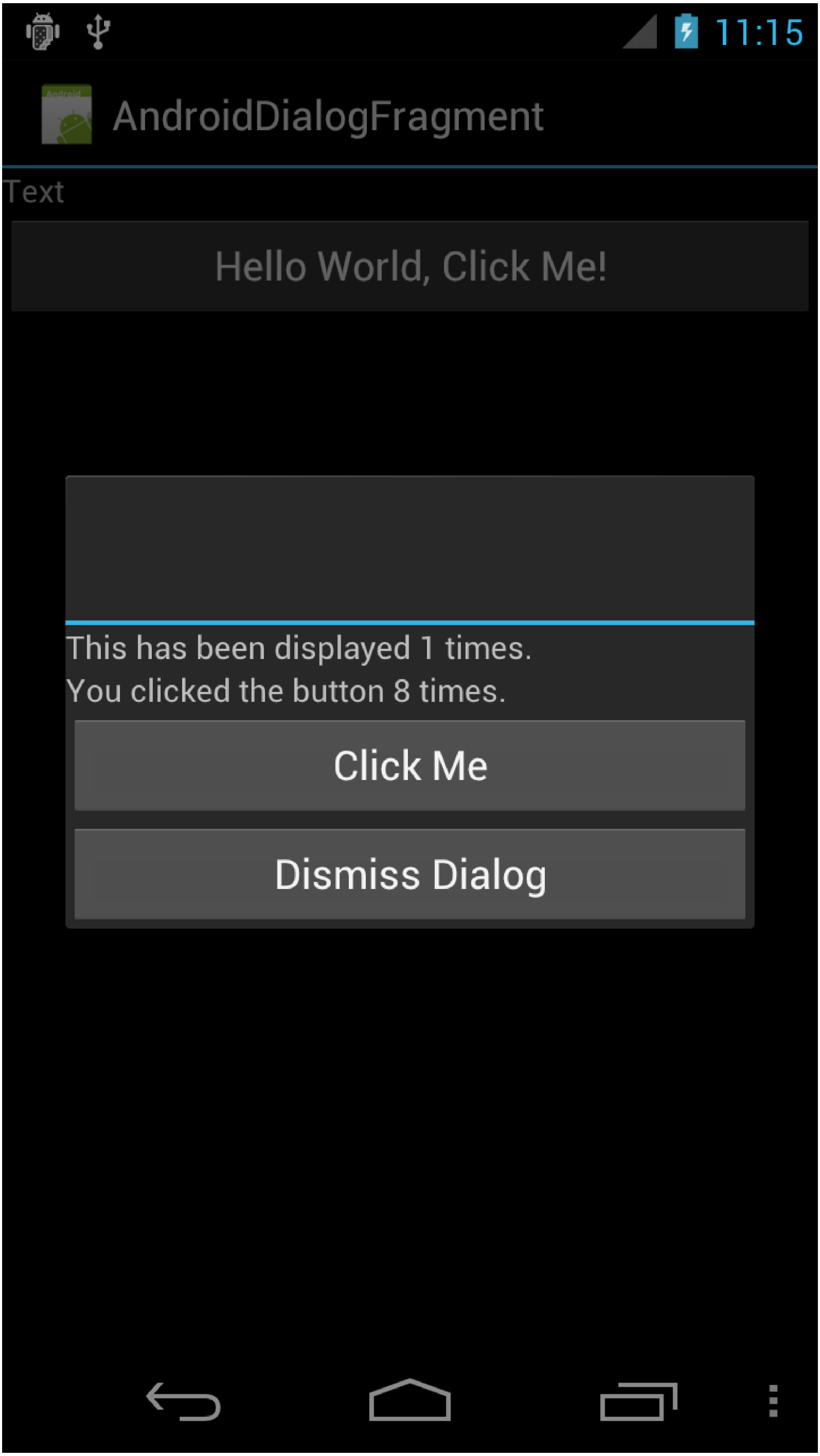
- Call `DialogFragment.Dismiss()` on the `DialogFragment` instance.
- Display another `DialogFragment`.

To create a `DialogFragment`, a class inherits from `Android.App.DialogFragment,` and then overrides one of the following two methods:

- **OnCreateView** – This creates and returns a view.
- **OnCreateDialog** – This creates a custom dialog. It is typically used to show an *AlertDialog*. When overriding this method, it is not necessary to override `OnCreateView` .

A Simple DialogFragment

The following screenshot shows a simple `DialogFragment` that has a `TextView` and two `Button`s:


11:15
AndroidDialogFragment
Text
Hello World, Click Me!
This has been displayed 1 times.
You clicked the button 8 times.
Click Me
Dismiss Dialog

The `TextView` will display the number of times that the user has clicked one button in the `DialogFragment`, while clicking the other button will close the Fragment. The code for `DialogFragment` is:

```
public class MyDialogFragment : DialogFragment
{
    private int _clickCount;
    public override void OnCreate(Bundle savedInstanceState)
    {
        _clickCount = 0;
    }

    public override View OnCreateView(LayoutInflater inflater, ViewGroup container, Bundle
savedInstanceState)
    {
        base.OnCreate(savedInstanceState)

        var view = inflater.Inflate(Resource.Layout.dialog_fragment_layout, container, false);
        var textView = view.FindViewById<TextView>(Resource.Id.dialog_text_view);

        view.FindViewById<Button>(Resource.Id.dialog_button).Click += delegate
            {
                textView.Text = "You clicked the button " + _clickCount++ + " times.";
            };

        // Set up a handler to dismiss this DialogFragment when this button is clicked.
        view.FindViewById<Button>(Resource.Id.dismiss_dialog_button).Click += (sender, args) => Dismiss();
        return view;
        }
    }
}
```

Displaying a Fragment

Like all Fragments, a `DialogFragment` is displayed in the context of a `FragmentTransaction`.

The `Show()` method on a `DialogFragment` takes a `FragmentTransaction` and a `string` as an input. The dialog will be added to the Activity, and the `FragmentTransaction` committed.

The following code demonstrates one possible way an Activity may use the `Show()` method to show a `DialogFragment`:

```
public void ShowDialog()
{
    var transaction = FragmentManager.BeginTransaction();
    var dialogFragment = new MyDialogFragment();
    dialogFragment.Show(transaction, "dialog_fragment");
}
```

Dismissing a Fragment

Calling `Dismiss()` on an instance of a `DialogFragment` causes a Fragment to be removed from the Activity and commits that transaction. The standard Fragment lifecycle methods that are involved with the destruction of a Fragment will be called.

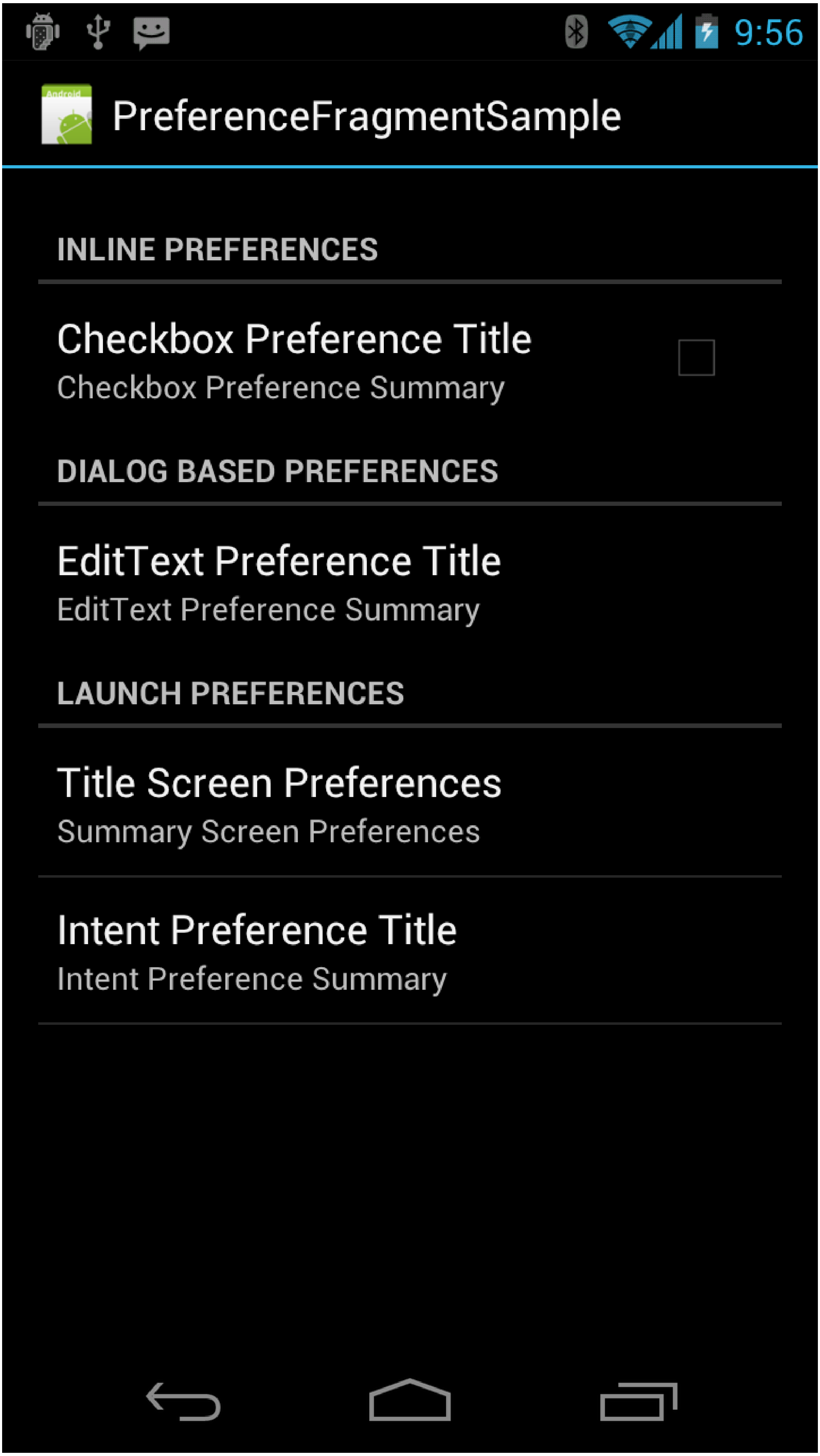
Alert Dialog

Instead of overriding `OnCreateView`, a `DialogFragment` may instead override `OnCreateDialog`. This allows an application to create an AlertDialog that is managed by a Fragment. The following code is an example that uses the `AlertDialog.Builder` to create a `Dialog`:

```
public class AlertDialogFragment : DialogFragment
{
    public override Dialog OnCreateDialog(Bundle savedInstanceState)
    {
        EventHandler<DialogClickEventArgs> okhandler;
        var builder = new AlertDialog.Builder(Activity)
            .SetMessage("This is my dialog.")
            .SetPositiveButton("Ok", (sender, args) =>
                                     {
                                         // Do something when this button is clicked.
                                     })
            .SetTitle("Custom Dialog");
        return builder.Create();
    }
}
```

PreferenceFragment

To help manage preferences, the Fragments API provides the `PreferenceFragment` subclass. The `PreferenceFragment` is similar to the PreferenceActivity – it will show a hierarchy of preferences to the user in a Fragment. As the user interacts with the preferences, they will be automatically saved to SharedPreferences. In Android 3.0 or higher applications, use the `PreferenceFragment` to deal with preferences in applications. The following picture shows an example of a `PreferenceFragment`:


9:56
PreferenceFragmentSample
INLINE PREFERENCES
Checkbox Preference Title
Checkbox Preference Summary
DIALOG BASED PREFERENCES
EditText Preference Title
EditText Preference Summary
LAUNCH PREFERENCES
Title Screen Preferences
Summary Screen Preferences
Intent Preference Title
Intent Preference Summary

Create A Preference Fragment from a Resource

The preference Fragment may be inflated from an XML resource file by using the PreferenceFragment.AddPreferencesFromResource method. A logical place to call this method in the lifecycle of the Fragment would be in the `OnCreate` method.

The `PreferenceFragment` pictured above was created by loading a resource from XML. The resource file is:

```
<?xml version="1.0" encoding="utf-8"?>

<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">

  <PreferenceCategory android:title="Inline Preferences">
    <CheckBoxPreference android:key="checkbox_preference"
                        android:title="Checkbox Preference Title"
                        android:summary="Checkbox Preference Summary" />

  </PreferenceCategory>

  <PreferenceCategory android:title="Dialog Based Preferences">

    <EditTextPreference android:key="edittext_preference"
                        android:title="EditText Preference Title"
                        android:summary="EditText Preference Summary"
                        android:dialogTitle="Edit Text Preferrence Dialog Title" />

  </PreferenceCategory>

  <PreferenceCategory android:title="Launch Preferences">

    <!-- This PreferenceScreen tag serves as a screen break (similar to page break
            in word processing). Like for other preference types, we assign a key
            here so it is able to save and restore its instance state. -->
    <PreferenceScreen android:key="screen_preference"
                      android:title="Title Screen Preferences"
                      android:summary="Summary Screen Preferences">

      <!-- You can place more preferences here that will be shown on the next screen. -->

      <CheckBoxPreference android:key="next_screen_checkbox_preference"
                          android:title="Next Screen Toggle Preference Title"
                          android:summary="Next Screen Toggle Preference Summary" />

    </PreferenceScreen>

    <PreferenceScreen android:title="Intent Preference Title"
                      android:summary="Intent Preference Summary">

      <intent android:action="android.intent.action.VIEW"
              android:data="http://www.android.com" />

    </PreferenceScreen>

  </PreferenceCategory>

</PreferenceScreen>
```

The code for the preference Fragment is as follows:

```
public class PrefFragment : PreferenceFragment
{
    public override void OnCreate(Bundle savedInstanceState)
    {
        base.OnCreate(savedInstanceState);
        AddPreferencesFromResource(Resource.Xml.preferences);
    }
}
```

Querying Activities to Create a Preference Fragment

Another technique for creating a `PreferenceFragment` involves querying Activities. Each Activity can use the METADATA_KEY_PREFERENCE attribute that will point to an XML resource file. In Xamarin.Android, this is done by adorning an Activity with the `MetaDataAttribute`, and then specifying the resource file to use. The `PreferenceFragment` class provides the method AddPreferenceFromIntent that can be used to query an Activity to find this XML resource and inflate a preference hierarchy for it.

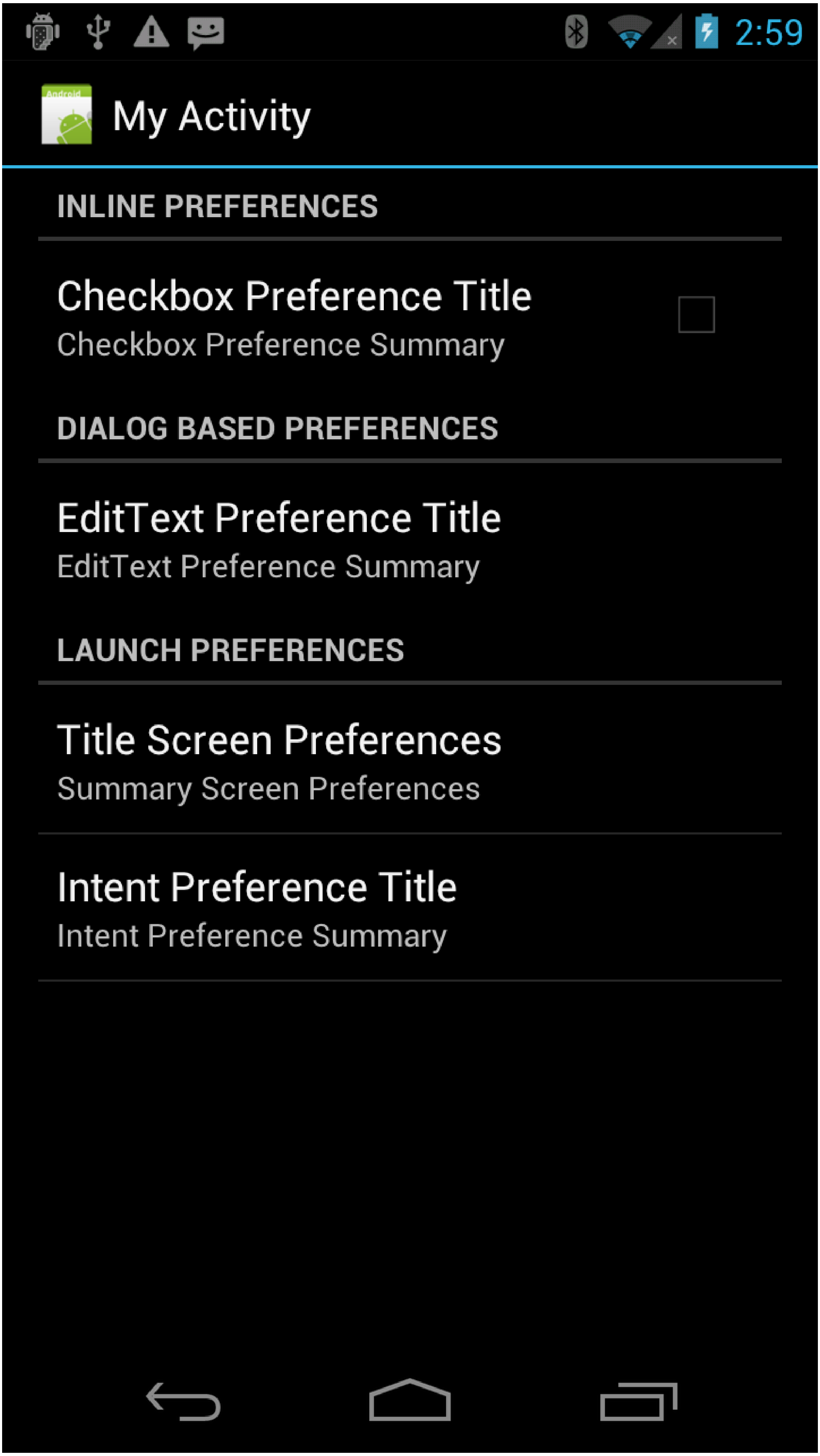
An example of this process is provided in the following code snippet, which uses `AddPreferencesFromIntent` to create a `PreferenceFragment`:

```
public class MyPreferenceFragment : PreferenceFragment
{
    public override void OnCreate(Bundle savedInstanceState)
    {
        base.OnCreate(savedInstanceState);
        var intent = new Intent(this.Activity, typeof (MyActivityWithPreferences));
        AddPreferencesFromIntent(intent);
    }
}
```

Android will look at the class `MyActivityWithPreference`. The class must be adorned with the `MetaDataAttribute,` as shown in the following code snippet:

```
[Activity(Label = "My Activity with Preferences")]
[MetaData(PreferenceManager.MetadataKeyPreferences, Resource = "@xml/preference_from_intent")]
public class MyActivityWithPreferences : Activity
{
    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);
        // This is deliberately blank
    }
}
```

The `MetaDataAttribute` declares an XML resource file that the `PreferenceFragment` will use to inflate the preference hierarchy. If the `MetatDataAttribute` is not provided, then an exception will be thrown at run time. When this code runs, the `PreferenceFragment` appears as in the following screenshot:


2:59
My Activity
INLINE PREFERENCES
Checkbox Preference Title
Checkbox Preference Summary
DIALOG BASED PREFERENCES
EditText Preference Title
EditText Preference Summary
LAUNCH PREFERENCES
Title Screen Preferences
Summary Screen Preferences
Intent Preference Title
Intent Preference Summary

Providing Backwards Compatibility with the Android Support Package

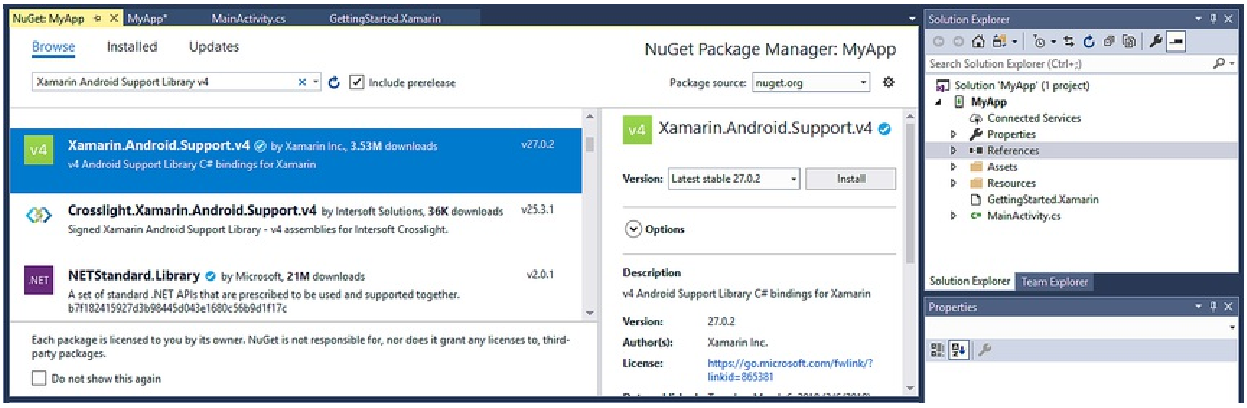
7/8/2021 • 2 minutes to read • Edit Online

The usefulness of Fragments would be limited without backwards compatibility with pre-Android 3.0 (API Level 11) devices. To provide this capability, Google introduced the Support Library (originally called the *Android Compatibility Library* when it was released) which backports some of the APIs from newer versions of Android to older versions of Android. It is the Android Support Package that enables devices running Android 1.6 (API level 4) to Android 2.3.3. (API level 10).

> **NOTE**
>
> Only the `ListFragment` and the `DialogFragment` are available via the Android Support Package. None of the other Fragment subclasses, such as the `PreferenceFragment,` are supported in the Android Support Package. They will not work in pre-Android 3.0 applications.

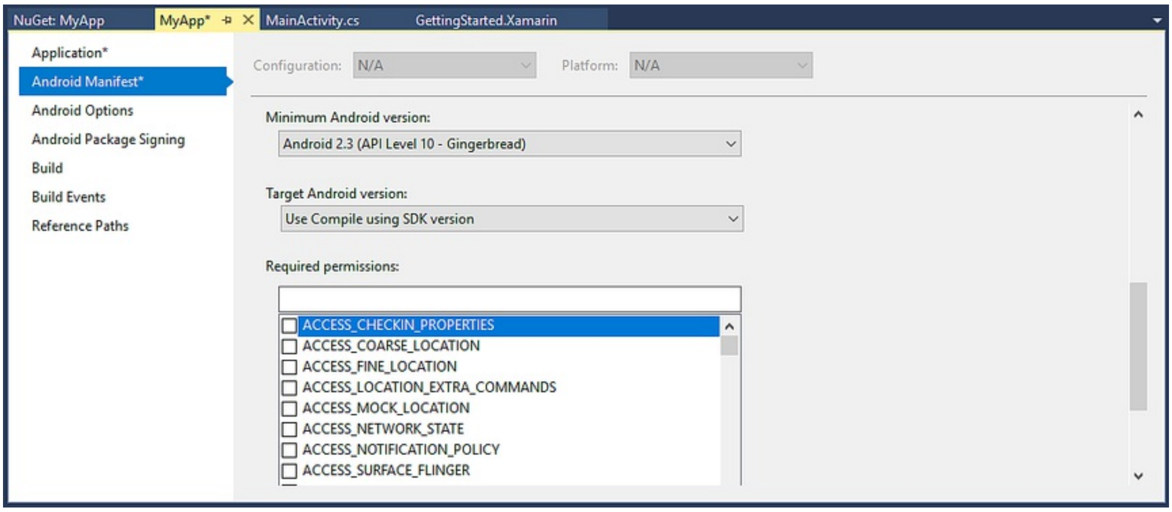
Adding the Support Package

The Android Support Package is not automatically added to a Xamarin.Android application. Xamarin provides the Android Support Library v4 NuGet package to simplify adding the support libraries to a Xamarin.Android application.To include the support packages into your Xamarin.Android application include the Android Support Library v4 component into your Xamarin.Android project, as illustrated in the following screenshot:



After these steps have been performed, it becomes possible to use Fragments in earlier versions of Android. The Fragment APIs will work the same now in these earlier versions, with the following exceptions:

- **Change the minimum Android Version** – The application no longer needs to target Android 3.0 or higher, as shown below:



- **Extend FragmentActivity** – The Activities that are hosting Fragments must now inherit from `Android.Support.V4.App.FragmentActivity` , and not from `Android.App.Activity` .
- **Update Namespaces** – Classes that inherit from `Android.App.Fragment` must now inherit from `Android.Support.V4.App.Fragment` . Remove the using statement " `using Android.App;` " at the top of the source code file and replace it with " `using Android.Support.V4.App` ".
- **Use SupportFragmentManager** – `Android.Support.V4.App.FragmentActivity` exposes a `SupportingFragmentManager` property that must be used to get a reference to the `FragmentManager` . For example:

```
FragmentTransaction fragmentTx = this.SupportingFragmentManager.BeginTransaction();
DetailsFragment detailsFrag = new DetailsFragment();
fragmentTx.Add(Resource.Id.fragment_container, detailsFrag);
fragmentTx.Commit();
```

With these changes in place, it will be possible to run a Fragment-based application on Android 1.6 or 2.x as well as on Honeycomb and Ice Cream Sandwich.

Related Links

- Android Support Library v4 NuGet

App-Linking in Android

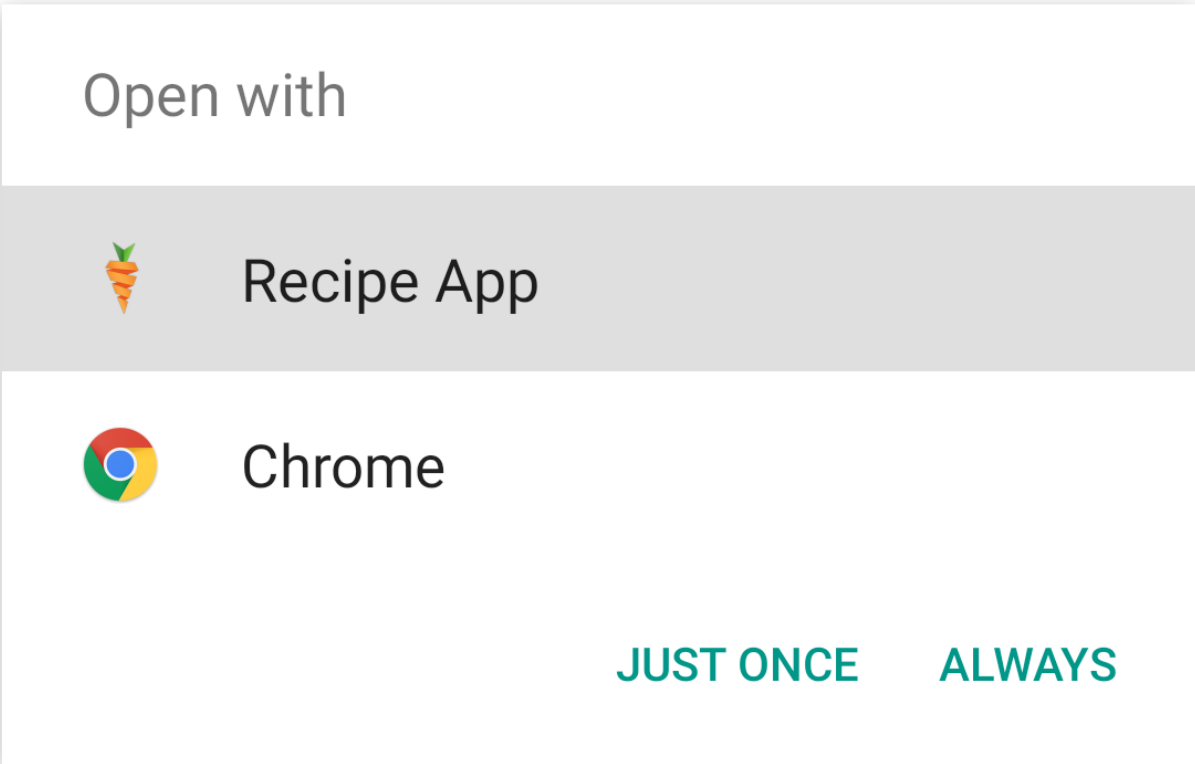
7/8/2021 • 7 minutes to read • Edit Online

This guide will discuss how Android 6.0 supports app-linking, a technique that allows mobile apps to respond to URLs on websites. It will discuss what app-linking is, how to implement app-linking in an Android 6.0 application, and how to configure a website to grant permissions to the mobile app for a domain.

App Linking Overview

Mobile applications no longer live in a silo – in many cases they are an important components of their businesses, along with their website. It's desirable for businesses to seamlessly connect their web presence and mobile applications, with links on a website launching mobile applications and displaying relevant content in the mobile app. *App-linking* (also referred to as *deep-linking*) is one technique that allows a mobile device to respond to a URI and launch a mobile application that corresponds to that URI.

Android handles app-linking through the *intent system* – when the user clicks on a link in a mobile browser, the mobile browser will dispatch an intent that Android will delegate to a registered application. For example, clicking on a link on a cooking website would open a mobile app that is associated with that website and display a specific recipe to the user. If there is more than one application registered to handle that intent, then Android will raise what is known as a *disambiguation dialog* that will ask a user what application to select the application that should handle the intent, for example:



Android 6.0 improves on this by using automatic link handling. It is possible for Android to automatically register an application as the default handler for a URI – the app will automatically launch and navigate directly to the relevant Activity. How Android 6.0 decides to handle a URI click depends on the following criteria:

1. **An existing app is already associated with the URI** – The user may have already associated an existing app with a URI. In that case, Android will continue to use that application.
2. **No existing app is associated with the URI, but a supporting app is installed** – In this scenario, the

user has not specified an existing app, so Android will use the installed supporting application to handle the request.

3. **No existing app is associated with the URI, but many supporting apps are installed** – Because there are multiple applications that support the URI, the disambiguation dialog will be displayed and the user must select which app will handle the URI.

If the user has no apps installed that support the URI, and one is subsequently installed, then Android will set that application as the default handler for the URI after verifying the association with the website that is associated with the URI.

This guide will discuss how to configure an Android 6.0 application and how to create and publish the Digital Asset Links file to support app-linking in Android 6.0.

Requirements

This guide requires Xamarin.Android 6.1 and an application that targets Android 6.0 (API level 23) or higher.

App-linking is possible in earlier versions of Android by using the Rivets NuGet package from the Xamarin Component store. The Rivets package is not compatible with app-linking in Android 6.0; it does not support Android 6.0 app linking.

Configuring App-Linking in Android 6.0

Setting up app-links in Android 6.0 involves two major steps:

1. **Adding one or more intent-filters for the website URI's** – the intent filters guide Android in how to handle a URL click in a mobile browser.
2. **Publishing a *Digital Asset Links JSON* file on the website** – this is a file that is uploaded to a website and is used by Android to verify the relationship between the mobile app and the domain of the website. Without this, Android cannot install the app as the default handle for URI's; the user must do so manually.

Configuring the Intent Filter

It is necessary to configure an intent filter that maps a URI (or possible a set of URIs) from a website to an Activity in an Android application. In Xamarin.Android, this relationship is established by adorning an Activity with the IntentFilterAttribute. The intent filter must declare the following information:

- `Intent.ActionView` – This will register the intent filter to respond to requests to view information
- `Categories` – The intent filter should register both **Intent.CategoryBrowsable** and **Intent.CategoryDefault** to be able to properly handle the web URI.
- `DataScheme` – The intent filter must declare `http` and/or `https`. These are the only two valid schemes.
- `DataHost` – This is the domain which the URIs will originate from.
- `DataPathPrefix` – This is an optional path to resources on the website.
- `AutoVerify` – The `autoVerify` attribute tells Android to verify the relationship between the application and the website. This will be discussed more below.

The following example shows how to use the IntentFilterAttribute to handle links from `https://www.recipe-app.com/recipes` and from `http://www.recipe-app.com/recipes`:

```
[IntentFilter(new [] { Intent.ActionView },
              Categories = new[] { Intent.CategoryBrowsable, Intent.CategoryDefault },
              DataScheme = "http",
              DataHost = "recipe-app.com",
              DataPathPrefix = "/recipe",
              AutoVerify=true)]
public class RecipeActivity : Activity
{
    // Code for the activity omitted
}
```

Android will verify every host that is identified by the intent filters against the Digital Assets File on the website before registering the application as the default handler for a URI. All the intent filters must pass verification before Android can establish the app as the default handler.

Creating the Digital Assets Link File

Android 6.0 app-linking requires that Android verify the association between the application and the website before setting the application as the default handler for the URI. This verification will occur when the application is first installed. The *Digital Assets Links* file is a JSON file that is hosted by the relevant webdomain(s).

> **NOTE**
>
> The `android:autoVerify` attribute must be set by the intent filter – otherwise Android will not perform the verification.

The file is placed by the webmaster of the domain at the location **https://domain/.well-known/assetlinks.json**.

The Digital Asset File contains the meta-data necessary for Android to verify the association. An **assetlinks.json** file has the following key-value pairs:

- `namespace` – the namespace of the Android application.
- `package_name` – the package name of the Android application (declared in the application manifest).
- `sha256_cert_fingerprints` – the SHA256 fingerprints of the signed application. Please see the guide Finding your Keystore's MD5 or SHA1 Signature for more information on how to obtain the SHA1 fingerprint of an application.

The following snippet is an example of **assetlinks.json** with a single application listed:

```
[
   {
      "relation":[
         "delegate_permission/common.handle_all_urls"
      ],
      "target":{
         "namespace":"android_app",
         "package_name":"com.example",
         "sha256_cert_fingerprints":[

"14:6D:E9:83:C5:73:06:50:D8:EE:B9:95:2F:34:FC:64:16:A0:83:42:E6:1D:BE:A8:8A:04:96:B2:3F:CF:44:E5"
         ]
      }
   }
]
```

It is possible to register more than one SHA256 fingerprint to support different versions or builds of your application. This next **assetlinks.json** file is an example of registering multiple applications:

```
[
   {
      "relation":[
         "delegate_permission/common.handle_all_urls"
      ],
      "target":{
         "namespace":"android_app",
         "package_name":"example.com.puppies.app",
         "sha256_cert_fingerprints":[

"14:6D:E9:83:C5:73:06:50:D8:EE:B9:95:2F:34:FC:64:16:A0:83:42:E6:1D:BE:A8:8A:04:96:B2:3F:CF:44:E5"
         ]
      }
   },
   {
      "relation":[
         "delegate_permission/common.handle_all_urls"
      ],
      "target":{
         "namespace":"android_app",
         "package_name":"example.com.monkeys.app",
         "sha256_cert_fingerprints":[

"14:6D:E9:83:C5:73:06:50:D8:EE:B9:95:2F:34:FC:64:16:A0:83:42:E6:1D:BE:A8:8A:04:96:B2:3F:CF:44:E5"
         ]
      }
   }
]
```

The Google Digital Asset Links website has an online tool that may assist with creating and testing the Digital Assets file.

Testing App-Links

After implementing app-links, the various pieces should be tested to ensure that they work as expected.

It is possible to confirm that the Digital Assets file is properly formatted and hosted by using Google's Digital Asset Links API, as shown in this example:

```
https://digitalassetlinks.googleapis.com/v1/statements:list?source.web.site=
  https://<WEB SITE ADDRESS>:&relation=delegate_permission/common.handle_all_urls
```

There are two tests that can be performed to ensure that the intent filters have been properly configured and that the app is set as the default handler for a URI:

1. The Digital Asset File is properly hosted as described above. The first test will dispatch an intent which Android should redirect to the mobile application. The Android application should launch and display the Activity registered for the URL. At a command prompt type:

   ```
   $ adb shell am start -a android.intent.action.VIEW \
       -c android.intent.category.BROWSABLE \
       -d "http://<domain1>/recipe/scalloped-potato"
   ```

2. Display the existing link handling policies for the applications installed on a given device. The following command will dump a listing of link policies for each user on the device with the following information. At the command prompt, type the following command:

   ```
   $ adb shell dumpsys package domain-preferred-apps
   ```

- `Package` – The package name of the application.
- `Domain` – The domains (separated by spaces) whose web links will be handled by the application
- `Status` – This is the current link-handling status for the app. A value of **always** means that the application has `android:autoVerify=true` declared and has passed system verification. It is followed by a hexadecimal number representing the Android system's record of the preference.

For example:

```
$ adb shell dumpsys package domain-preferred-apps

App linkages for user 0:
Package: com.android.vending
Domains: play.google.com market.android.com
Status: always : 200000002
```

Summary

This guide discussed how app-linking works in Android 6.0. It then covered how to configure an Android 6.0 application to support and respond to app links. It also discussed how to test app-linking in an Android application.

Related Links

- Finding your Keystore's MD5 or SHA1 Signature
- AppLinks
- Google Digital Assets Links
- Statement List Generator and Tester

AndroidX with Xamarin

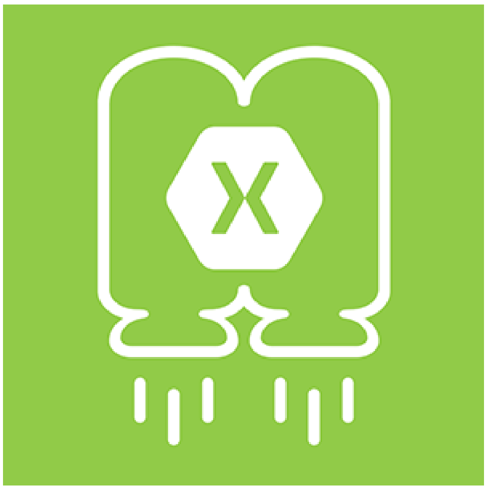
7/8/2021 • 3 minutes to read • Edit Online

How to get started developing apps with AndroidX using Xamarin.Android.

AndroidX is a major improvement to the original Android Support Library, which is no longer maintained. **AndroidX** packages fully replace the Android Support Library by providing feature parity and new libraries you can use in your Android applications.

AndroidX includes the following features:

- All packages inside AndroidX now have a consistent namespace starting with `androidx`. This means all Android Support Library packages map to a corresponding `androidx.*` package.
- `androidx` packages are separately maintained and updated. This means that you can update AndroidX libraries independently of each other.
- As of v28 of the Android Support Library, there will be no more releases. All development will be included in `androidx` instead.



Requirements

The following list is required to use AndroidX features in Xamarin-based apps:

- **Visual Studio** - On Windows update to Visual Studio 2019 version 16.4 or later. On macOS, update to Visual Studio 2019 for Mac version 8.4 or later.
- **Xamarin.Android** - Xamarin.Android 10.0 or later must be installed with Visual Studio (Xamarin.Android is automatically installed as part of the **Mobile Development With .NET** workload on Windows and installed as part of the **Visual Studio for Mac Installer**)
- **Java Developer Kit** - Xamarin.Android 10.0 development requires JDK 8. Microsoft's distribution of the OpenJDK is automatically installed as part of Visual Studio.
- **Android SDK** - Android SDK API 28 or higher must be installed via the Android SDK Manager.

Get started

You can get started with AndroidX by including any AndroidX NuGet package inside of your Android project. Learn more about installing and using a package in Visual Studio or Visual Studio for Mac

Behavior changes

Because AndroidX is a redesign of the Android Support Library, it includes migration steps that will affect Android applications built with the Android Support Library.

Package Name Change

The package names have been changed between the old and new packages. Below you can see an example of these changes:

OLD	NEW
android.support.**	androidx.@
android.design.**	com.google.android.material.@
android.support.test.**	androidx.test.@
android.arch.**	androidx.@
android.arch.persistence.room.**	androidx.room.@
android.arch.persistence.**	androidx.sqlite.@

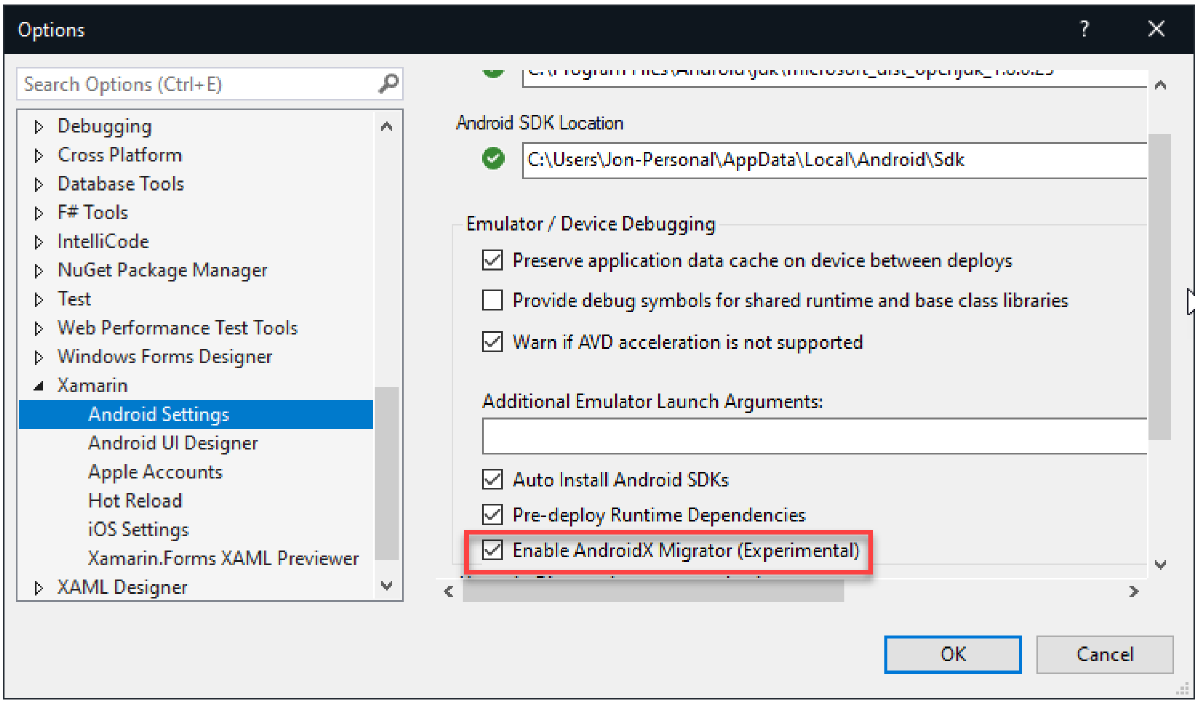
For more details on package naming, see the following documentation.

Migration Tooling

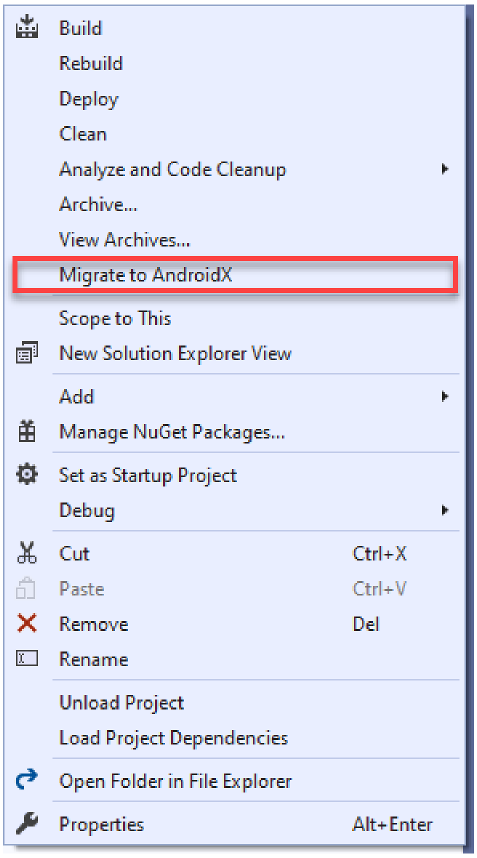
There are three migration steps that you'll want to be aware of for your application.

1. If your application includes **Android Support Library namespaces and you'd like to migrate them to AndroidX namespaces**, you can use our **Migrate to AndroidX** IDE tooling to take care of most namespace scenarios.

Enable the **AndroidX Migrator** via **Tools > Options > Xamarin > Android Settings** inside Visual Studio 2019 (you can skip this step on Visual Studio for Mac).



Right-click your project and **Migrate to AndroidX**.



NOTE

You will need to make some manual namespace changes for scenarios the tool doesn't cover. While we will map the correct package for you, it is encouraged that you to take a look at the official artifact mappings and class mappings to help your project migration.

2. If your application includes **any dependencies that have not been migrated to the AndroidX namespace**, you'll have to use the Android Support Library to AndroidX Migration package.
3. If your application **does not include any dependencies that require AndroidX namespace migration**, you can use the AndroidX libraries on NuGet today.

Troubleshooting

- Certain architecture packages within AndroidX will conflict with the Support Library versions. To fix this, you should use the AndroidX version of these packages and remove the Support Library version. For example, if you are referencing `Xamarin.Android.Arch.Work.Runtime` in your project, it will conflict with the types of the newly added `AndroidX.Work` package.

Summary

This article introduced AndroidX and explained how to install and configure the latest tools and packages for Xamarin.Android development with AndroidX. It provided an overview of what AndroidX is. It included links to API documentation and Android Developer topics to help you get started in creating apps using AndroidX. It also highlighted the most important AndroidX behavior changes and troubleshooting topics that could impact existing apps.

Related links

- Introduction to AndroidX | The Xamarin Show
- AndroidX
- Xamarin AndroidX GitHub Repository
- Xamarin AndroidX Migration GitHub Repository

Android 10 with Xamarin

7/8/2021 • 6 minutes to read • Edit Online

How to get started developing apps for Android 10 using Xamarin.Android.

Android 10 is now available from Google. A number of new features and APIs are being made available in this release, and many of them are necessary to take advantage of new hardware capabilities in the latest Android devices.

This article is structured to help you get started in developing Xamarin.Android apps for Android 10. It explains how to install the necessary updates, configure the SDK, and prepare an emulator or device for testing. It also provides an outline of the new features in Android 10 and provides example source code that illustrates how to use some of the key Android 10 features.

Xamarin.Android 10.0 provides support for Android 10. For more information about Xamarin.Android support for Android 10, see the Xamarin.Android 10.0 release notes.

Requirements

The following list is required to use Android 10 features in Xamarin-based apps:

- **Visual Studio** - Visual Studio 2019 is recommended. On Windows update to Visual Studio 2019 version 16.3 or later. On macOS, update to Visual Studio 2019 for Mac version 8.3 or later.
- **Xamarin.Android** - Xamarin.Android 10.0 or later must be installed with Visual Studio (Xamarin.Android is automatically installed as part of the **Mobile Development With .NET** workload on Windows and installed as part of the **Visual Studio for Mac Installer**)
- **Java Developer Kit** - Xamarin.Android 10.0 development requires JDK 8. Microsoft's distribution of the OpenJDK is automatically installed as part of Visual Studio.
- **Android SDK** - Android SDK API 29 must be installed via the Android SDK Manager.

Get started

To get started developing Android 10 apps with Xamarin.Android, you must download and install the latest tools and SDK packages before you can create your first Android 10 project:

1. **Visual Studio 2019 is recommended**. Update to Visual Studio 2019 version 16.3 or later. If you are using Visual Studio for Mac 2019, update to Visual Studio 2019 for Mac version 8.3 or later.
2. Install **Android 10 (API 29)** packages and tools via the SDK Manager.
   - Android 10 (API 29) SDK Platform
   - Android 10 (API 29) System Image
   - Android SDK Build-Tools 29.0.0+
   - Android SDK Platform-Tools 29.0.0+
   - Android Emulator 29.0.0+
3. Create a new Xamarin.Android project that targets Android 10.0.

4. Configure an emulator or device for testing Android 10 apps.

Each of these steps is explained below:

Update Visual Studio

Visual Studio 2019 is recommended for building Android 10 apps using Xamarin.

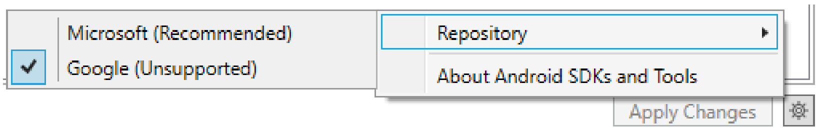
If you are using Visual Studio 2019, update to Visual Studio 2019 version 16.3 or later (for instructions, see Update Visual Studio 2019 to the most recent release). On macOS, update to Visual Studio 2019 for Mac 8.3 or later (for instructions, see Update Visual Studio 2019 for Mac to the most recent release).

Install the Android SDK

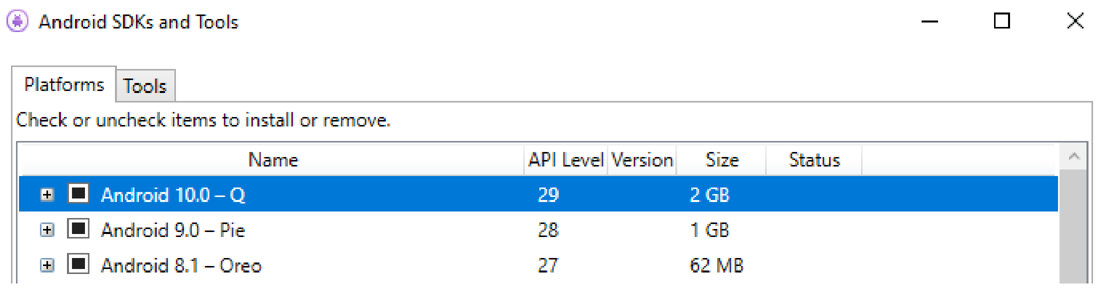
To create a project with Xamarin.Android 10.0, you must first use the Android SDK Manager to install the SDK platform for **Android 10 (API level 29)**.

1. Start the SDK Manager. In Visual Studio, click **Tools > Android > Android SDK Manager**. In Visual Studio for Mac, click **Tools > SDK Manager**.

2. In the lower right-hand corner, click the gear icon and select **Repository > Google (Unsupported)**:



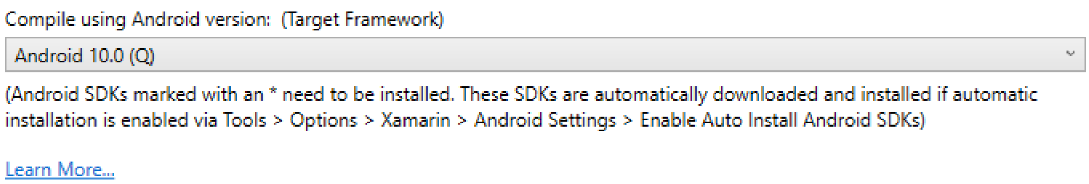
3. Install the **Android 10 SDK Platform** packages, which are listed as **Android SDK Platform 29** in the **Platforms** tab (for more information about using the SDK Manager, see Android SDK setup):



Create a Xamarin.Android project

Create a new Xamarin.Android project. If you are new to Android development with Xamarin, see Hello, Android to learn about creating Xamarin.Android projects.

When you create an Android project, you must configure the version settings to target Android 10.0 or later. For example, to target your project for Android 10, you must configure the target Android API level of your project to **Android 10.0 (API 29)**. This includes both your **Target Framework Version** and **Target Android SDK Version** to API 29 or later. For more information about configuring Android API levels, see Understanding Android API Levels.



Configure a device or emulator

If you are using a physical device such as a Pixel, you can download the Android 10 update by going to the System > System update > Check for update in your phone's settings. If you'd prefer to flash your device, please see the instructions on flashing a Factory Image or OTA Image to your device.

If you are using an emulator, create a virtual device for API level 29 and select an x86-based image. For information about using the Android Device Manager to create and manage virtual devices, see Managing Virtual Devices with the Android Device Manager. For information about using the Android Emulator for testing and debugging, see Debugging on the Android Emulator.

New features

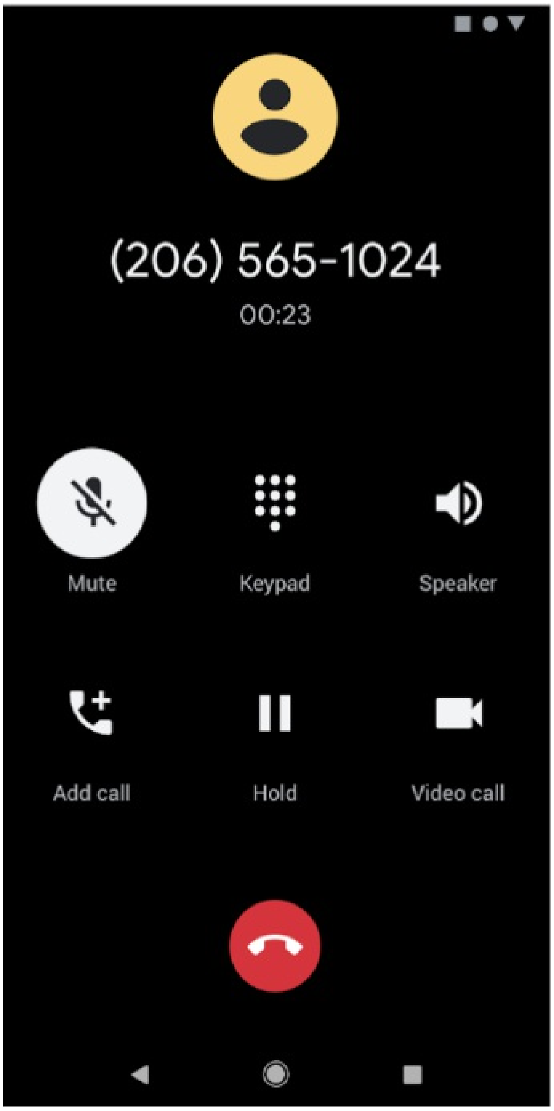
Android 10 introduces a variety of new features. Some of these new features are intended to leverage new hardware capabilities offered by the latest Android devices, while others are designed to further enhance the Android user experience:

Enhance your app with Android 10 features and APIs

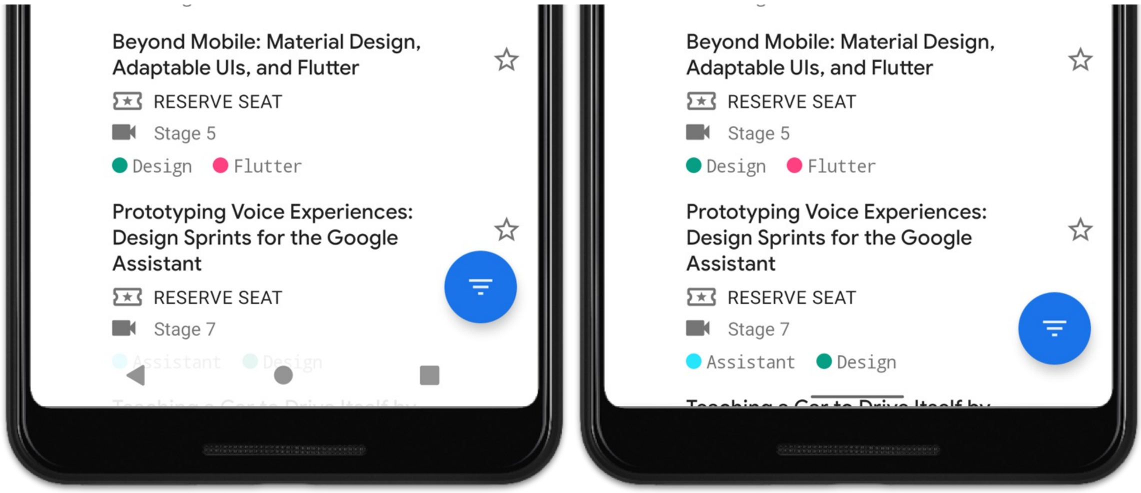
Next, when you're ready, dive into Android 10 and learn about the new features and APIs that you can use. Here are some of the top features to get started with.

These features are recommend for every app:

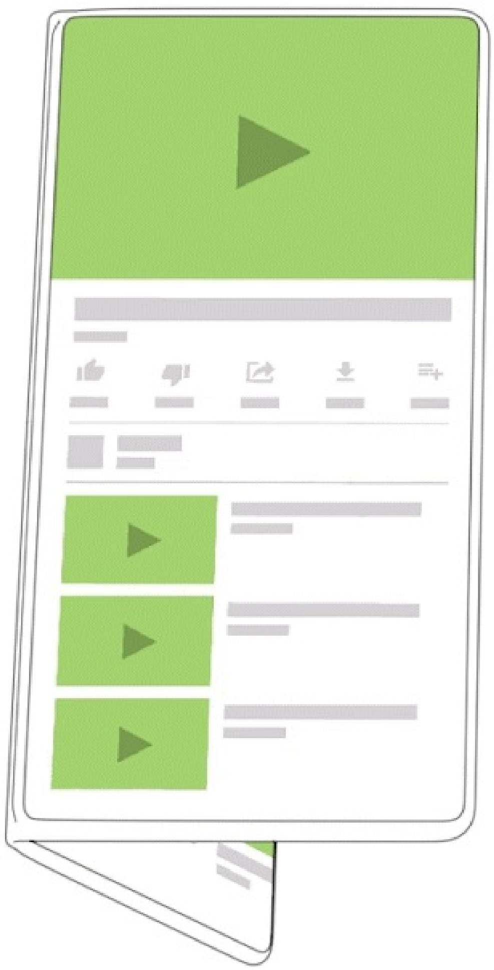
- **Dark Theme:** Ensure a consistent experience for users who enable system-wide dark theme by adding a Dark Theme or enabling Force Dark.



- **Support gestural navigation** in your app by going edge-to-edge and making sure your custom gestures are complementary to the system navigation gestures.



- **Optimize for foldables:** Deliver seamless, edge-to-edge experiences on today's innovative devices by optimizing for foldables.



These features are recommended if relevant for your app:

- **More interactive notifications:** If your notifications include messages, enable suggested replies and actions in notifications to engage users and let them take action instantly.
- **Better biometrics:** If you use biometric auth, move to BiometricPrompt, the preferred way to support fingerprint auth on modern devices.
- **Enriched recording:** To support captioning or gameplay recording, enable audio playback capture. It's a great way to reach more users and make your app more accessible.
- **Better codecs:** For media apps, try AV1 for video streaming and HDR10+ for high dynamic range video. For

speech and music streaming, you can use Opus encoding, and for musicians, a native MIDI API is available.

- **Better networking APIs:** If your app manages IoT devices over Wi-Fi, try the new network connection APIs for functions like configuring, downloading, or printing.

These are just a few of the many new features and APIs in Android 10. To see them all, visit the Android 10 site for developers.

Behavior changes

When the Target Android Version is set to API level 29, there are several platform changes that cann affect your app's behavior even if you are not implementing the new features described above. The following list is a brief summary of these changes:

- To ensure app stability and compatibility, the Android platform now restricts non-SDK interfaces your app can use in Android 10.
- Shared memory has changed.
- Android runtime & AOT correctness.
- Permissions for fullscreen intents must request `USE_FULL_SCREEN_INTENT`.
- Support for foldables.

Summary

This article introduced Android 10 and explained how to install and configure the latest tools and packages for Xamarin.Android development with Android 10. It provided an overview of the key features available in Android 10. It included links to API documentation and Android Developer topics to help you get started in creating apps for Android 10. It also highlighted the most important Android 10 behavior changes that could impact existing apps.

Related links

- Android 10

Android Pie features

7/8/2021 • 10 minutes to read • Edit Online

How to get started developing apps for Android 9 Pie using Xamarin.Android.

Android 9 Pie is now available from Google. A number of new features and APIs are being made available in this release, and many of them are necessary to take advantage of new hardware capabilities in the latest Android devices.

This article is structured to help you get started in developing Xamarin.Android apps for Android Pie. It explains how to install the necessary updates, configure the SDK, and prepare an emulator or device for testing. It also provides an outline of the new features in Android Pie and provides example source code that illustrates how to use some of the key Android Pie features.

Xamarin.Android 9.0 provides support for Android Pie. For more information about Xamarin.Android support for Android Pie, see the Android P Developer Preview 3 release notes.

Requirements

The following list is required to use Android Pie features in Xamarin-based apps:

- **Visual Studio** – Visual Studio 2019 is recommended. If you are using Visual Studio 2017, on Windows update to Visual Studio 2017 version 15.8 or later. On macOS, update to Visual Studio 2017 for Mac version 7.6 or later.
- **Xamarin.Android** – Xamarin.Android 9.0.0.17 or later must be installed with Visual Studio (Xamarin.Android is automatically installed as part of the **Mobile development with .NET** workload).
- **Java Developer Kit** – Xamarin Android 9.0 development requires JDK 8 (or you can try the preview of Microsoft's distribution of the OpenJDK). JDK8 is automatically installed as part of the **Mobile development with .NET** workload.
- **Android SDK** – Android SDK API 28 or later must be installed via the Android SDK Manager.

Getting started

To get started developing Android Pie apps with Xamarin.Android, you must download and install the latest tools and SDK packages before you can create your first Android Pie project:

1. Visual Studio 2019 is recommended. If you are using Visual Studio 2017, update to Visual Studio 2017 version 15.8 or later. If you are using Visual Studio for Mac, update to Visual Studio 2017 for Mac version 7.6 or later.
2. Install **Android Pie (API 28)** packages and tools via the SDK Manager.
3. Create a new Xamarin.Android project that targets **Android 9.0**.
4. Configure an emulator or device for testing Android Pie apps.

Each of these steps is explained in the following sections:

Update Visual Studio

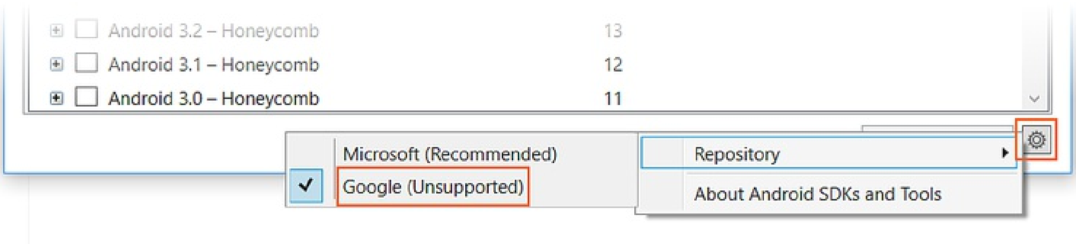
Visual Studio 2019 is recommended for building Android Pie apps using Xamarin.

If you are using Visual Studio 2017, update to Visual Studio 2017 version 15.8 or later (for instructions, see Update Visual Studio 2017 to the most recent release). On macOS, update to Visual Studio 2017 for Mac 7.6 or later (for instructions, see Setup and Install Visual Studio for Mac).

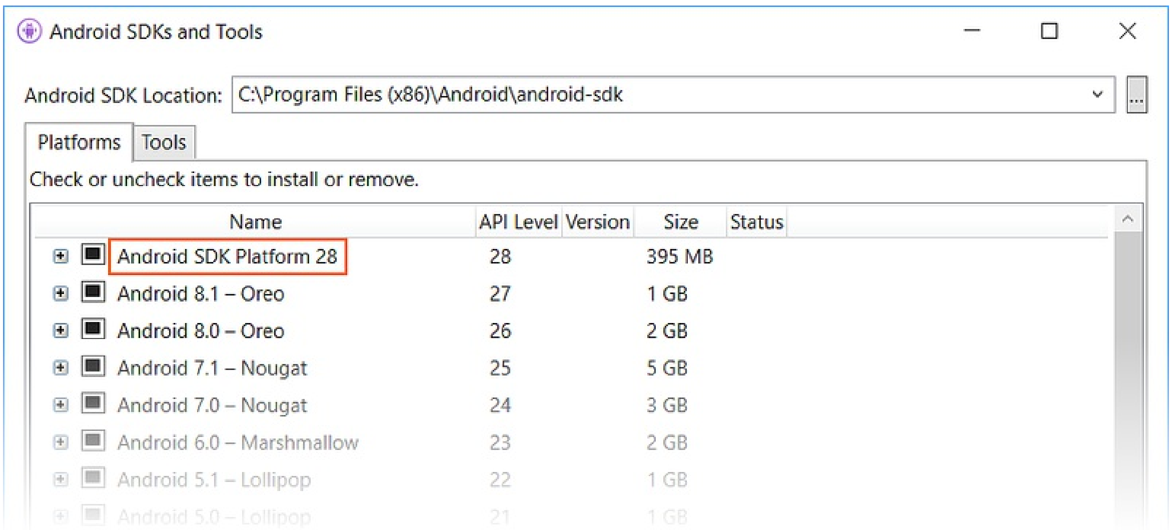
Install the Android SDK

To create a project with Xamarin.Android 9.0, you must first use the Android SDK Manager to install the SDK platform for **Android Pie (API level 28)** or later.

1. Start the SDK Manager. In Visual Studio, click **Tools > Android > Android SDK Manager**. In Visual Studio for Mac, click **Tools > SDK Manager**.
2. In the lower right-hand corner, click the gear icon and select **Repository > Google (Unsupported)**:



3. Install the **Android Pie** SDK packages, which are listed as **Android SDK Platform 28** in the **Platforms** tab (for more information about using the SDK Manager, see Android SDK Setup):



4. If you are using an emulator, create a virtual device that supports **API Level 28**. For more information about creating virtual devices, see Managing Virtual Devices with the Android Device Manager.

Start a Xamarin.Android project

Create a new Xamarin.Android project. If you are new to Android development with Xamarin, see Hello, Android to learn about creating Xamarin.Android projects.

When you create an Android project, you must configure the version settings to target Android 9.0 or later. For example, to target your project for Android Pie, you must configure the target Android API level of your project to **Android 9.0** (API 28). It is recommended that you also set your Target Framework level to API 28 or later. For more about configuring Android API levels, see Understanding Android API Levels.

Configure a device or emulator

If you are using a physical device such as a Nexus or a Pixel, you can update your device to Android Pie by following the instructions in Factory Images for Nexus and Pixel Devices.

If you are using an emulator, create a virtual device for API level 28 and select an x86-based image. For information about using the Android Device Manager to create and manage virtual devices, see Managing Virtual Devices with the Android Device Manager. For information about using the Android emulator for testing and debugging, see Debugging on the Android Emulator.

New features

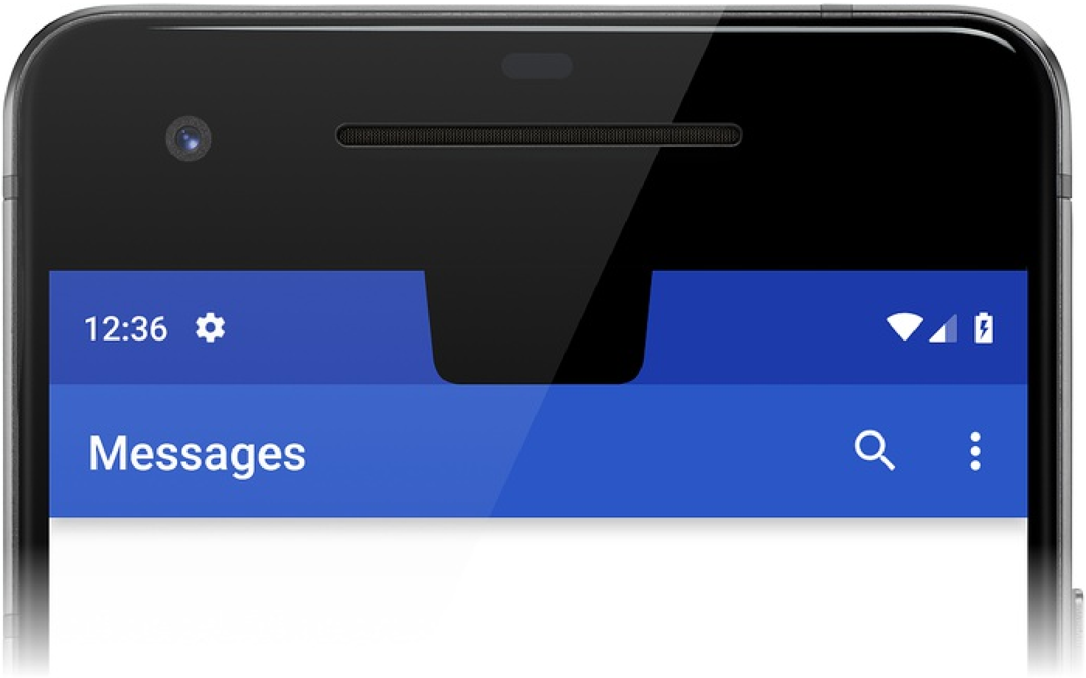
Android Pie introduces a variety of new features. Some of these new features are intended to leverage new hardware capabilities offered by the latest Android devices, while others are designed to further enhance the Android user experience:

- **Display Cutout Support** – Provides APIs to find the location and shape of the *cutout* at the top of the screen on newer Android devices.
- **Notification Enhancements** – Notification messages can now display images, and a new `Person` class is used to simplify conversation participants.
- **Indoor Positioning** – Platform support for the WiFi Round-Trip-Time protocol, which makes it possible for apps to use WiFi devices for navigation in indoor settings.
- **Multi-Camera Support** – Offers the capability to access streams simultaneously from multiple physical cameras (such as dual-front and dual-back cameras).

The following sections highlight these features and provide brief code examples to help you get started using them in your app.

Display cutout support

Many newer Android devices with edge-to-edge screens have a *Display Cutout* (or "notch") at the top of the display for camera and speaker. The following screenshot provides an emulator example of a cutout:



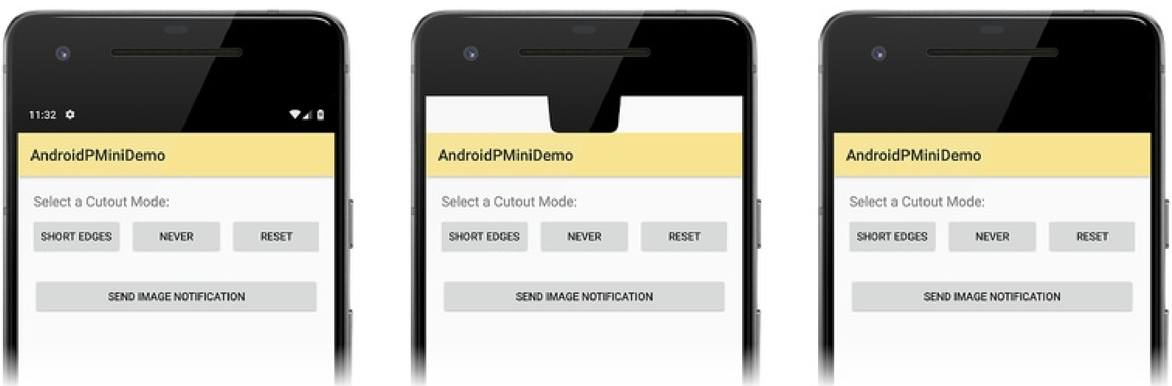
To manage how your app window displays its content on devices with a display cutout, Android Pie has added a new [LayoutInDisplayCutoutMode](https://developer.xamarin.com/api/property/Android.Views.WindowManagerLayoutParams.LayoutInDisplayCutoutMode/) window layout attribute. This attribute can be set to one of the following values:

- LayoutInDisplayCutoutModeNever – The window is never allowed to overlap with the cutout area.
- LayoutInDisplayCutoutModeShortEdges – The window is allowed to extend into the cutout area but only on the short edges of the screen.
- LayoutInDisplayCutoutModeDefault – The window is allowed to extend into the cutout area if the cutout is contained within a system bar.

For example, to prevent the app window from overlapping with the cutout area, set the layout cutout mode to *never*:

```
Window.Attributes.LayoutInDisplayCutoutMode =
    Android.Views.LayoutInDisplayCutoutMode.Never;
```

The following examples provide examples of these cutout modes. The first screenshot on the left is of the app in non-fullscreen mode. In the center screenshot, the app goes full-screen with `LayoutInDisplayCutoutMode` set to `LayoutInDisplayCutoutModeShortEdges`. Notice that the app's white background extends into the display cutout area:



Non Full-screen Cutout Short Edges Cutout Never

In the final screenshot (above on the right), `LayoutInDisplayCutoutMode` is set to `LayoutInDisplayCutoutModeShortNever` before it goes to full-screen. Notice that the app's white background is not allowed to extend into the display cutout area.

If you need more detailed information about the cutout area on the device, you can use the new DisplayCutout class. `DisplayCutout` represents the area of the display that cannot be used to display content. You can use this information to retrieve the location and shape of the cutout so that your app does not attempt to display content in this non-functional area.

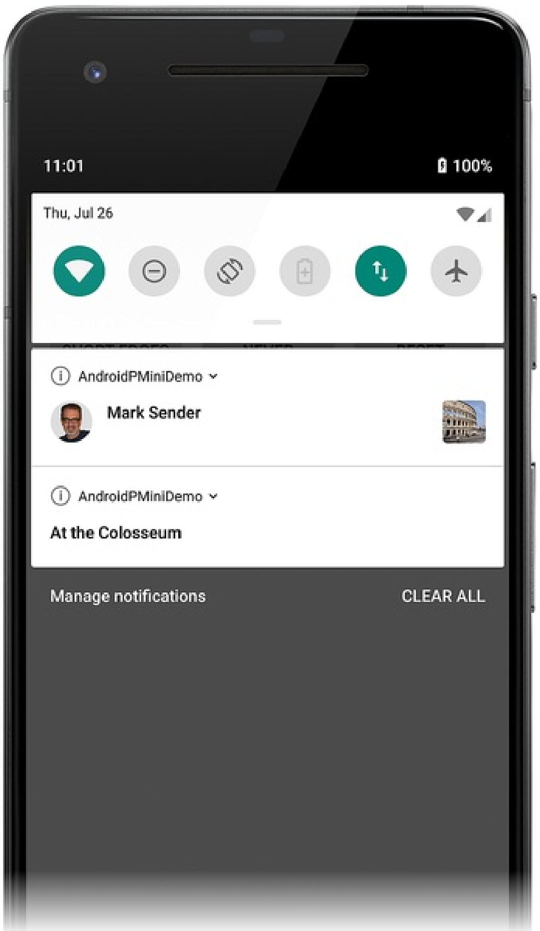
For more information about the new cutout features in Android P, see Display cutout support.

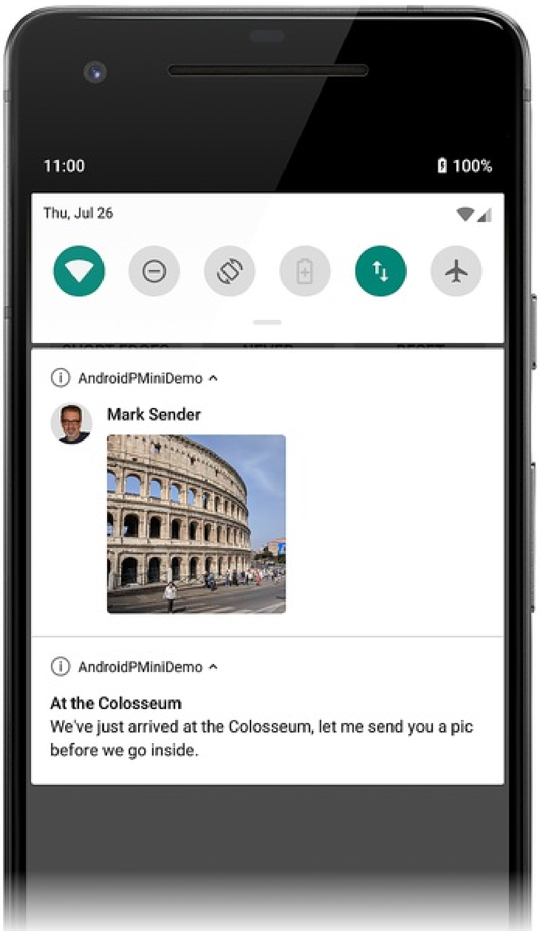
Notifications enhancements

Android Pie introduces the following enhancements to improve the messaging experience:

- Notification channels (introduced in Android Oreo) now supports blocking of channel groups.
- The notification system has three new Do-Not-Disturb categories (prioritizing alarms, system sounds, and media sources). In addition, there are seven new Do-Not-Disturb modes that can be used to suppress visual interruptions (such as badges, notification lights, status bar appearances, and launching of full-screen activities).
- A new Person class has been added to represent the sender of a message. Use of this class helps to optimize the rendering of each notification by identifying people involved in a conversation (including their avatars and URIs).
- Notifications can now display images.

The following example illustrates how to use the new APIs to generate a notification that contains an image. In the following screenshots, a text notification is posted and is followed by a notification with an embedded image. When the notifications are expanded (as seen on the right), the text of the first notification is displayed and the image embedded in the second notification is enlarged:





The following example illustrates how to include an image in an Android Pie notification, and it demonstrates usage of the new `Person` class:

1. Create a `Person` object that represents the sender. For example, the sender's name and icon are included in `fromPerson`:

```
Icon senderIcon = Icon.CreateWithResource(this, Resource.Drawable.sender_icon);
Person fromPerson = new Person.Builder()
    .SetIcon(senderIcon)
    .SetName("Mark Sender")
    .Build();
```

2. Create a `Notification.MessagingStyle.Message` that contains the image to send, passing the image to the new Notification.MessagingStyle.Message.SetData method. For example:

```
Uri imageUri = Uri.Parse("android.resource://com.xamarin.pminidemo/drawable/example_image");
Notification.MessagingStyle.Message message = new Notification.MessagingStyle
        .Message("Here's a picture of where I'm currently standing", 0, fromPerson)
        .SetData("image/", imageUri);
```

3. Add the message to a `Notification.MessagingStyle` object. For example:

```
Notification.MessagingStyle style = new Notification.MessagingStyle(fromPerson)
        .AddMessage(message);
```

4. Plug this style into the notification builder. For example:

```
builder = new Notification.Builder(this, MY_CHANNEL)
    .SetContentTitle("Tour of the Colosseum")
    .SetContentText("I'm standing right here!")
    .SetSmallIcon(Resource.Mipmap.ic_notification)
    .SetStyle(style)
    .SetChannelId(MY_CHANNEL);
```

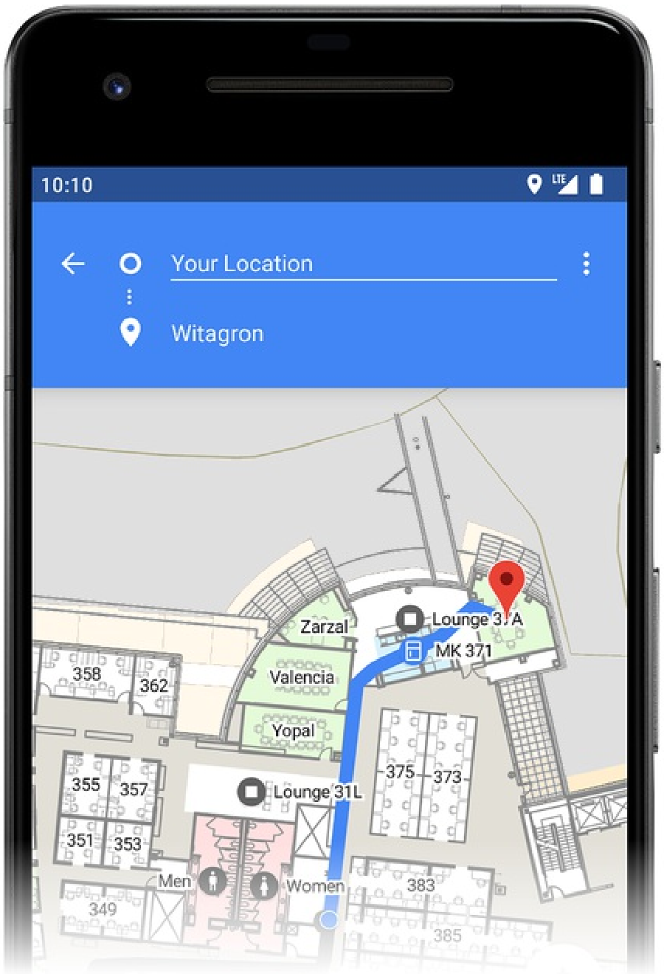
5. Publish the notification. For example:

```
const int notificationId = 1000;
notificationManager.Notify(notificationId, builder.Build());
```

For more information about creating notifications, see Local Notifications.

Indoor positioning

Android Pie provides support for IEEE 802.11mc (also known as *WiFi Round-Trip-Time* or *WiFi RTT*), which makes it possible for apps to detect the distance to one or more Wi-Fi access points. Using this information, it is possible for your app to take advantage of *indoor positioning* with an accuracy of one to two meters. On Android devices that provide hardware support for IEEE 801.11mc, your app can offer navigation features such as location-based control of smart appliances or turn-by-turn instructions through a store:



The new WifiRttManager class and several helper classes provides the means for measuring distance to Wi-Fi devices. For more information about the indoor positioning APIs introduced in Android P, see Android.Net.Wifi.Rtt.

Multi-Camera support

Many newer Android devices have dual-front and/or dual-back cameras that are useful for such features as stereo vision, enhanced visual effects, and improved zoom capability. Android P introduces a new Multi-Camera API that makes it possible for your app to use a *logical camera* (or *logical multi-camera*) that is backed by two or more physical cameras. To determine if the device supports a logical multi camera, you can look at the capabilities of each camera on the device to see if it supports RequestAvailableCapabilitiesLogicalMultiCamera.

Android Pie also includes a new SessionConfiguration class that can be used to help reduce delays during initial capture and eliminate the need to start and start the camera stream.

For more information about Multi-Camera support in Android P, see Multi-camera support and camera updates.

Other features

In addition, Android Pie supports several other new features:

- The new AnimatedImageDrawable class, which can be used for drawing and displaying animated images.
- A new ImageDecoder class that replaces `BitmapFactory`. `ImageDecoder` can be used to decode an `AnimatedImageDrawable`.
- Support for HDR (High Dynamic Range) video and HEIF (High Efficiency Image File Format) images.
- The JobScheduler has been enhanced to more intelligently handle network-related jobs. The new GetNetwork method of the JobParameters class returns the best network for performing any network requests for a given job.

For more information about the latest Android Pie features, see Android 9 features and APIs.

Behavior changes

When the Target Android Version is set to API level 28, there are several platform changes that can affect your app's behavior even if you are not implementing the new features described above. The following list is a brief summary of these changes:

- Apps must now request foreground permission before using foreground services.
- If your app has more than one process, it cannot share a single WebView data directory across processes.
- Directly accessing another app's data directory by path is no longer allowed.

For more information about behavior changes for apps targeting Android P, see Behavior Changes.

Sample code

AndroidPMiniDemo is a Xamarin.Android sample app for Android Pie that demonstrates how to set display cutout modes, how to use the new `Person` class, and how to send a notification that includes an image.

Summary

This article introduced Android Pie and explained how to install and configure the latest tools and packages for Xamarin.Android development with Android Pie. It provided an overview of the key features available in Android Pie, with example source code for several of these features. It included links to API documentation and Android Developer topics to help you get started in creating apps for Android Pie. It also highlighted the most important Android Pie behavior changes that could impact existing apps.

Related links

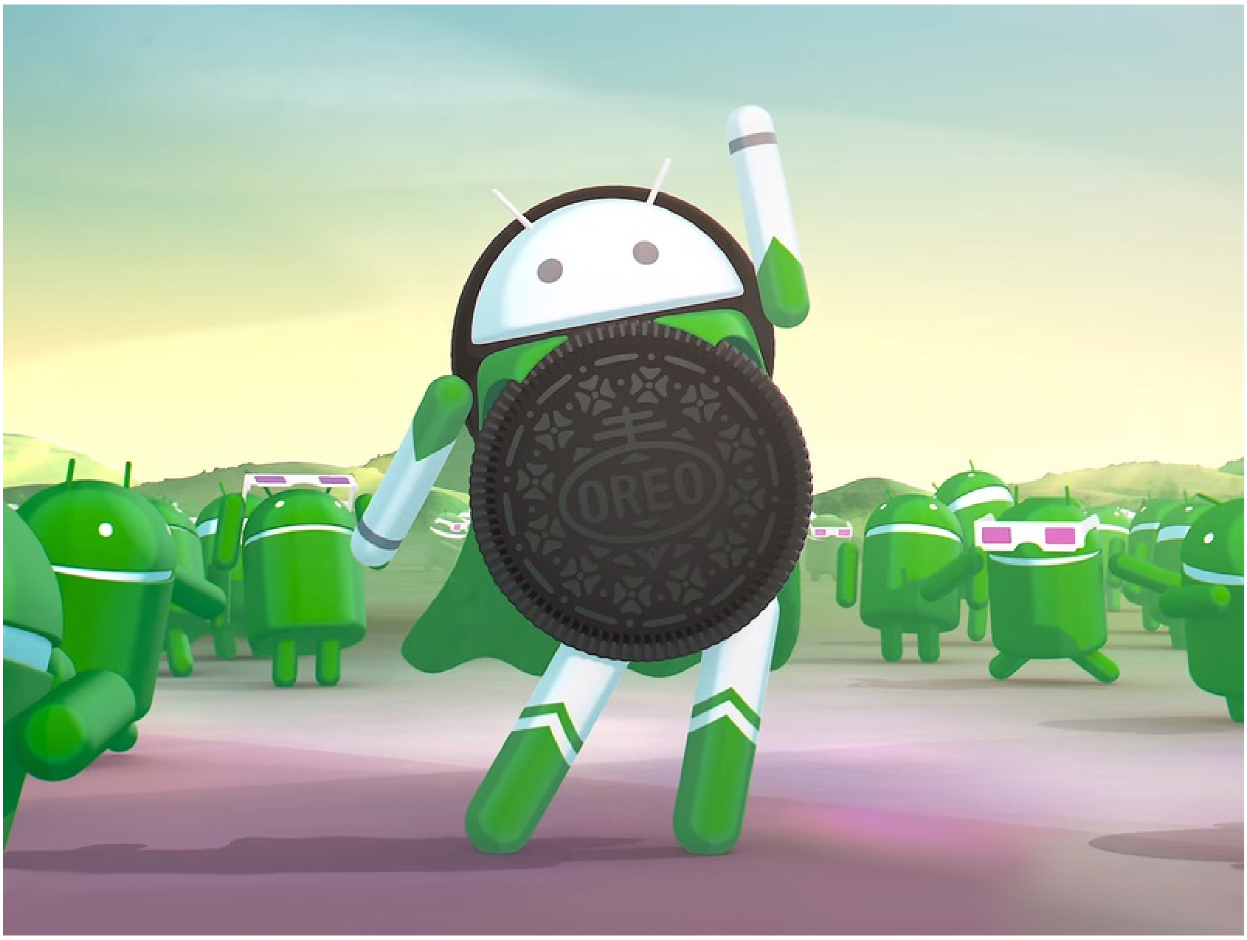
- Android 9 Pie

Oreo Features

7/8/2021 • 14 minutes to read • Edit Online

How to get started using Xamarin.Android to develop apps for the latest version of Android.

Android 8.0 Oreo is the latest version of Android available from Google. Android Oreo offers many new features of interest to Xamarin.Android developers. These features include notification channels, notification badges, custom fonts in XML, downloadable fonts, autofill, and picture in picture (PIP). Android Oreo includes new APIs for these new capabilities, and these APIs are available to Xamarin.Android apps when you use Xamarin.Android 8.0 and later.



This article is structured to help you get started in developing Xamarin.Android apps for Android 8.0 Oreo. It explains how to install the necessary updates, configure the SDK, and create an emulator (or device) for testing. It also provides an outline of the new features in Android 8.0 Oreo, with links to sample apps that illustrate how to use Android Oreo features in Xamarin.Android apps.

Requirements

The following is required to use Android Oreo features in Xamarin-based apps:

- **Visual Studio** – If you are using Windows, version 15.5 or later of Visual Studio is required. If you are using a Mac, Visual Studio for Mac version 7.2.0 is required.
- **Xamarin.Android** – Xamarin.Android 8.0 or later must be installed and configured with Visual Studio.
- **Android SDK** – Android SDK 8.0 (API 26) or later must be installed via the Android SDK Manager.

Getting Started

To get started using Android Oreo with Xamarin.Android, you must download and install the latest tools and SDK packages before you can create an Android Oreo project:

1. Update to the latest version of Visual Studio.

2. Install the **Android 8.0.0 (API 26)** or later packages and tools via the SDK Manager.

3. Create a new Xamarin.Android project that targets Android Oreo (API 26).

4. Configure an emulator or device for testing Android Oreo apps.

Each of these steps is explained in the following sections:

Update Visual Studio and Xamarin.Android

To add Android Oreo support to Visual Studio, do the following:

- Visual Studio
- Visual Studio for Mac

- For Visual Studio 2019, use the SDK Manager to install API level 26.0 or later.

- If you are using Visual Studio 2017:

  1. Update to Visual Studio 2017 version 15.7 or later (see Update Visual Studio 2017).

  2. Use the SDK Manager to install API level 26.0 or later.

For more information about Xamarin support for Android Oreo, see the Xamarin.Android 8.0 release notes.

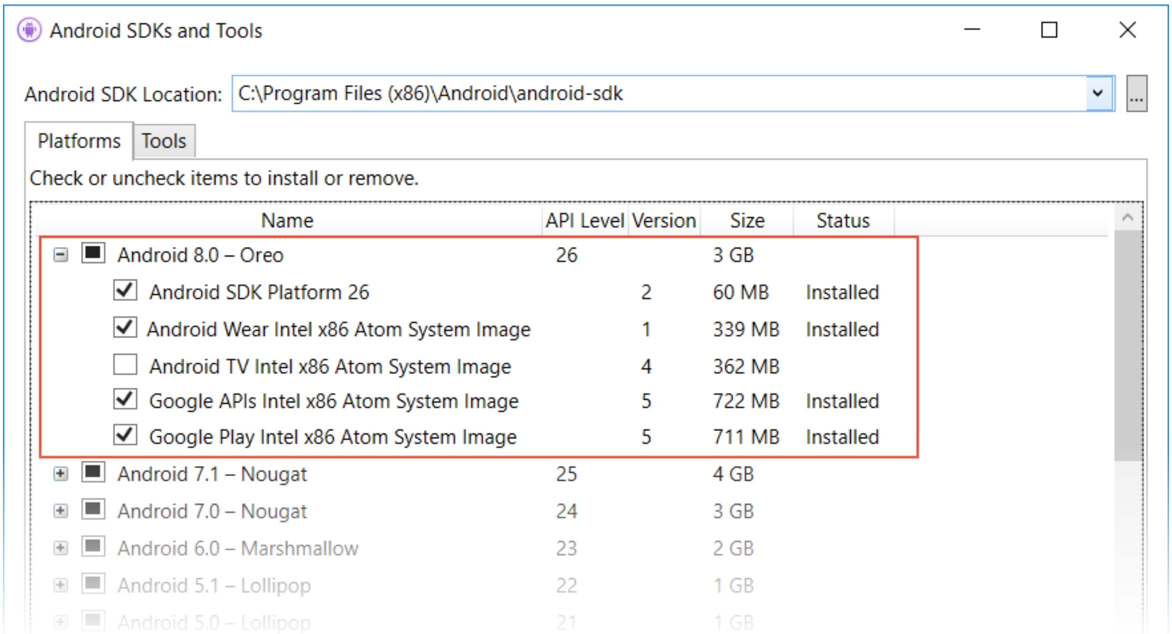
Install the Android SDK

To create a project with Xamarin.Android 8.0, you must first use the Xamarin Android SDK Manager to install the SDK platform for **Android 8.0 - Oreo** or later. You must also install Android SDK Tools 26.0 or later.

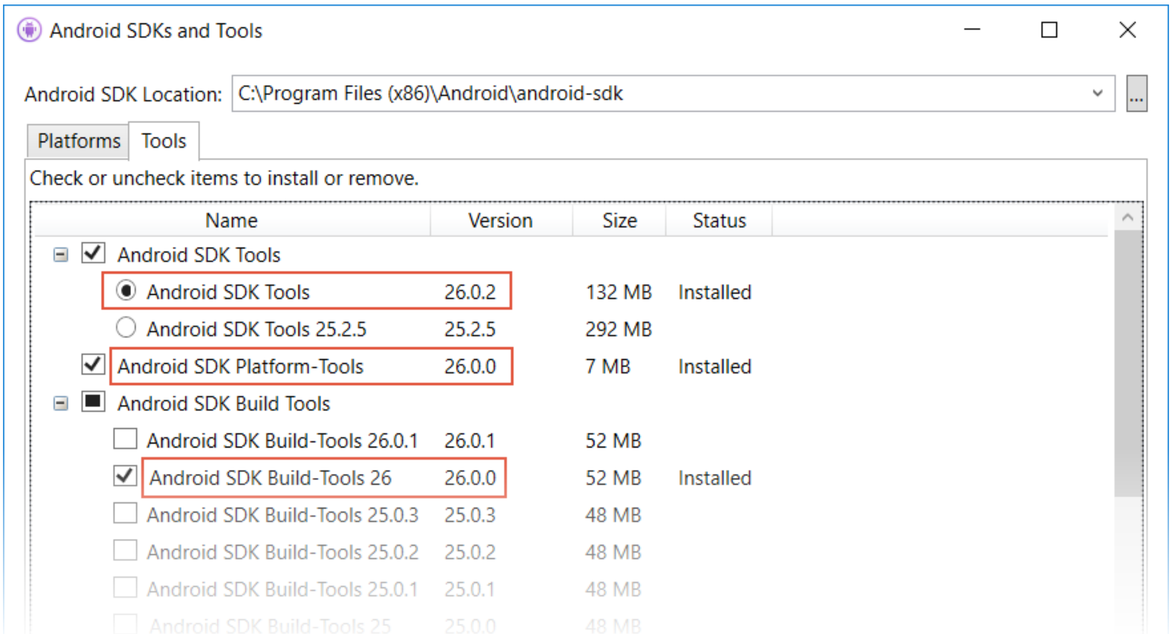
- Visual Studio
- Visual Studio for Mac

1. Start the SDK Manager (in Visual Studio, click **Tools > Android > Android SDK Manager**).

2. Install the **Android 8.0 - Oreo** packages. If you are using the Android SDK emulator, be sure to include the **x86** system images that you will need:



3. Install **Android SDK Tools 26.0.2** or later, **Android SDK Platform-Tools 26.0.0** or later, and **Android SDK Build-Tools 26.0.0** (or later):



Start a Xamarin.Android Project

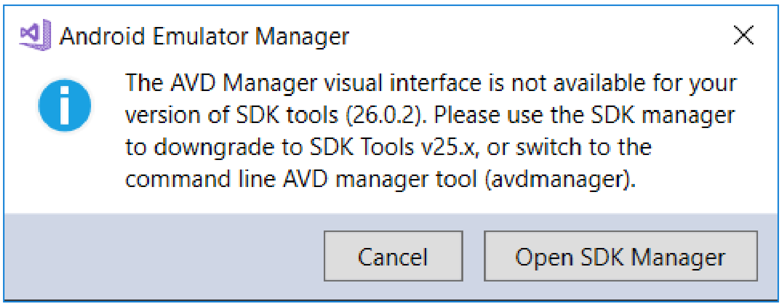
Create a new Xamarin.Android project. If you are new to Android development with Xamarin, see Hello, Android to learn about creating Xamarin.Android projects.

When you create an Android project, you must configure the version settings to target Android 8.0 or later. For example, to target your project for Android 8.0, you must configure the target Android API level of your project to **Android 8.0 (API 26)**. It is recommended that you also set your target framework level to API 26 or later. For more about configuring Android API level levels, see Understanding Android API Levels.

Configure an Emulator or Device

If you attempt to launch the default Google GUI-based AVD Manager after installing Android SDK Tools 26.0 or later, you may get the following error dialog, which instructs you to use the command line AVD manager tool **avdmanager** instead:

- Visual Studio
- Visual Studio for Mac



This message is displayed because Google no longer provides a standalone GUI AVD manager that supports API 26.0 and later. For Android 8.0 Oreo, you must use either the Xamarin Android Emulator Manager or the command-line `avdmanager` tool to create virtual devices for Android Oreo.

To use the Android Device Manager to create and manage virtual devices, see Managing Virtual Devices with the Android Device Manager. To create virtual devices without the Android Device Manager, follow the steps in the next section.

Creating Virtual Devices Using avdmanager

To use **avdmanager** to create a new virtual device, follow these steps:

- Visual Studio
- Visual Studio for Mac

1. Open a Command Prompt window and set `JAVA_HOME` to the location of the Java SDK on your computer. For a typical Xamarin installation, you can use the following command:

```
setx JAVA_HOME "C:\Program Files\Java\jdk1.8.0_131"
```

2. Add the location of the Android SDK `bin` folder to your `PATH`. For a typical Xamarin installation, you can use the following command:

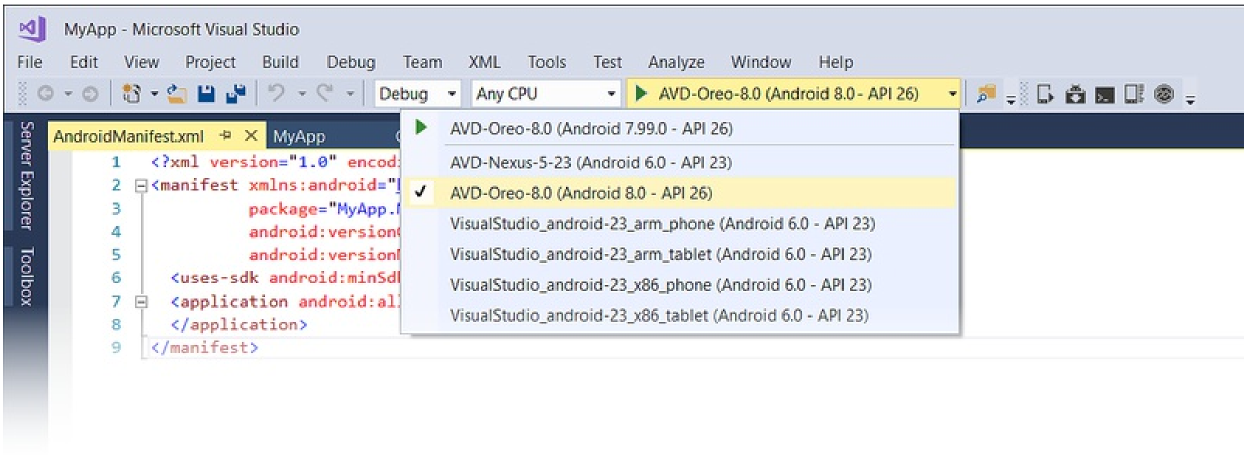
```
setx PATH "%PATH%;C:\Program Files (x86)\Android\android-sdk\tools\bin"
```

3. Close the Command Prompt window and open a new Command Prompt window. Create a new virtual device by using the avdmanager command. For example, to create an AVD named **AVD-Oreo-8.0** using the x86 system image for API level 26, use the following command:

```
avdmanager create avd -n AVD-Oreo-8.0 -k "system-images;android-26;google_apis;x86"
```

4. When you are prompted with **Do you wish to create a custom hardware profile [no]** you can enter **no** and accept the default hardware profile. If you say **yes**, **avdmanager** will prompt you with a list of questions for customizing the hardware profile.

After you **avdmanager** to create your virtual device, it will be included in the device pull-down menu:



For more information about configuring an Android emulator for testing and debugging, see Debugging on the Android Emulator.

If you are using a physical device such as a Nexus or a Pixel, you can either update your device through automatic over the air (OTA) updates or download a system image and flash your device directly. For more information about manually updating your device to Android Oreo, see Factory Images for Nexus and Pixel Devices.

New Features

Android Oreo introduces a variety of new features and capabilities, such as notification channels, notification badges, custom fonts in XML, downloadable fonts, autofill, and picture-in-picture. The following sections highlight these features and provide links to help you get started using them in your app.

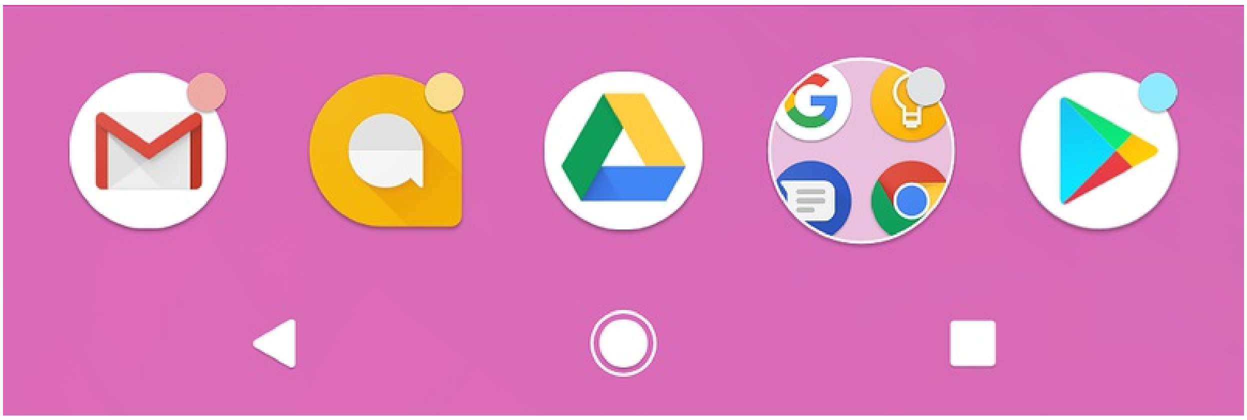
Notification Channels

Notification Channels are app-defined categories for notifications. You can create a notification channel for each type of notification that you need to send, and you can create notification channels to reflect choices made by users of your app. The new notification channels feature makes it possible for you to give users fine-grained control over different kinds of notifications. For example, if you are implementing a messaging app, you can create separate notification channels for each conversation group that is created by a user.

Notification Channels explains how to create a notification channel and use it for posting local notifications. For a real-world code example, see the NotificationChannels sample; this sample app manages two channels and sets additional notification options.

Notification Badges

Notification badges are small dots that appear over app icons as shown in this screenshot:



These dots indicate that there are new notifications for one or more notification channels in the app associated with that app icon – these are notifications that the user has not yet dismissed or acted upon. Users can long-press on an icon to glance at the notifications associated with a notification badge, dismissing or acting on notifications from the long-press menu that appeaars.

For more information about notification badges, see the Android Developer Notification Badges topic.

Custom Fonts in XML

Android Oreo introduces *Fonts in XML*, which makes it possible for you to incorporate custom fonts as resources. OpenType (**.otf**) and TrueType (**.ttf**) font formats are supported. To add fonts as resources, do the following:

1. Create a **Resources/font** folder.
2. Copy your font files (example, **.ttf** and **.otf** files) to **Resources/font**.
3. If necessary, rename each font file so that it adheres to the Android file naming conventions (i.e., use only lowercase *a-z*, *0-9*, and underscores in file names). For example, the font file `Pacifico-Regular.ttf` could be renamed to something like `pacifico.ttf`.
4. Apply the custom font by using the new `android:fontFamily` attribute in your layout XML. For example, the following `TextView` declaration uses the added **pacifico.ttf** font resource:

```
<TextView
  android:text="Example Text in Pacifico Regular"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:fontFamily="@font/pacifico" />
```

You can also create a font family XML file that describes multiple fonts as well as style and weight details. For more information, see the Android Developer Fonts in XML topic.

Downloadable Fonts

Beginning with Android Oreo, apps can request fonts from a provider rather than bundling them into the APK. Fonts are downloaded from the network only as needed. This feature reduces APK size, conserving phone memory and cellular data usage. You can also use this feature on Android API versions 14 and higher by installing the Android Support Library 26 package.

When your app needs a font, you create a `FontsRequest` object (specifying the font to download) and then pass it to a `FontsContract` method to download the font. The following steps describe the font download process in more detail:

1. Instantiate a FontRequest object.
2. Subclass and instantiate FontsContract.FontRequestCallback.
3. Implement the FontRequestCallback.OnTypeFaceRetrieved method, which is used to handle completion of the font request.
4. Implement the FontRequestCallback.OnTypeFaceRequestFailed method, which is used to inform your app of any errors that take place during the font request process.
5. Call the FontsContract.RequestFonts method to retrieve the font from the font provider.

When you call the `RequestFonts` method, it first checks to see if the font is locally cached (from a previous call to `RequestFont`). If it is not cached, it calls the font provider, retrieves the font asynchronously, and then passes the results back to your app by invoking your `OnTypeFaceRetrieved` method.

The Downloadable Fonts sample demonstrates how to use the Downloadable Fonts feature introduced in Android Oreo.

For more information about downloading fonts, see the Android Developer Downloadable Fonts topic.

Autofill

The new *Autofill* framework in Android Oreo makes it easier for users to handle repetitive tasks such as login, account creation, and credit card transactions. Users spend less time re-typing information (which can lead to input errors). Before your app can work with the Autofill Framework, an autofill service must be enabled in the system settings (users can enable or disable autofill).

The AutofillFramework sample demonstrates the use of the Autofill Framework. It includes implementations of client Activities with views that should be autofilled, and a Service that can provide autofill data to client Activities.

For more information about the new Autofill feature and how to optimize your app for autofill, see the Android Developer Autofill Framework topic.

Picture in Picture (PIP)

Android Oreo makes it possible for an Activity to launch in picture-in-picture (PIP) mode, overlaying the screen of another Activity. This feature is intended for video playback.

To specify that your app's Activity can use PIP mode, set the following flag to true in the Android manifest:

```
android:supportsPictureInPicture
```

To specify how your activity should behave when it is in PIP mode, you use the new PictureInPictureParams object. `PictureInPictureParams` represents a set of parameters that you use to initialize and update an Activity in PIP mode (for example, the Activity's preferred aspect ratio). The following new PIP methods were added to `Activity` in Android Oreo:

- EnterPictureInPictureMode – puts the Activity in PIP mode. The Activity is placed in the corner of the screen, and the rest of the screen is filled with the previous Activity that was on the screen.
- SetPictureInPictureParams – Updates the Activity's PIP configuration settings (for example, a change in aspect ratio).

The PictureInPicture sample demonstrates basic usage of the Picture-in-Picture (PiP) mode for handheld devices introduced in Oreo. The sample plays a video which continues uninterrupted while switching back and forth between display modes or other activities.

Other Features

Android Oreo contains many other new features such as the Emoji support library, Location API, background limits, wide-gamut color for apps, new audio codecs, WebView enhancements, improved keyboard navigation support, and a new AAudio (pro audio) API for high-performance low-latency audio, For more information about these features, see the Android Developer Android Oreo Features and APIs topic.

Behavior Changes

Android Oreo includes a variety of system and API behavior changes that can have an impact on the functionality of existing apps. These changes are described as follows.

Background Execution Limits

To improve the user experience, Android Oreo imposes limitations on what apps can do while running in the background. For example, if the user is watching a video or playing a game, an app running in the background could impair the performance of a video-intensive app running in the foreground. As a result, Android Oreo places the following restrictions on apps that are not directly interacting with the user:

1. **Background Service Limitations** – When an app is running in the background, it has a window of

several minutes in which it is still allowed to create and use services. At the end of that window, Android stops the app's background service and treats it as being *idle*.

2. **Broadcast Limitations** – Android 7.0 (API 25) placed limitations on broadcasts that an app registers to receive. Android Oreo makes these limitations more stringent. For example, Android Oreo apps can no longer register broadcast receivers for implicit broadcasts in their manifests.

For more information about the new background execution limits, see the Android Developer Background Execution Limits topic.

Breaking Changes

Apps that target Android Oreo or higher must modify their apps to support the following changes, where applicable:

- Android Oreo deprecates the ability to set the priority of individual notifications. Instead, you set a recommended importance level when creating a notification channel. The importance level you assign to a notification channel applies to all notification messages that you post to it.
- For apps targeting Android Oreo, `PendingIntent.GetService()` does not work due to new limits placed on services started in the background. If you are targeting Android Oreo, you should use PendingIntent.GetBroadcast instead.

Sample Code

Several Xamarin.Android samples are available to show you how to take advantage of Android Oreo features:

- NotificationsChannels demonstrates how to use the new Notification Channels system introduced in Android Oreo. This sample manages two notifications channels: one with default importance and the other with high importance.
- PictureInPicture demonstrates basic usage of the Picture-in-Picture (PiP) mode for handheld devices introduced in Oreo. The sample plays a video which continues uninterrupted while switching back and forth between display modes or other activities.
- AutofillFramework demonstrates the use of the Autofill Framework. It includes implementations of client Activities with views that should be autofilled, and a Service that can provide autofill data to client Activities.
- Downloadable Fonts provides an example of how to use the Downloadable Fonts feature described earlier.
- EmojiCompat demonstrates usage of EmojiCompat support library. You can use this library to prevent your app from showing missing emoji characters as "tofu" characters.
- Location Updates Pending Intent illustrates usage of the Location API to get updates about a device's location using a `PendingIntent`.
- Location Updates Foreground Service demonstrates how to use the Location API to get updates about a device's location using a bound and started foreground service.

Video

Android 8.0 Oreo development with C#

Summary

This article introduced Android Oreo and explained how to install and configure the latest tools and packages for Xamarin.Android development on Android Oreo. It provided an overview of the key features available in Android Oreo, with links to example source code for several new features. It included links to API documentation and Android Developer topics to help you get started in creating apps for Android Oreo. It also highlighted the most important Android Oreo behavior changes that could impact existing apps.

Related Links

- Android 8.0 Oreo

Nougat Features

7/8/2021 • 10 minutes to read • Edit Online

How to get started using Xamarin.Android to develop apps for Android Nougat.

This article provides an outline of the features introduced in Android Nougat, explains how to prepare Xamarin.Android for Android Nougat development, and provides links to sample applications that illustrate how to use Android Nougat features in Xamarin.Android apps.

Overview

Android Nougat is Google's followup to Android 6.0 Marshmallow. Xamarin.Android provides support for **Android 7.x Bindings** in Xamarin Android 7.0 and later. Android Nougat adds many new APIs for the Nougat features described below; these APIs are available to Xamarin.Android apps when you use Xamarin.Android 7.0.

For more information about Android 7.x APIs, see Android 7.1 for Developers. For a list of known Xamarin.Android 7.0 issues, please see the release notes.

Android Nougat provides many new features of interest to Xamarin.Android developers. These features include:

- **Multi-window support** – This enhancement makes it possible for users to open two apps on the screen at once.
- **Notification Enhancements** – The redesigned notifications system in Android Nougat includes a *Direct Reply* feature that allow users to quickly respond to text messages directly from the notification UI. Also, if your app creates notifications for received messages, the new *bundled notifications* feature can bundle notifications together as a single group when more than one message is received.
- **Data Saver** – This feature is a new system service that helps reduce cellular data use by apps; it gives users control over how apps use cellular data.

In addition, Android Nougat brings many other enhancements of interest to app developers such as a new network security configuration feature, Doze on the Go, key attestation, new Quick Settings APIs, multi-locale support, ICU4J APIs, WebView enhancements, access to Java 8 language features, scoped directory access, a custom pointer API, platform VR support, virtual files, and background processing optimizations.

This article explains how to get started building apps with Android Nougat to try out the new features and plan migration or feature work to target the new Android Nougat platform.

Requirements

The following is required to use the new Android Nougat features in Xamarin-based apps:

- **Visual Studio or Visual Studio for Mac** – If you are using Visual Studio, version 4.2.0.628 or later of Visual Studio Tools for Xamarin is required. If you are using Visual Studio for Mac, version 6.1.0 or later of Visual Studio for Mac is required.

- **Xamarin.Android** – Xamarin.Android 7.0 or later must be installed and configured with either Visual Studio or Visual Studio for Mac.

- **Android SDK** - Android SDK 7.0 (API 24) or later must be installed via the Android SDK Manager.

- **Java Developer Kit** – Xamarin Android 7.0 development requires JDK 8 or later if you are developing for API level 24 or greater (JDK 8 also supports API levels earlier than 24). The 64-bit version of JDK 8 is required if you are using custom controls or the Forms Previewer.

> **IMPORTANT**
>
> Xamarin.Android does not support JDK 9.

Note that apps must be rebuilt with Xamarin C6SR4 or later to work reliably with Android Nougat. Because Android Nougat can link only to NDK-provided native libraries, existing apps using libraries such as **Mono.Data.Sqlite.dll** may crash when running on Android Nougat if they are not properly rebuilt.

Getting Started

To get started using Android Nougat with Xamarin.Android, you must download and install the latest tools and SDK packages before you can create an Android Nougat project:

1. Install the latest Xamarin.Android updates from the Xamarin.

2. Install the **Android 7.0 (API 24)** packages and tools or later.

3. Create a new Xamarin.Android project that targets Android Nougat.

4. Configure an emulator or device for Android Nougat.

Each of these steps is explained in the following sections:

Install Xamarin Updates

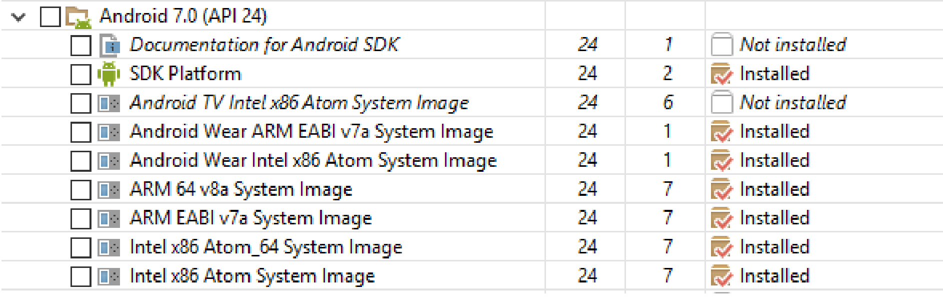
To add Xamarin support for Android Nougat, change the updates channel in Visual Studio or Visual Studio for Mac to the Stable channel and apply the latest updates. If you also need features that are currently available only in the Alpha or Beta channel, you can switch to the Alpha or Beta channel (the Alpha and Beta channels also provide support for Android 7.x). For information about how to change the updates (releases) channel, see Changing the Updates Channel.

Install the Android SDK

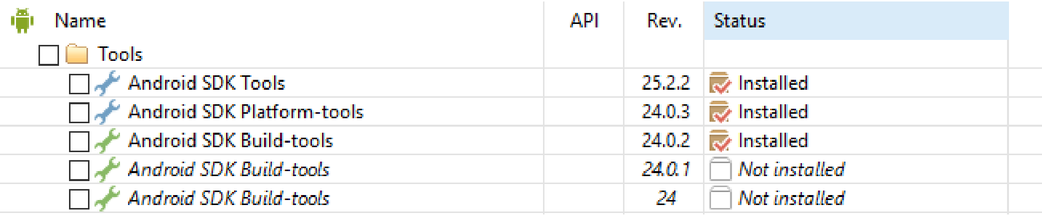
To create a project with Xamarin Android 7.0, you must first use the Android SDK Manager to install **SDK Platform Android N (API 24)** or later. You must also install the latest **Android SDK Tools**:

1. Start the Android SDK Manager (in Visual Studio for Mac, use **Tools > Open Android SDK Manager...**; in Visual Studio, use **Tools > Android > Android SDK Manager**).

2. Install **Android 7.0 (API 24)** or later:

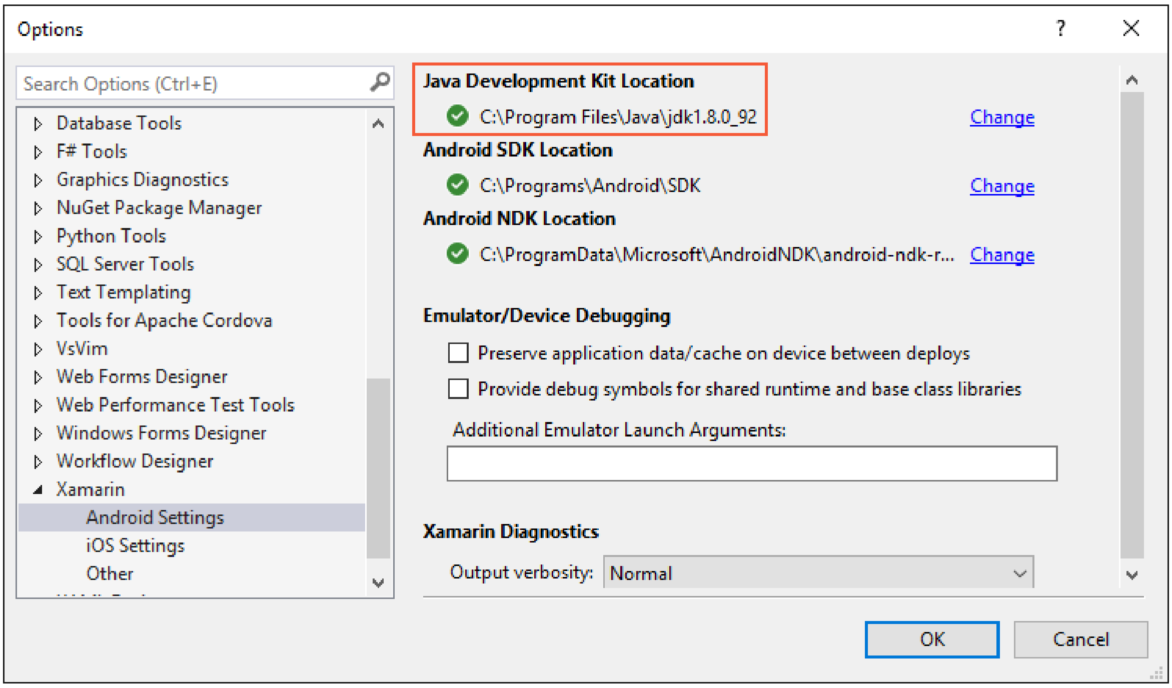


3. Install the latest Android SDK tools:



You must install Android SDK Tools revision 25.2.2 or later, Android SDK Platform tools 24.0.3 or later, and Android SDK Build tools 24.0.2 or later.

4. Verify that the **Java Development Kit Location** is configured for JDK 1.8:



To view this setting in Visual Studio, click **Tools > Options > Xamarin > Android Settings**. In Visual Studio for Mac, click **Preferences > Projects > SDK Locations > Android**.

Start a Xamarin.Android Project

Create a new Xamarin.Android project. If you are new to Android development with Xamarin, see Hello, Android to learn about creating Xamarin.Android projects.

When you create an Android project, you must configure the version settings to target Android 7.0 or later. For example, to target your project for Android 7.0, you must configure the target Android API level of your project to **Android 7.0 (API 24 - Nougat)**. It is recommended that you set your target framework level to API 24 or later. For more about configuring Android API level levels, see Understanding Android API Levels.

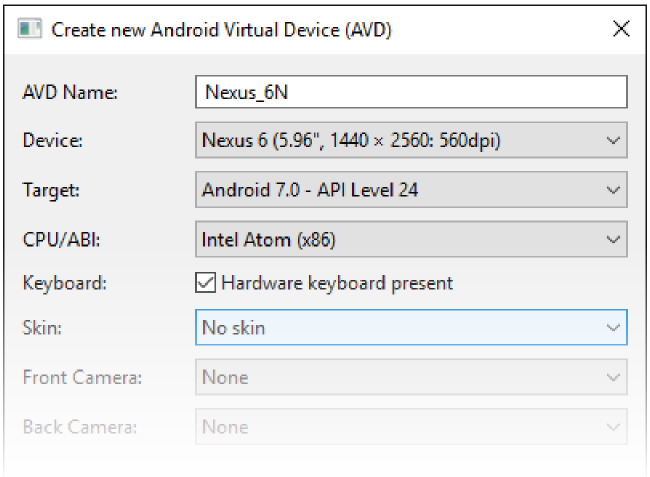
> **NOTE**
>
> Currently you must set the **Minimum Android version** to **Android 7.0 (API 24 - Nougat)** to deploy your app to Android Nougat devices or emulators.

Configure an Emulator or Device

If you are using an emulator, start the Android AVD Manager and create a new device using the following settings:

- Device: Nexus 5X, Nexus 6, Nexus 6P, Nexus Player, Nexus 9, or Pixel C.
- Target: Android 7.0 - API Level 24
- ABI: x86 or x86_64

For example, this virtual device is configured to emulate a Nexus 6:



If you are using a physical device such as a Nexus 5X, 6, or 9, you can either update your device through automatic over the air (OTA) updates or download a system image and flash your device directly. For more information about manually updating your device to Android Nougat, see OTA Images for Nexus Devices.

Note that Nexus 5 devices are not supported by Android Nougat.

New Features

Android Nougat introduces a variety of new features and capabilities, such as Multi-window Support, Notifications enhancements, and Data Saver. The following sections highlight these features and provide links to help you get started using them in your app.

Multi-Window Mode

Multi-window mode makes it possible for users to open two apps at once with full multitasking support. These apps can run side-by-side (landscape) or one-above-the-other (portrait) in split-screen mode. Users can drag a divider between the apps to resize them, and they can cut and paste content the between apps. When two apps are presented in multi-window mode, the selected activity continues to run while the unselected activity is paused but still visible. Multi-window mode does not modify the Android activity lifecycle.

You can configure how the activities of your Xamarin.Android app support multi-window mode. For example, you can configure attributes that set the minimum size and the default height and width of your app in multi-window mode. You can use the new `Activity.IsInMultiWindowMode` property to determine if your activity is in multi-window mode. For example:

```
if (!IsInMultiWindowMode) {
    multiDisabledMessage.Visibility = ViewStates.Visible;
} else {
    multiDisabledMessage.Visibility = ViewStates.Gone;
}
```

The MultiWindowPlayground sample app includes C# code that demonstrates how to take advantage of multiple window user interfaces with your app.

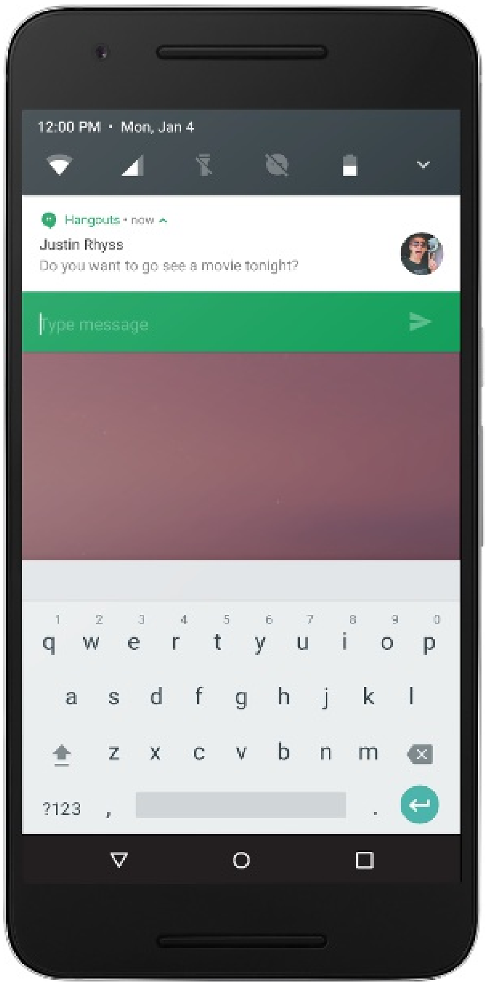
For more information about multi-window mode, see the Multi-Window Support.

Enhanced Notifications

Android Nougat introduces a redesigned notification system. It features a new Direct Reply feature that makes it possible for users to quickly reply to notifications for incoming text messages directly in the notification UI. Starting with Android 7.0, notification messages can be bundled together as a single group when more than one message is received. Also, developers can customize notification views, leverage system decorations in notifications, and take advantage of new notification templates when generating notifications.

Direct Reply

When a user receives a notification for incoming message, Android Nougat makes it possible to reply to the message within the notification (rather than open up the messaging app to send a reply). This inline reply feature makes it possible for users to quickly respond to an SMS or text message directly within the notification interface:



To support this feature in your app, you must add *inline reply actions* to your app via a RemoteInput object so that users can reply via text directly from the notification UI. For example, the following code builds a `RemoteInput` for receiving text input, builds a pending intent for the reply action, and creates a remote input enabled action:

```
// Build a RemoteInput for receiving text input:
var remoteInput = new Android.Support.V4.App.RemoteInput.Builder (EXTRA_REMOTE_REPLY)
    .SetLabel (GetString (Resource.String.reply))
    .Build ();

// Build a Pending Intent for the reply action to trigger:
PendingIntent replyIntent = PendingIntent.GetBroadcast (ApplicationContext,
                                conversation.ConversationId,
                                GetMessageReplyIntent (conversation.ConversationId),
                                PendingIntentFlags.UpdateCurrent);

// Build an Android 7.0 compatible Remote Input enabled action:
NotificationCompat.Action actionReplyByRemoteInput = new NotificationCompat.Action.Builder (
    Resource.Drawable.notification_icon,
    GetString (Resource.String.reply),
    replyIntent).AddRemoteInput (remoteInput).Build ();
```

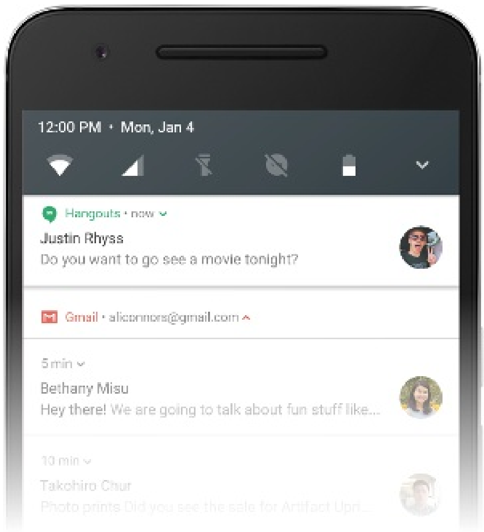
This action is added to the notification:

```
// Create the notification:
NotificationCompat.Builder builder = new NotificationCompat.Builder (ApplicationContext)
   .SetSmallIcon (Resource.Drawable.notification_icon)
   ...
   .AddAction (actionReplyByRemoteInput);
```

The Messaging Service sample app includes C# code that demonstrates how to extend notifications with a `RemoteInput` object. For more information about adding inline reply actions to your app for Android 7.0 or later, see the Android Replying to Notifications topic.

Bundled Notifications

Android Nougat can group notification messages together (for example, by message topic) and display the group rather than each separate message. This *bundled notifications* feature makes it possible for users to dismiss or archive a group of notifications in one action. The user can slide down to expand the bundle of notifications to view each notification in detail:



To support bundled notifications, your app can use the Builder.SetGroup method to bundle similar notifications. For more information about bundled notification groups in Android N, see the Android Bundling Notifications topic.

Custom Views

Android Nougat makes it possible for you to create custom notification views with system notification headers, actions, and expandable layouts. For more information about custom notification views in Android Nougat, see the Android Notification Enhancements topic.

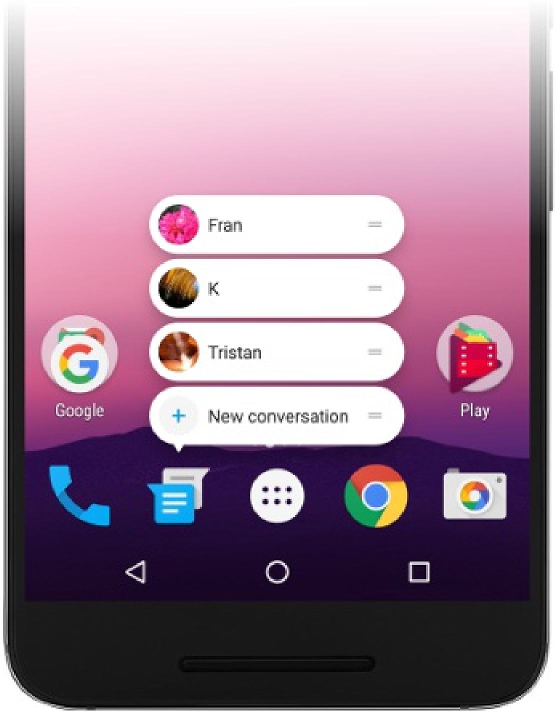
Data Saver

Beginning with Android Nougat, users can enable a new *Data Saver* setting that blocks background data usage. This setting also signals your app to use less data in the foreground wherever possible. The ConnectivityManager has been extended in Android Nougat so that your app can check whether the user has enabled Data Saver so that your app can make an effort to limit its data usage when Data Saver is enabled.

For more information about the new Data Saver feature in Android Nougat, see the Android Optimizing Network Data Usage topic.

App Shortcuts

Android 7.1 introduced an *App Shortcuts* feature that makes it possible for users to quickly start common or recommended tasks with your app. To activate the menu of shortcuts, the user long-presses the app icon for a second or more – the menu appears with a quick vibration. Releasing the press causes the menu to remain:



This feature is available only API level 25 or higher. For more information about the new App Shortcuts feature in Android 7.1, see the Android App Shortcuts topic.

Sample Code

Several Xamarin.Android samples are available to show you how to take advantage of Android Nougat features:

- MultiWindowPlayground demonstrates the use of the multi-window API available in Android Nougat. You can switch the sample app into multi-windows mode to see how it affects the app's lifecycle and behavior.
- Messaging Service is a simple service that sends notifications using the `NotificationCompatManager`. It also extends the notification with a `RemoteInput` object to allow Android Nougat devices to reply via text directly from the notification without having to open an app.
- Active Notifications demonstrates how to use the `NotificationManager` API to tell you how many notifications your application is currently displaying.
- Scoped Directory Access Demonstrates how to use the scoped directory access API to easily access specific directories. This serves as an alternative to having to define `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE` permissions in your manifest.
- Direct Boot Illustrates how to store data in a device-encrypted storage which is always available while the device is booted both before and after any user credentials(PIN/Pattern/Password) are entered.

Summary

This article introduced Android Nougat and explained how to install and configure the latest tools and packages for Xamarin.Android development on Android Nougat. It also provided an overview of the key features available in Android Nougat, with links to example source code to help you get started in creating apps for Android Nougat.

Related Links

- Android 7.1 For Developers
- Xamarin Android 7.0 Release Notes

Marshmallow Features

7/8/2021 • 11 minutes to read • Edit Online

This article helps you get started using in using Xamarin.Android to develop apps for Android 6.0 Marshmallow.

This article provides an outline of the new features in Android 6.0 Marshmallow, explains how to prepare Xamarin.Android for Android Marshmallow development, and provides links to sample applications that illustrate how to make use of new Android Marshmallow features in Xamarin.Android apps.

Overview

Android 6.0 Marshmallow, is the next major Android release after Android Lollipop. Xamarin.Android supports Android Marshmallow and includes:

- **API 23/Android 6.0 Bindings** – Android 6.0 adds many new APIs for the new features described below; these APIs are available to Xamarin.Android apps when you target API Level 23. For more information about Android 6.0 APIs, see Android 6.0 APIs.

Although the Marshmallow release is mainly focused on "polish and quality", it also provides many new features of interest to Xamarin.Android developers. These features include:

- **Runtime Permissions** – This enhancement makes it possible for users to approve security permissions on a case-by-case basis at run time.
- **Authentication Improvements** – Starting with Android Marshmallow, apps can now use fingerprint sensors to authenticate users, and a new *confirm credential* feature minimizes the need for entering passwords.
- **App Linking** – This feature helps to eliminate the necessity of having the **App Chooser** pop up by automatically associating apps with web domains.
- **Direct Share** – You can define *direct share targets* that make sharing quick and intuitive for users; this feature allows uers to share content with other apps.
- **Voice Interactions** – This new API allows you to build conversational voice features into your app.
- **4K Display Mode** – In Android Marshmallow, your app can request a 4K display resolution on hardware that supports it.
- **New Audio Features** – Starting with Marshmallow, Android now supports the MIDI protocol. It also

provides new classes to create digital audio capture and playback objects, and it offers new API hooks for associating audio and input devices.

- **New Video Features** – Marshmallow provides a new class that helps apps render audio and video streams in sync; this class also provides support for dynamic playback rate.

- **Android for Work** – Marshmallow includes enhanced controls for corporate-owned, single-user devices. It supports silent install and uninstall of apps by the device owner, auto-acceptance of system updates, improved certificate management, data usage tracking, permissions management, and work status notifications.

- **Material Design Support Library** – The new *Design Support Library* provides design components and patterns that makes it easier for you to build Material Design look and feel into your app.

In addition, many core Android library updates were released with Android M, and these updates provide new features for both Android M and earlier versions of Android.

In addition, many core Android library updates were released with Android Marshmallow, and these updates provide new features for both Android Marshmallow and earlier versions of Android. This article explains how to get started building apps with Android Marshmallow, and it provides an overview of the new feature highlights in Android 6.0.

Requirements

The following is required to use the new Android Marshmallow features in Xamarin-based apps:

- **Xamarin.Android** – Xamarin.Android 5.1.7.12 or later must be installed and configured with either Visual Studio or Xamarin Studio.

- **Visual Studio for Mac** or **Visual Studio** – If you are using Visual Studio for Mac, version 5.9.7.22 or later is required. If you are using Visual Studio, version 3.11.1537 or later of the Xamarin tools for Visual Studio is required.

- **Android SDK** – Android SDK 6.0 (API 23) or later must be installed via the Android SDK Manager.

- **Java Developer Kit** – Xamarin.Android requires JDK 1.8 or later if you are developing for API level 24 or greater (JDK 1.8 also supports API levels earlier than 24, including Marshmallow). The 64-bit version of JDK 1.8 is required if you are using custom controls or the Forms Previewer.

You can continue to use JDK 1.7 if you are developing specifically for API level 23 or earlier.

Getting Started

To get started using Android Marshmallow with Xamarin.Android, you must download and install the latest tools and SDK packages before you can create an Android Marshmallow project:

1. Install the latest Xamarin updates from the **Stable** channel.

2. Install the Android 6.0 Marshmallow SDK packages and tools.

3. Create a new Xamarin.Android project that targets Android 6.0 Marshmallow (API Level 23).

4. Configure an emulator or device for Android Marshmallow.

Each of these steps is explained in the following sections:

Install Xamarin Updates

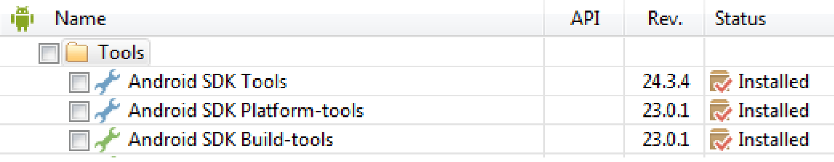
To update Xamarin so that it includes support for Android 6.0 Marshmallow, change the update channel to **Stable** and install all updates. For more information about installing updates from the updates channel, see

[Change the Updates Channel](#).

Install the Android 6.0 SDK

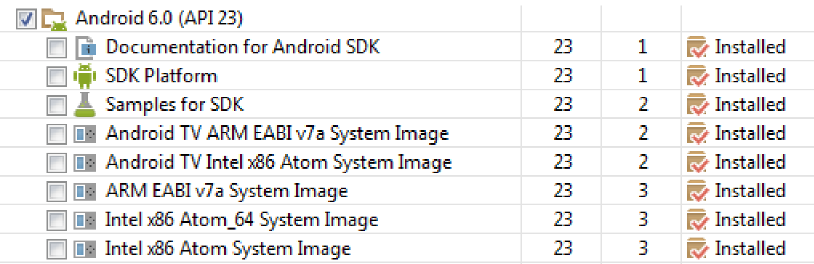
To create a Xamarin.Android project for Android Marshmallow, you must first use the Android SDK Manager to install the Android 6.0 SDK:

- Start the Android SDK Manager (in Visual Studio for Mac, use **Tools > SDK Manager**; in Visual Studio, use **Tools > Android > Android SDK Manager**) and install the latest Android SDK Tools:



Name	API	Rev.	Status
Tools			
Android SDK Tools		24.3.4	Installed
Android SDK Platform-tools		23.0.1	Installed
Android SDK Build-tools		23.0.1	Installed

- Also, install the latest **Android 6.0** SDK packages:



Name	API	Rev.	Status
Android 6.0 (API 23)			
Documentation for Android SDK	23	1	Installed
SDK Platform	23	1	Installed
Samples for SDK	23	2	Installed
Android TV ARM EABI v7a System Image	23	2	Installed
Android TV Intel x86 Atom System Image	23	2	Installed
ARM EABI v7a System Image	23	3	Installed
Intel x86 Atom_64 System Image	23	3	Installed
Intel x86 Atom System Image	23	3	Installed

You must install Android SDK Tools revision 24.3.4 or later. For more information about using the Android SDK Manager to install the Android 6.0 SDK, see [SDK Manager](#).

Start a Xamarin.Android Project

Create a new Xamarin.Android project. If you are new to Android development with Xamarin, see [Hello, Android](#) to learn about creating Android projects.

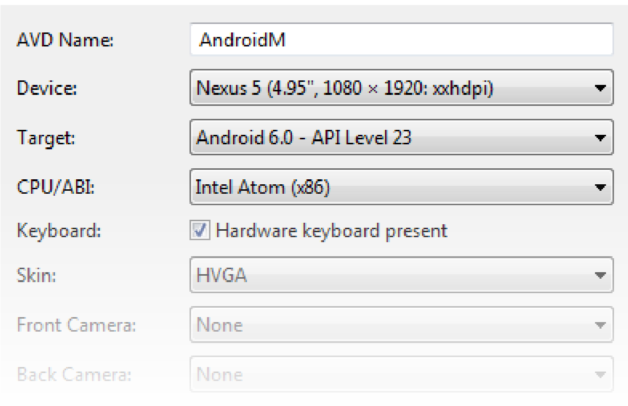
When you create an Android project, you must configure the version settings to target Android 6.0 MarshMallow. To target your project for Marshmallow, you must configure your project for **API level 23 (Xamarin.Android v6.0 Support)**. For more about configuring Android API level levels, see [Understanding Android API Levels](#).

Configure an Emulator or Device

If you are using an emulator, start the Android AVD Manager and create a new device using the following settings:

- Device: Nexus 5, 6, or 9.
- Target: Android 6.0 - API Level 23
- ABI: x86

For example, this virtual device is configured to emulate a Nexus 5:



If you are using a physical device such as a Nexus 5, 6, or 9, you can install a preview image of Android Marshmallow. For more information about updating your device to Android Marshmallow, see Hardware System Images.

New Features

Many of the changes introduced in Android Marshmallow are focused on improving the Android user experience, increasing performance, and fixing bugs. However, Marshmallow also introduced some extensive changes to the fundamentals of the Android platform. The following sections highlight these enhancements and provide links to help you get started in using the new Android Marshmallow features in your app.

Runtime Permissions

The Android Permissions system has been significantly optimized and simplified since Android Lollipop. In Android Marshmallow, users grant permissions on a case-by-case basis at runtime rather than at install time. To support this feature on Android Marshmallow and later, you design your app to prompt the user for permissions at runtime (in the context of where the permissions are needed). This change makes it easier for users to start using your app immediately because it streamlines the process of installing and upgrading your app.

See Requesting Runtime Permissions in Android Marshmallow for more details (including code examples) about implementing Runtime Permissions in Xamarin.Android apps. Xamarin also provides a sample app that illustrates how runtime permissions work in Android Marshmallow (and later): RuntimePermissions.

This sample app demonstrates the following:

- How to check and request permissions at run time.
- How to declare permissions for Android M devices.

To use this sample app:

1. Tap the **Camera** or **Contacts** buttons to display a permissions request dialog.
2. Grant permission to view Camera or Contacts fragments.

For more information about the new runtime permissions features in Android Marshmallow, see Working with System Permissions.

Authentication Enhancements

Android Marshmallow includes two authentication enhancements that help eliminate the need for passwords:

- **Fingerprint Authentication** – Uses a fingerprint scan to authenticate users.
- **Confirm Credential** – Authenticates users based on how long the device has been unlocked.

The links and sample apps described next can help you become familiar with these new features.

Fingerprint Authentication

On devices that support fingerprint scanning hardware, you can use the new `FingerPrintManager` class to authenticate a user. For more information about the fingerprint authentication feature in Android Marshmallow, see Fingerprint Authentication.

Xamarin provides a sample app that illustrates how to use registered fingerprints to authenticate a user in your app: FingerprintDialog.

To use this sample app:

1. Touch the **Purchase** button to open a fingerprint authentication dialog.
2. Scan in your registered fingerprint to authenticate.

Note that this sample app requires a device with a fingerprint reader. This app does not store your fingerprint (or your password).

Voice Interactions

The new Voice Interactions feature introduced in Android Marshmallow allows users of your app to use their voice to confirm actions and select from a list of options. For more information about Voice Interactions, see Overview of the Voice Interaction API.

See Add a Conversation to your Android App with Voice Interactions for more details (including code examples) about implementing Voice Interactions in Xamarin.Android apps. A sample app is available that illustrates how to use the Voice Interaction API in a Xamarin.Android app: Voice Interactions.

Confirm Credential

Using the new *confirm credential* feature of Android Marshmallow, you can free users from having to remember and enter app-specific passwords by authenticating them based on how long their device has been unlocked. To do this, you use the new `SetUserAuthenticationValidityDurationSeconds` method of the `KeyGenerator`. Use the `KeyGuardManager`'s `CreateConfirmDeviceCredentialIntent` method to re-authenticate the user from within your app. For more information about this new feature in Android Marshmallow, see Confirm Credential.

Xamarin provides a sample app that illustrates how to use device credentials (such as PIN, pattern, or password) in your app: ConfirmCredential

To use this sample app:

1. Setup a secure lock screen on your device (**Secure > Security > Screenlock**).
2. Tap the **Purchase** button and confirm the secure lock screen credentials.

Chrome Custom Tabs

App developers face a choice when a user taps a URL: the app can either launch a browser or use an in-app browser based on a `WebView`. Both options present challenges – launching the browser is a heavy context switch that isn't customizable, while `WebView`s do not share state with the browser. Also, use of `WebView`s can add extra maintenance overhead.

Chrome Custom Tabs makes it possible for you to easily and elegantly display websites with the power of Chrome without having your users leave your app. This feature gives your app more control over the user's web experience; it make transitions between native and web content more seamless without having to resort to a `WebView`. Your app can also affect how Chrome looks and feels by customizing the following:

- Toolbar color
- Enter and exit animations
- Custom actions in the Chrome toolbar and overflow menu
- Chrome pre-start and content pre-fetch (for faster loading)

To take advantage of this feature in your Xamarin.Android app, download and install the Android Support Custom Tabs Library. For more information about this feature, see Chrome Custom Tabs.

Material Design Support Library

Android Lollipop introduced Material Design as a new design language to refresh the Android experience (see Material Theme for information about using material design in Xamarin.Android apps). With Android Marshmallow, Google introduced the *Android Design Support Library* to make it easier for app developers to adopt material design look and feel. This library includes the following components:

- **CoordinatorLayout** – The new `CoordinatorLayout` widget is similar to but more powerful than a `FrameLayout`. You can use `CoordinatorLayout` as a container for child views or as a top-level layout, and it provides a `layout_anchor` attribute that can be used to anchor views relative to other views.

- **Collapsing Toolbars** – The new `CollapsingToolbarLayout` is a collapsing app bar that is a wrapper for `Toolbar`. (Note that the *app bar* is what was formerly referred to as the *action bar*.)

- **Floating Action Button** – A round button that denotes the primary action on your app's interface.

- **Floating Labels for Editing Text** – Uses a new `TextInputLayout` widget (which wraps `EditText`) to show a floating label when a hint is hidden when a user inputs text.

- **Navigation View** – The new `NavigationView` widget helps you use the navigation drawer in a way that is easier for users to navigate.

- **Snackbar** – The new `SnackBar` widget is a lightweight feedback mechanism (similar to a toast) that displays a brief message at the bottom of the screen, appearing above all other elements on the screen.

- **Material Tabs** – The new `TabLayout` widget provides a horizontal layout for displaying tabs as way to implement top-level navigation in your app.

To take advantage of the Design Support Library in your Xamarin.Android app, download and install the Xamarin Xamarin Support Library Design NuGet package.

See Beautiful Material Design with the Android Support Design Library for more details (including code examples) about using the Material Design Support Library in Xamarin.Android apps. Xamarin provides a sample app that demos the new Android Design library on Xamarin.Android – Cheesesquare. This sample demonstrates the following features of the Design library:

- Collapsing toolbar
- Floating action button
- View anchoring
- NavigationView
- Snackbar

For more information about the Design library, see Android Design Support Library in the Android Developer's blog.

Additional Library Updates

In addition to Android Marshmallow, Google has announced related updates to several core Android libraries. Xamarin provides Xamarin.Android support for these updates through several preview-release NuGet packages:

- Google Play Services – The latest version of Google Play Services includes the new *App Invites* feature, which makes it possible for users to share their app with friends. For more information about this feature, see Expand Your App's Reach with Google's App Invites.

- Android Support Libraries – These NuGets offer features that are only available for library APIs while providing backward-compatible versions of the Android framework APIs.

- Android Wearable Library – this NuGet includes Google Play Services bindings. The latest version of the wearable library brings new features (including easier navigation for custom apps) to the Android Wear platform.

Summary

This article introduced Android Marshmallow and explained how to install and configure the latest tools and packages for Xamarin.Android development on Marshmallow. It also provided an overview of the most exciting new Android Marshmallow features for Xamarin.Android development.

Related Links

- Android 6.0 Marshmallow
- Get the Android SDK
- Feature Overview
- Release Notes
- RuntimePermissions (sample)
- ConfirmCredential (sample)
- FingerprintDialog (sample)

Lollipop Features

7/8/2021 • 21 minutes to read • Edit Online

This article provides a high level overview of the new features introduced in Android 5.0 (Lollipop). These features include a new user interface style called Material Theme, as well as new supporting features such as animations, view shadows, and drawable tinting. Android 5.0 also includes enhanced notifications, two new UI widgets, a new job scheduler, and a handful of new APIs to improve storage, networking, connectivity, and multimedia capabilities.

Lollipop Overview

Android 5.0 (Lollipop) introduces a new design language, *Material Design*, and with it a supporting cast of new features to make apps easier and more intuitive to use. With Material Design, Android 5.0 not only gives Android phones a facelift; it also provides a new set of design rules for Android-based tablets, desktop computers, watches, and smart TVs. These design rules emphasize simplicity and minimalism while making use of familiar tactile attributes (such as realistic surface and edge cues) to help users quickly and intuitively understand the interface.

Material Theme is the embodiment of these UI design principles in Android. This article begins by covering Material Theme's supporting features:

- **Animations** – *Touch feedback* animations, *activity transition* animations, *view state transition* animations, and a *reveal effect*.
- **View shadows and elevation** – Views now have an `elevation` property; views with higher `elevation` values cast larger shadows on the background.
- **Color features** – *Drawable tinting* makes it possible for you to reuse image assets by changing their color, and *prominent color extraction* helps you dynamically theme your app based on colors in an image.

Many Material Theme features are already built into the Android 5.0 UI experience, while others must be explicitly added to apps. For example, some standard views (such as buttons) already include touch feedback animations, while apps must enable most view shadows.

In addition to the UI improvements brought about through Material Theme, Android 5.0 also includes several other new features that are covered in this article:

- **Enhanced notifications** – Notifications in Android 5.0 have been significantly updated with a new look, support for lockscreen notifications, and a new *Heads-up* notification presentation format.
- **New UI widgets** – The new `RecyclerView` widget makes it easier for apps to convey large data sets and complex information, and the new `CardView` widget provides a simplified card-like presentation format for displaying text and images.
- **New APIs** – Android 5.0 adds new APIs for multiple network support, improved Bluetooth connectivity, easier storage management, and more flexible control of multimedia players and camera devices. A new job scheduling feature is available to run tasks asynchronously at scheduled times. This feature helps to improve battery life by, for example, scheduling tasks to take place when the device is plugged in and charging.

Requirements

The following is required to use the new Android 5.0 features in Xamarin-based apps:

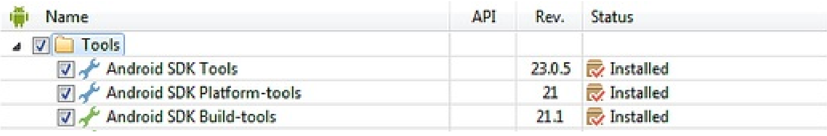
- **Xamarin.Android** – Xamarin.Android 4.20 or later must be installed and configured with either Visual Studio or Visual Studio for Mac.
- **Android SDK** – Android 5.0 (API 21) or later must be installed via the Android SDK Manager.
- **Java Developer Kit** – Xamarin.Android requires JDK 1.8 or later if you are developing for API level 24 or greater (JDK 1.8 also supports API levels earlier than 24, including Lollipop). The 64-bit version of JDK 1.8 is required if you are using custom controls or the Forms Previewer.

You can continue to use JDK 1.7 if you are developing specifically for API level 23 or earlier.

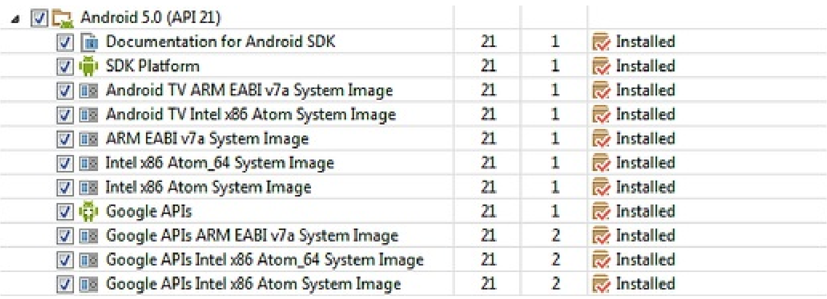
Setting Up an Android 5.0 Project

To create an Android 5.0 project, you must install the latest tools and SDK packages. Use the following steps to set up a Xamarin.Android project that targets Android 5.0:

1. Install Xamarin.Android tools and activate your Xamarin license. See Setup and Installation for more information about installing Xamarin.Android.
2. If you are using Visual Studio for Mac, install the latest Android 5.0 updates.
3. Start the Android SDK Manager (in Visual Studio for Mac, use **Tools > Open Android SDK Manager...**) and install Android SDK Tools 23.0.5 or later:

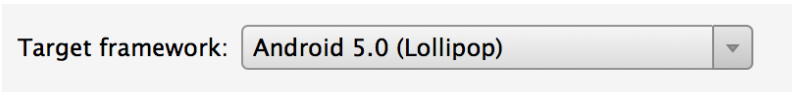


Also, install the latest Android 5.0 SDK packages (API 21 or later):

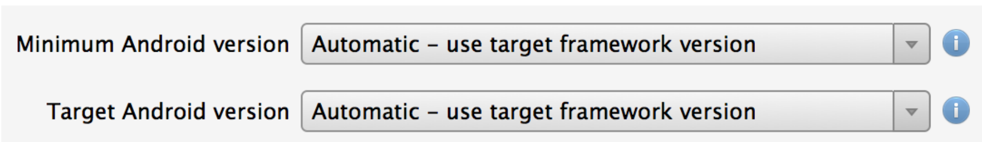


For more information about using the Android SDK Manager, see SDK Manager.

4. Create a new Xamarin.Android project. If you are new to Android development with Xamarin, see Hello, Android to learn about creating Android projects. When you create an Android project, be sure to configure the version settings for Android 5.0. In Visual Studio for Mac, navigate to **Project Options > Build > General** and set **Target framework** to **Android 5.0 (Lollipop)** or later:



Under **Project Options > Build > Android Application**, set minimum and target Android version to **Automatic - use target framework version**:



5. Configure an emulator or an Android device to test your app. If you are using an emulator, see Android

Emulator Setup to learn how to configure an Android emulator for use with Xamarin Studio or Visual Studio. If you are using an Android device, see Setting Up the Preview SDK to learn how to update your device for Android 5.0. To configure your Android device for running and debugging Xamarin.Android applications, see Set Up Device for Development.

Note: If you are updating an existing Android project that was targeting the Android L Preview, you must update the **Target Framework** and **Android version** to the values described above.

Important Changes

Previously published Android apps could be affected by changes in Android 5.0. In particular, Android 5.0 uses a new runtime and a significantly changed notification format.

Android Runtime

Android 5.0 uses the new Android Runtime (ART) as the default runtime instead of Dalvik. ART implements several major new features:

- **Ahead-of-time (AOT) compilation** – AOT can improve app performance by compiling app code before the app is first launched. When an app is installed, ART generates a compiled app executable for the target device.

- **Improved garbage collection (GC)** – GC improvements in ART can also improve app performance. Garbage collection now uses one GC pause instead of two, and concurrent GC operations complete in a more timely fashion.

- **Improved app debugging** – ART provides more diagnostic detail to help in analyzing exceptions and crash reports.

Existing apps should work without change under ART – except for apps that exploit techniques unique to the previous Dalvik runtime, which may not work under ART. For more information about these changes, see Verifying App Behavior on the Android Runtime (ART).

Notification Changes

Notifications have changed significantly in Android 5.0:

- **Sounds and vibration are handled differently** – Notification sounds and vibrations are now handled by `Notification.Builder` instead of `Ringtone`, `MediaPlayer`, and `Vibrator`.

- **New color scheme** – In accordance with Material Theme, notifications are rendered with dark text over white or very light backgrounds. Also, alpha channels in notification icons may be modified by Android to coordinate with system color schemes.

- **Lockscreen notifications** – Notifications can now appear on the device lockscreen.

- **Heads-up** – High-priority notifications now appear in a small floating window (Heads-up notification) when the device is unlocked and the screen is turned on.

In most cases, porting existing app notification functionality to Android 5.0 requires the following steps:

1. Convert your code to use `Notification.Builder` (or `NotificationsCompat.Builder`) for creating notifications.

2. Verify that your existing notification assets are viewable in the new Material Theme color scheme.

3. Decide what visibility your notifications should have when they are presented on the lockscreen. If a notification is not public, what content should show up on the lockscreen?

4. Set the category of your notifications so they are handled correctly in the new Android 5.0 *Do not disturb*

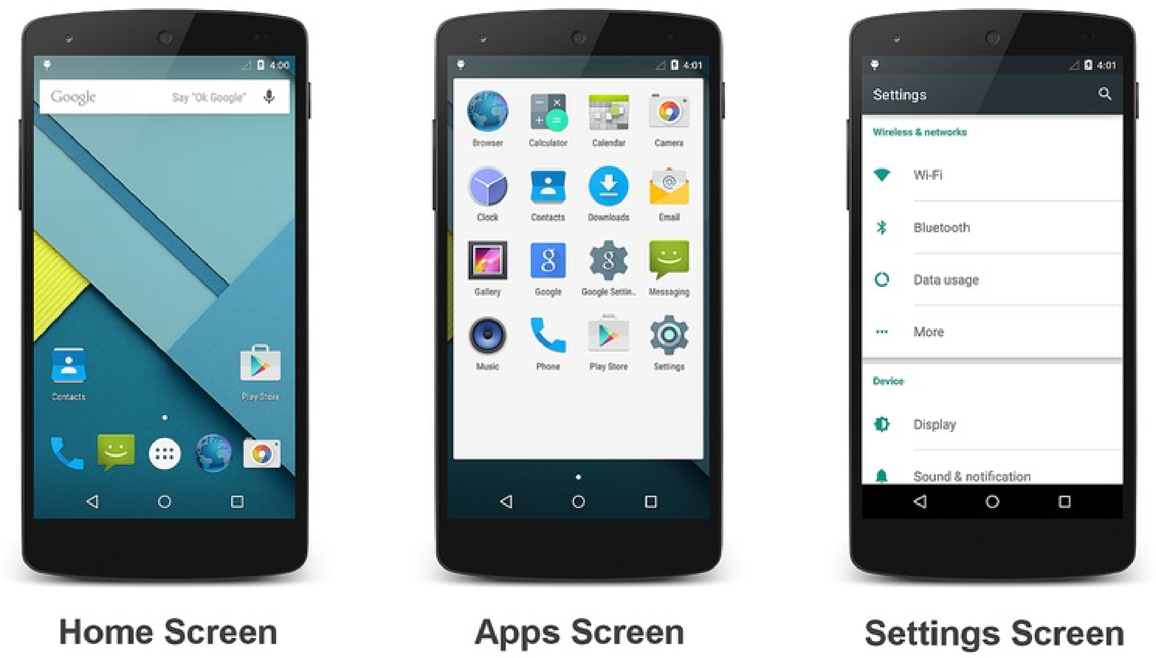
mode.

If your notifications present transport controls, display media playback status, use `RemoteControlClient`, or call `ActivityManager.GetRecentTasks`, see Important Behavior Changes for more information about updating your notifications for Android 5.0.

For information about creating notifications in Android, see Local Notifications.

Material Theme

The new Android 5.0 Material Theme brings sweeping changes to the look and feel of the Android UI. Visual elements now use tactile surfaces that take on the bold graphics, typography, and bright colors of print-based design. Examples of Material Theme are depicted in the following screenshots:



Android 5.0 greets you with the home screen shown on the left. The center screenshot is the first screen of the app list, and the screenshot on the right is the **Settings** screen. Google's Material Design specification explains the underlying design rules behind the new Material Theme concept.

Material Theme includes three built-in flavors that you can use in your app: the `Theme.Material` dark theme (the default), the `Theme.Material.Light` theme, and the `Theme.Material.Light.DarkActionBar` theme:

Theme.Material   Theme.Material.Light   Theme.Material.Light.DarkActionBar

For more about using Material Theme features in Xamarin.Android apps, see Material Theme.

Animations

Android 5.0 provides touch feedback animations, activity transition animations, and view state transition animations to make app interfaces more intuitive to use. Also, Android 5.0 apps can use *reveal effect* animations to hide or reveal views. You can use *curved motion* settings to configure how quickly or slowly animations are rendered.

Touch Feedback Animations

Touch feedback animations provide users with visual feedback when a view has been touched. For example, buttons now display a ripple effect when they are touched – this is the default touch feedback animation in Android 5.0. Ripple animation is implemented by the new `RippleDrawable` class. The ripple effect can be configured to end at the bounds of the view or extend beyond the bounds of the view. For example, the following sequence of screenshots illustrates the ripple effect in a button during touch animation:

Initial touch contact with the button occurs in the first image on the left, while the remaining sequence (from left to right) illustrates how the ripple effect spreads out to the edge of the button. When the ripple animation ends, the view returns to its original appearance. The default ripple animation takes place in a fraction of a second, but the length of the animation can be customized for longer or shorter lengths of time.

For more on touch feedback animations in Android 5.0, see Customize Touch Feedback.

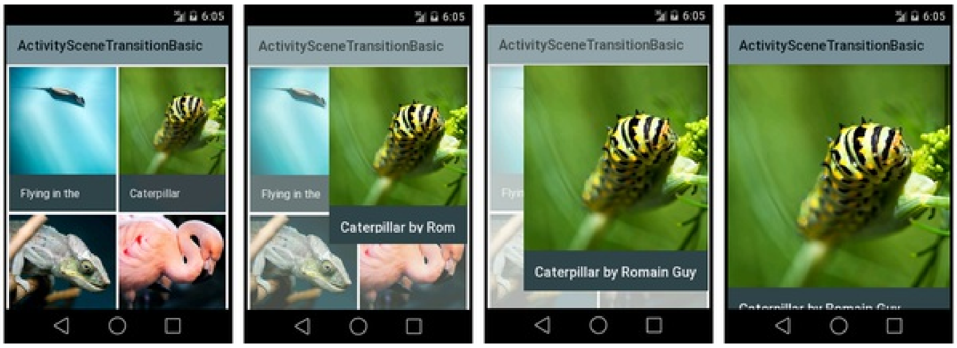
Activity Transition Animations

Activity transition animations give users a sense of visual continuity when one activity transitions to another. Apps can specify three types of transition animations:

- **Enter transition** – For when an activity enters the scene.
- **Exit transition** – For when an activity exits the scene.
- **Shared element transition** – For when a view that is common to two activities changes as the first

activity transitions to the next.

For example, the following sequence of screenshots illustrates a shared element transition:



A shared element (a photo of a caterpillar) is one of several views in the first activity; it enlarges to become the only view in the second activity as the first activity transitions to the second.

Enter Transition Animation Types

For enter transitions, Android 5.0 provides three types of animations:

- **Explode animation** – Enlarges a view from the center of the scene.
- **Slide animation** – Moves a view in from one of the edges of a scene.
- **Fade animation** – Fades a view into the scene.

Exit Transition Animation Types

For exit transitions, Android 5.0 provides three types of animations:

- **Explode animation** – Shrinks a view to the center of the scene.
- **Slide animation** – Moves a view out to one of the edges of a scene.
- **Fade animation** – Fades a view out of the scene.

Shared Element Transition Animation Types

Shared element transitions support multiple types of animations, such as:

- Changing the layout or clip bounds of a view.
- Changing the scale and rotation of a view.
- Changing the size and scale type for a view.

For more about activity transition animations in Android 5.0, see Customize Activity Transitions.

View State Transition Animations

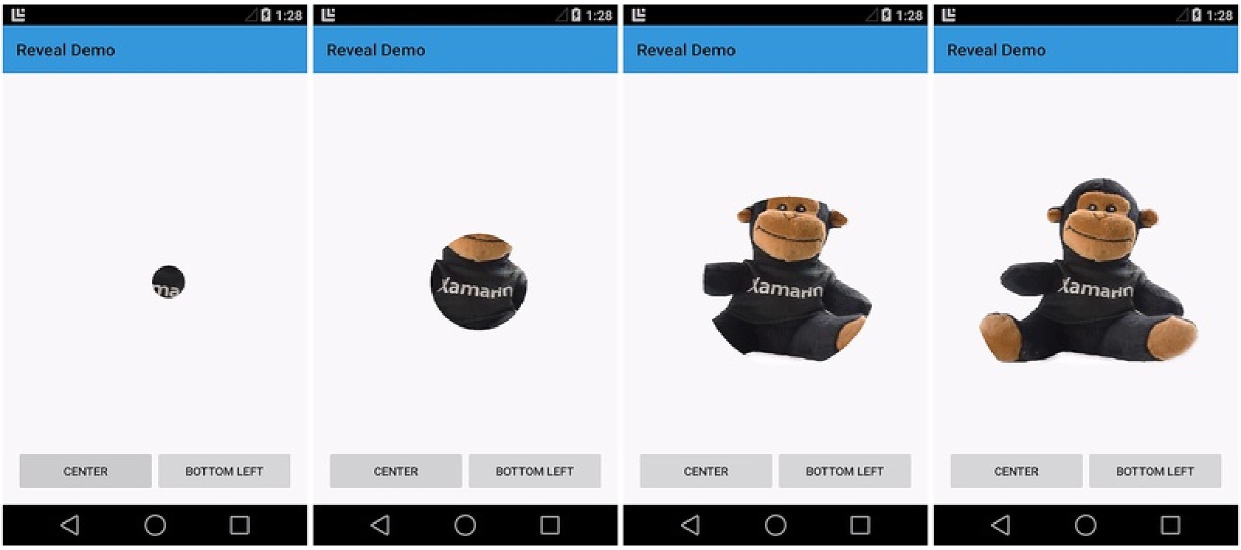
Android 5.0 makes it possible for animations to run when the state of a view changes. You can animate view state transitions by using one of the following techniques:

- Create drawables that animate state changes associated with a particular view. The new `AnimatedStateListDrawable` class lets you create drawables that display animations between view state changes.
- Define animation functionality that runs when the state of a view changes. The new `StateListAnimator` class lets you define an animator that runs when the state of a view changes.

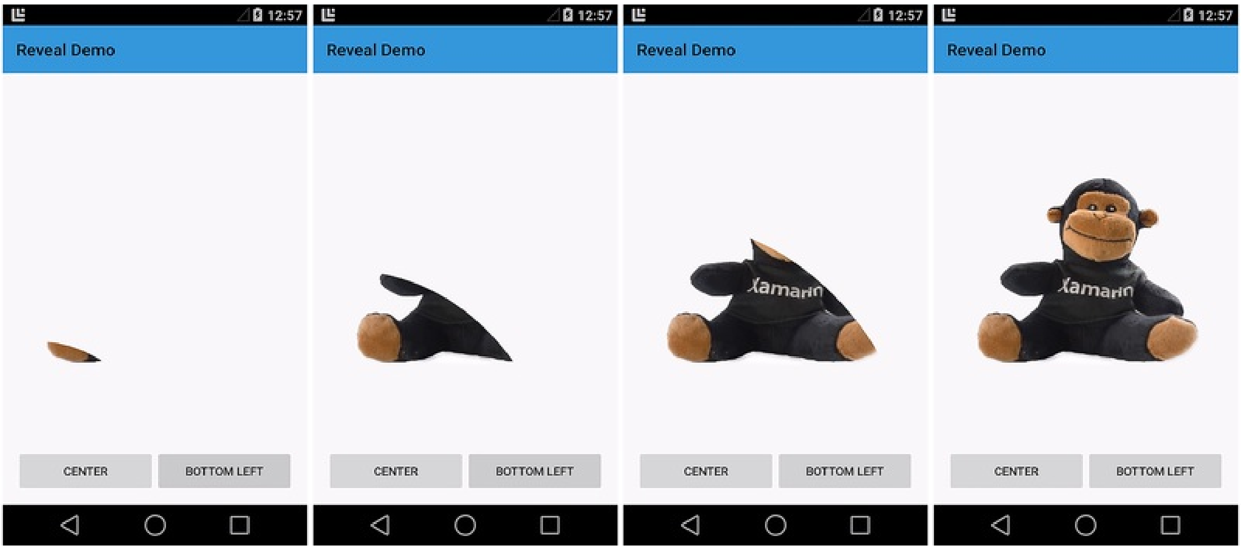
For more about view state transition animations in Android 5.0, see Animate View State Changes.

Reveal Effect

The *reveal effect* is a clipping circle that changes radius to reveal or hide a view. You can control this effect by setting the initial and final radius of the clipping circle. The following sequence of screenshots illustrates a reveal effect animation from the center of the screen:



The next sequence illustrates a reveal effect animation that takes place from the bottom left corner of the screen:



Reveal animations can be reversed; that is, the clipping circle can shrink to hide the view rather than enlarge to reveal the view.

For more information on the Android 5.0 reveal effect in, see Use the Reveal Effect.

Curved Motion

In addition to these animation features, Android 5.0 also provides new APIs that enable you to specify the time and motion curves of animations. Android 5.0 uses these curves to interpolate temporal and spatial movement during animations. Three curves are defined in Android 5.0:

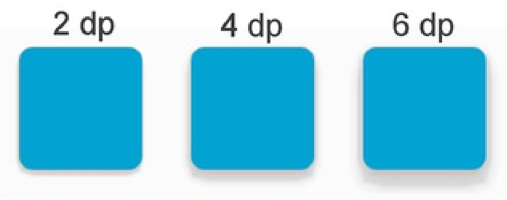
- **Fast_out_linear_in** – Accelerates quickly and continues to accelerate until the end of the animation.
- **Fast_out_slow_in** – Accelerates quickly and slowly decelerates towards the end of the animation.
- **Linear_out_slow_in** – Begins with a peak velocity and slowly decelerates to the end of the animation.

You can use the new `PathInterpolator` class to specify how motion interpolation takes place. `PathInterpolator` is an interpolator that traverses animation paths according to specified control points and motion curves. For more information about how to specify curved motion settings in Android 5.0, see Use Curved Motion.

View Shadows & Elevation

In Android 5.0, you can specify the *elevation* of a view by setting a new `Z` property. A greater `Z` value causes the view to cast a larger shadow on the background, making the view appear to float higher above the background. You can set the initial elevation of a view by configuring its `elevation` attribute in the layout.

The following example illustrates the shadows cast by an empty `TextView` control when its elevation attribute is set to 2dp, 4dp, and 6dp, respectively:



View shadow settings can be static (as shown above) or they can be used in animations to make a view appear to temporarily rise above the view's background. You can use the `ViewPropertyAnimator` class to animate the elevation of a view. The elevation of a view is the sum of its layout `elevation` setting plus a `translationZ` property that you can set through a `ViewPropertyAnimator` method call.

For more about view shadows in Android 5.0, see Defining Shadows and Clipping Views.

Color Features

Android 5.0 provides two new features for managing color in apps:

- *Drawable tinting* lets you alter the colors of image assets by changing a layout attribute.
- *Prominent color extraction* makes it possible for you to dynamically customize your app's color theme to coordinate with the color palette of a displayed image.

Drawable Tinting

Android 5.0 layouts recognize a new `tint` attribute that you can use to set the color of drawables without having to create multiple versions of these assets to display different colors. To use this feature, you define a bitmap as an alpha mask and use the `tint` attribute to define the color of the asset. This makes it possible for you to create assets once and color them in your layout to match your theme.

In the following example, a single image asset – a white logo with a transparent background – is used to create tint variations:

This logo is displayed above a blue circular background as shown in the following examples. The image on the left is how the logo appears without a `tint` setting. In the center image, the logo's `tint` attribute is set to a dark gray. In the image on the right, `tint` is set to a light gray:

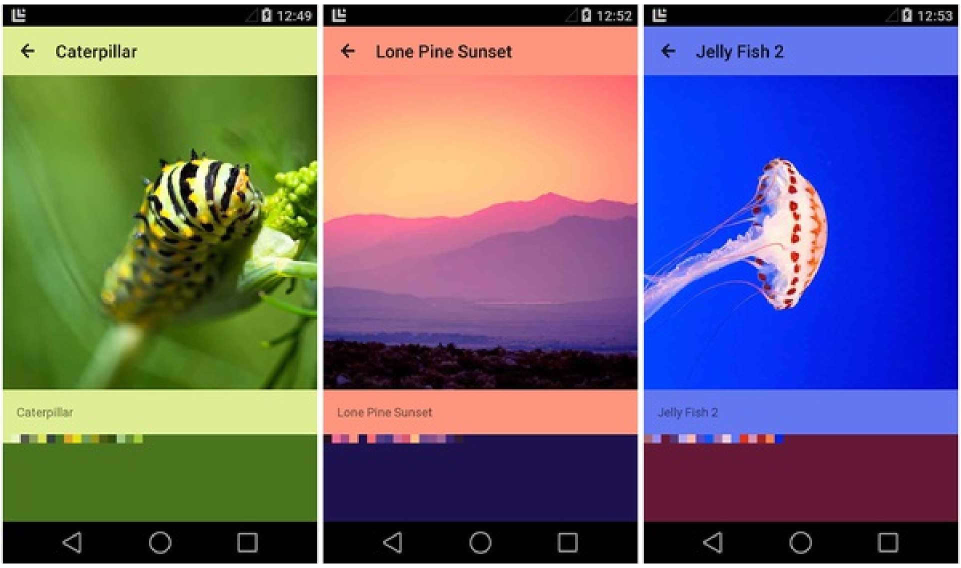
For more about drawable tinting in Android 5.0, see Drawable Tinting.

Prominent Color Extraction

The new Android 5.0 `Palette` class lets you extract colors from an image so that you can dynamically apply them to a custom color palette. The `Palette` class extracts six colors from an image and labels these colors according to their relative levels of color saturation and brightness:

- Vibrant
- Vibrant dark
- Vibrant light
- Muted
- Muted dark
- Muted light

For example, in the following screenshots, a photo viewing app extracts the prominent colors from the image on display and uses these colors to adapt the color scheme of the app to match the image:



In the above screenshots, the action bar is set to the extracted "vibrant light" color and the background is set to the extracted "vibrant dark" color. In each example above, a row of small color squares is included to illustrate the palette colors that were extracted from the image.

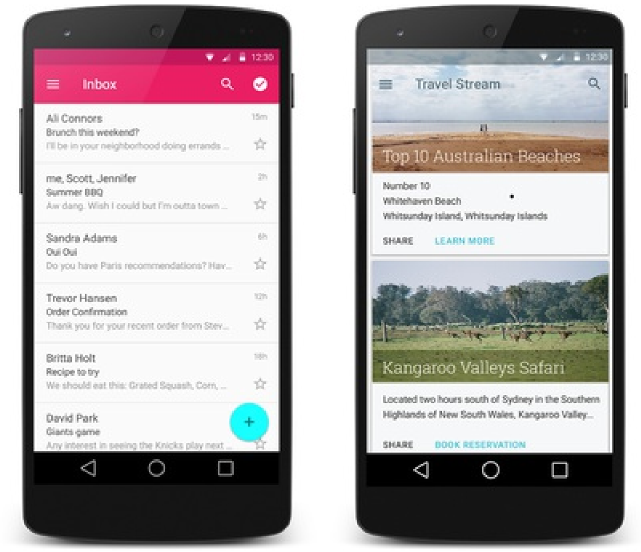
For more about color extraction in Android 5.0, see Extracting Prominent Colors from an Image.

New UI Widgets

Android 5.0 introduces two new UI widgets:

- `RecyclerView` – A view group that displays a list of scrollable items.
- `CardView` – A basic layout with rounded corners.

Both widgets include baked-in support for Material Theme features; for example, `RecyclerView` uses animations for adding and removing views, and `CardView` uses view shadows to make each card appear to float above the background. Examples of these new widgets are shown in the following screenshots:

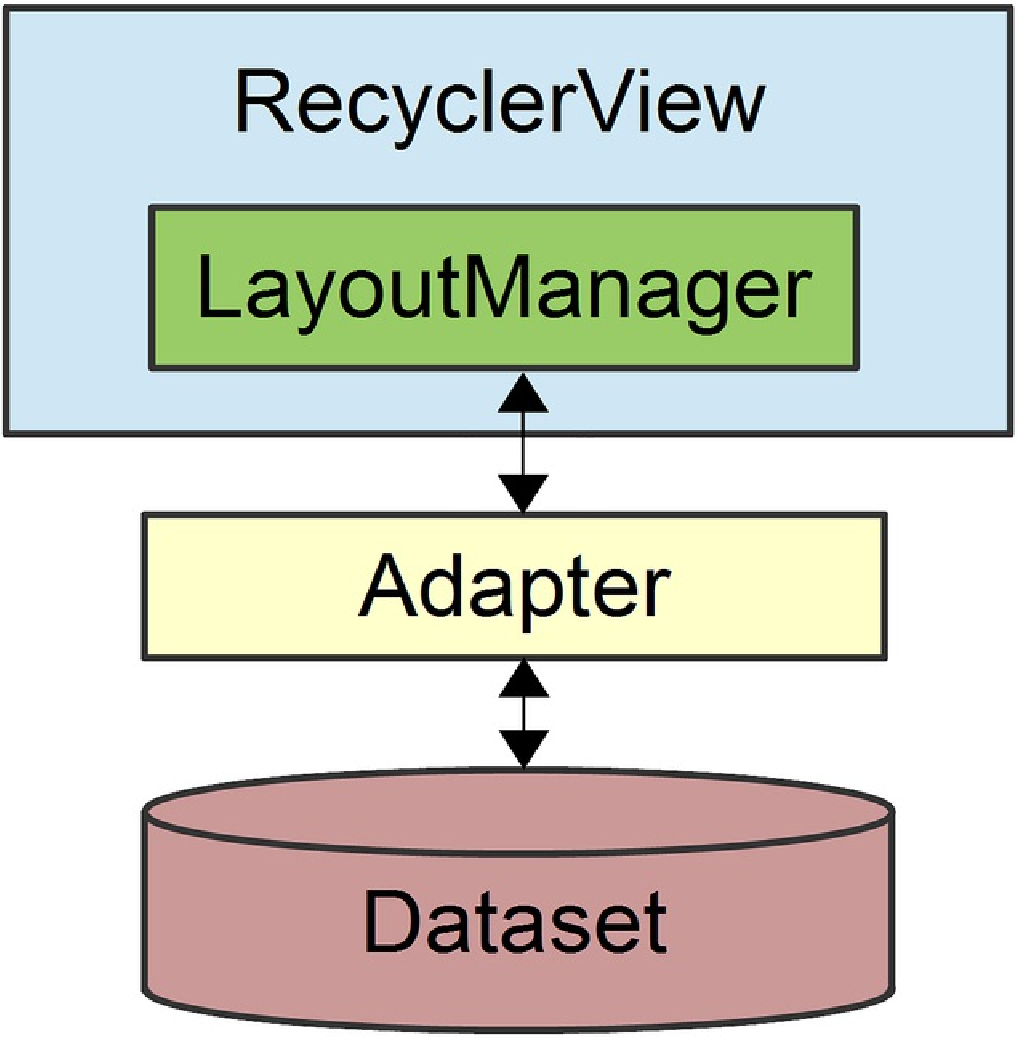


The screenshot on the left is an example of `RecyclerView` as used in an email app, and the screenshot on the right is an example of `CardView` as used in a travel reservation app.

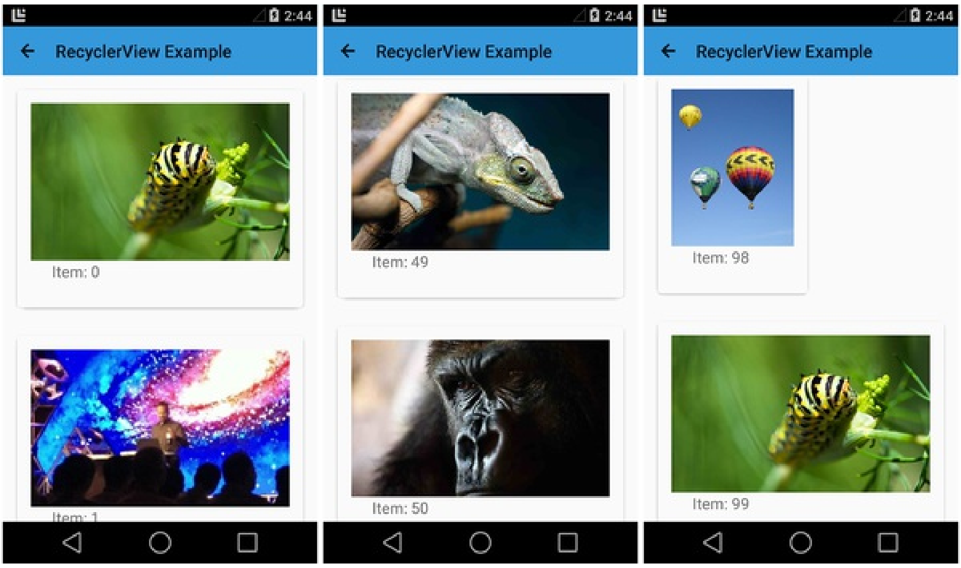
RecyclerView

`RecyclerView` is similar to `ListView,` but it is better suited for large sets of views or lists with elements that change dynamically. Like `ListView,` you specify an adapter to access the underlying data set. However, unlike `ListView,` you use a *layout manager* to position items within `RecyclerView`. The layout manager also takes care of view recycling; it manages the reuse of item views that are no longer visible to the user.

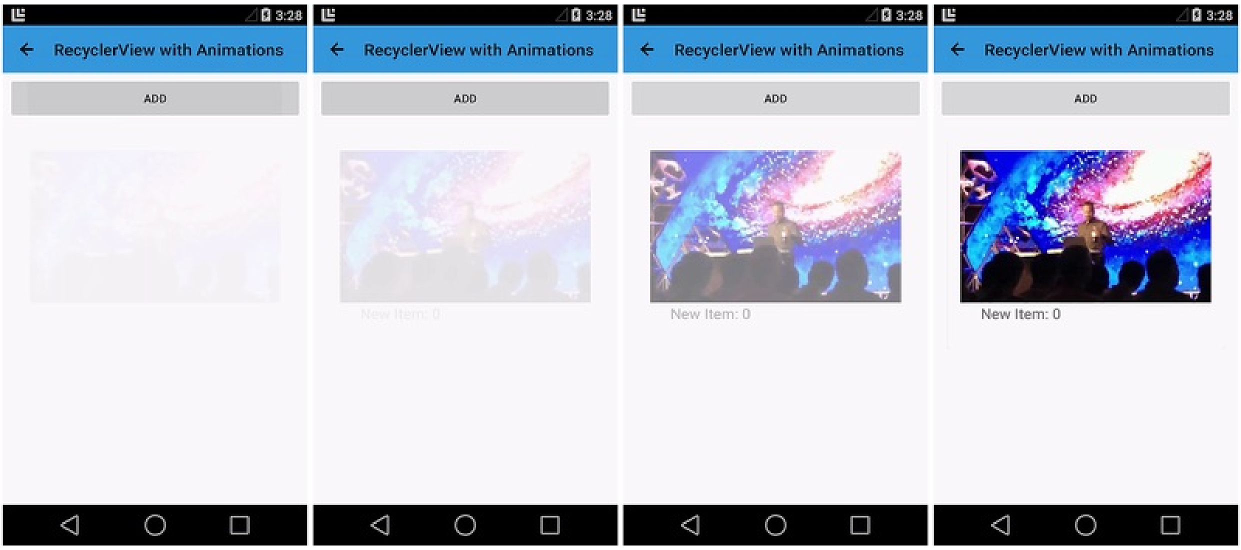
When you use a `RecyclerView` widget, you must specify a `LayoutManager` and an adapter. As shown in this figure, `LayoutManager` is the intermediary between the adapter and the `RecyclerView`:



The following screenshots illustrate a `RecyclerView` that contains 100 items (each item consists of an `ImageView` and a `TextView`):



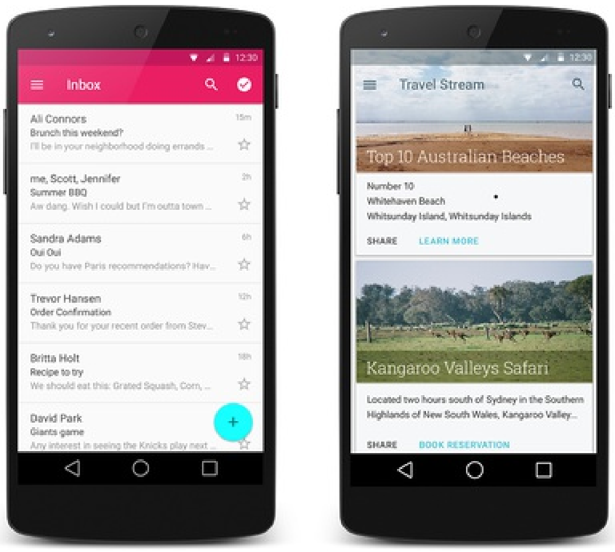
`RecyclerView` handles this large data set with ease – scrolling from the beginning of the list to end of the list in this sample app takes only a few seconds. `RecyclerView` also supports animations; in fact, animations for adding and removing items are enabled by default. When an item is added to a `RecyclerView`, it fades in as shown in this sequence of screenshots:



For more about `RecyclerView`, see RecyclerView.

CardView

`CardView` is a simple view that simulates a floating card with rounded corners. Because `CardView` has built-in view shadows, it provides an easy way for you to add visual depth to your app. The following screenshots show three text-oriented examples of `CardView`:



Each of the cards in the above example contains a `TextView`; the background color is set via the `cardBackgroundColor` attribute.

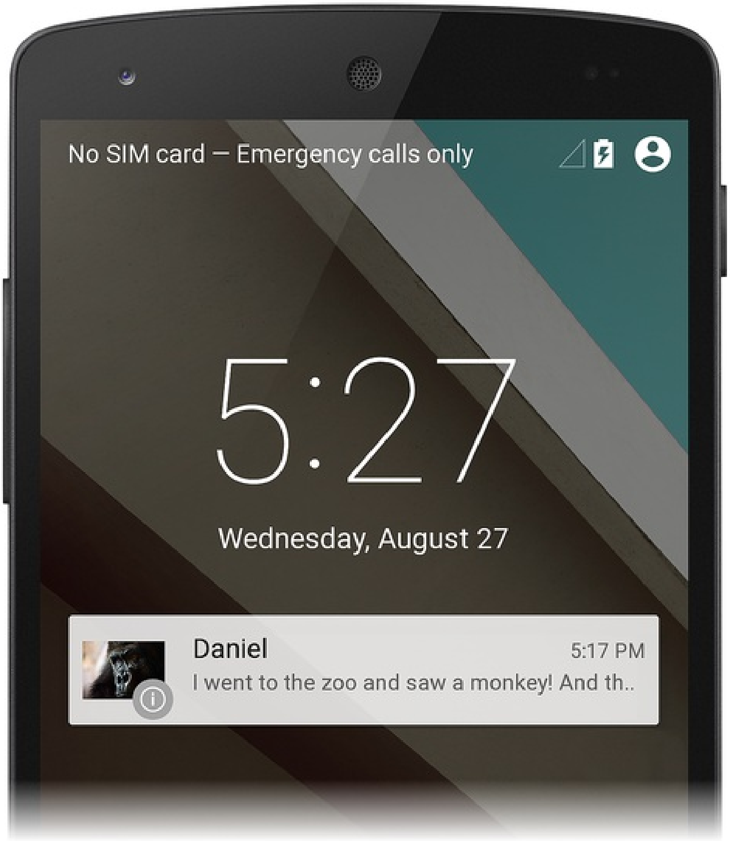
For more about `CardView`, see CardView.

Enhanced Notifications

The notification system in Android 5.0 has been significantly updated with a new visual format and new features. Notifications have a new look in Android 5.0. For example, notifications in Android 5.0 now use dark text over a light background:

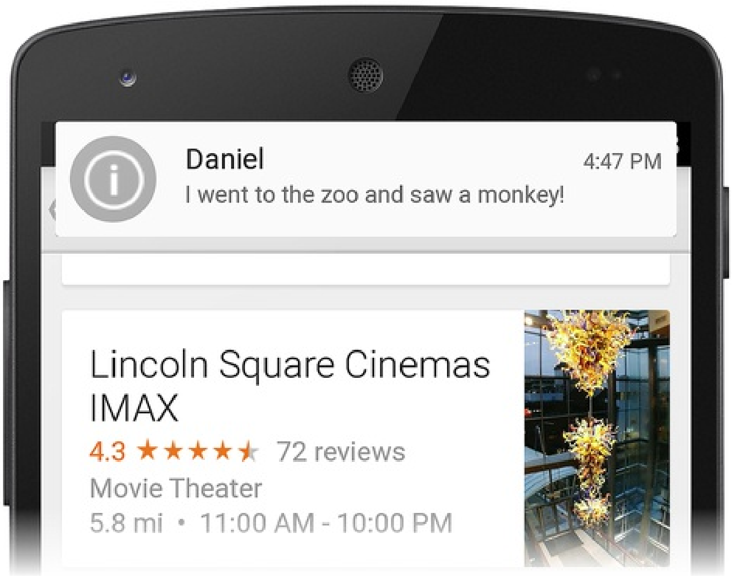
When a large icon is displayed in a notification (as shown in the above example), Android 5.0 presents the small icon as a badge over the large icon.

In Android 5.0, notifications can also appear on the device lockscreen. For example, here is an example screenshot of a lockscreen with a single notification:



Users can double-tap a notification on the lockscreen to unlock the device and jump to the app that originated that notification, or swipe to dismiss the notification. Notifications have a new *visibility* setting that determines how much content can be displayed on the lockscreen. Users can choose whether to allow sensitive content to be shown in lockscreen notifications.

Android 5.0 introduces a new high-priority notification presentation format called *Heads-up*. Heads-up notifications slide down from the top of the screen for a few seconds and then retreat back to the notification shade at the top of the screen. Heads-up notifications make it possible for the system UI to put important information in front of the user without disrupting the currently running activity. The following example illustrates a simple Heads-up notification that displays on top of an app:



Heads-up notifications are typically used for the following events:

- A new next message
- An incoming phone call
- Low battery indication
- An alarm

Android 5.0 displays a notification in Heads-up format only when it has a high or max priority setting.

In Android 5.0, you can provide notification metadata to help Android sort and display notifications more intelligently. Android 5.0 organizes notifications according to priority, visibility, and category. Notification categories are used to filter which notifications can be presented when the device is in *Do not disturb* mode.

For detailed information about creating and launching notifications with the latest Android 5.0 features, see Local Notifications.

New APIs

In addition to the new look-and-feel features described above, Android 5.0 adds new APIs that extend the capabilities of existing multimedia, storage, and wireless/connectivity functionality. Also, Android 5.0 includes new APIs that provide support for a new job scheduler feature.

Camera

Android 5.0 provides several new APIs for enhanced camera capabilities. The new `Android.Hardware.Camera2` namespace includes functionality for accessing individual camera devices connected to an Android device. Also, `Android.Hardware.Camera2` models each camera device as a pipeline: it accepts a capture request, captures the image, and then outputs the result. This approach makes it possible for apps to queue multiple capture requests to a camera device.

The following APIs make these new features possible:

- `CameraManager.GetCameraIdList` – Helps you to programmatically access camera devices; you use `CameraManager.OpenCamera` to connect to a specific camera device.
- `CameraCaptureSession` – Captures or streams images from the camera device. You implement a `CameraCaptureSession.CaptureListener` interface to handle new image capture events.
- `CaptureRequest` – Defines capture parameters.
- `CaptureResult` – Provides the results of an image capture operation.

For more about the new camera APIs in Android 5.0, see Media.

Audio Playback

Android 5.0 updates the `AudioTrack` class for better audio playback:

- `ENCODING_PCM_FLOAT` – Configures `AudioTrack` to accept audio data in floating-point format for better dynamic range, greater headroom, and higher quality (thanks to increased precision). Also, floating-point format helps to avoid audio clipping.
- `ByteBuffer` – You can now supply audio data to `AudioTrack` as a byte array.
- `WRITE_NON_BLOCKING` – This option simplifies buffering and multithreading for some apps.

For more about `AudioTrack` improvements in Android 5.0, see Media.

Media Playback Control

Android 5.0 introduces the new `Android.Media.MediaController` class, which replaces `RemoteControlClient`.

`Android.Media.MediaController` provides simplified transport control APIs and offers thread-safe control of playback outside of the UI context. The following new APIs handle transport control:

- `Android.Media.Session.MediaSession` – A media control session that handles multiple controllers. You call `MediaSession.GetSessionToken` to request a token that your app uses to interact with the session.
- `MediaController.TransportControls` – Handles transport commands such as **Play**, **Stop**, and **Skip**.

Also, you can use the new `Android.App.Notification.MediaStyle` class to associate a media session with rich notification content (such as extracting and showing album art).

For more about the new media playback control features in Android 5.0, see Media.

Storage

Android 5.0 updates the Storage Access Framework to make it easier for applications to work with directories and documents:

- To select a directory subtree, you can build and send an `Android.Intent.Action.OPEN_DOCUMENT_TREE` intent. This intent causes the system to display all provider instances that support subtree selection; the user then browses and selects a directory.
- To create and manage new documents or directories anywhere under a subtree, you use the new `CreateDocument`, `RenameDocument`, and `DeleteDocument` methods of `DocumentsContract`.
- To get paths to media directories on all shared storage devices, you call the new `Android.Content.Context.GetExternalMediaDirs` method.

For more about new storage APIs in Android 5.0, see Storage.

Wireless & Connectivity

Android 5.0 adds the following API enhancements for wireless and connectivity:

- New *multi-network* APIs that make it possible for apps to find and select networks with specific capabilities before making a connection.
- Bluetooth broadcasting functionality that enables an Android 5.0 device to act as a low-energy Bluetooth peripheral.
- NFC enhancements that make it easier to use near-field communications functionality for sharing data with other devices.

For more about the new wireless and connectivity APIs in Android 5.0, see Wireless and Connectivity.

Job Scheduling

Android 5.0 introduces a new `JobScheduler` API that can help users minimize battery drain by scheduling certain tasks to run only when the device is plugged in and charging. This job scheduler feature can also be used for scheduling a task to run when conditions are more suitable to that task, such as downloading a large file when the device is connected over a Wi-Fi network instead of a metered network.

For more about the new job scheduling APIs in Android 5.0, see Scheduling Jobs.

Summary

This article provided an overview of important new features in Android 5.0 for Xamarin.Android app developers:

- Material Theme
- Animations

- View shadows and elevation
- Color features, such as drawable tinting and prominent color extraction
- The new `RecyclerView` and `CardView` widgets
- Notification enhancements
- New APIs for camera, audio playback, media control, storage, wireless/connectivity, and job scheduling

If you are new to Xamarin Android development, read Setup and Installation to help you get started with Xamarin.Android. Hello, Android is an excellent introduction for learning how to create Android projects.

Related Links

- Android L Developer Preview
- Get the Android SDK
- Material Design

KitKat Features

7/8/2021 • 21 minutes to read • Edit Online

Android 4.4 (KitKat) comes loaded with a cornucopia of features for users and developers. This guide highlights several of these features and provides code examples and implementation details to help you make the most out of KitKat.

Overview

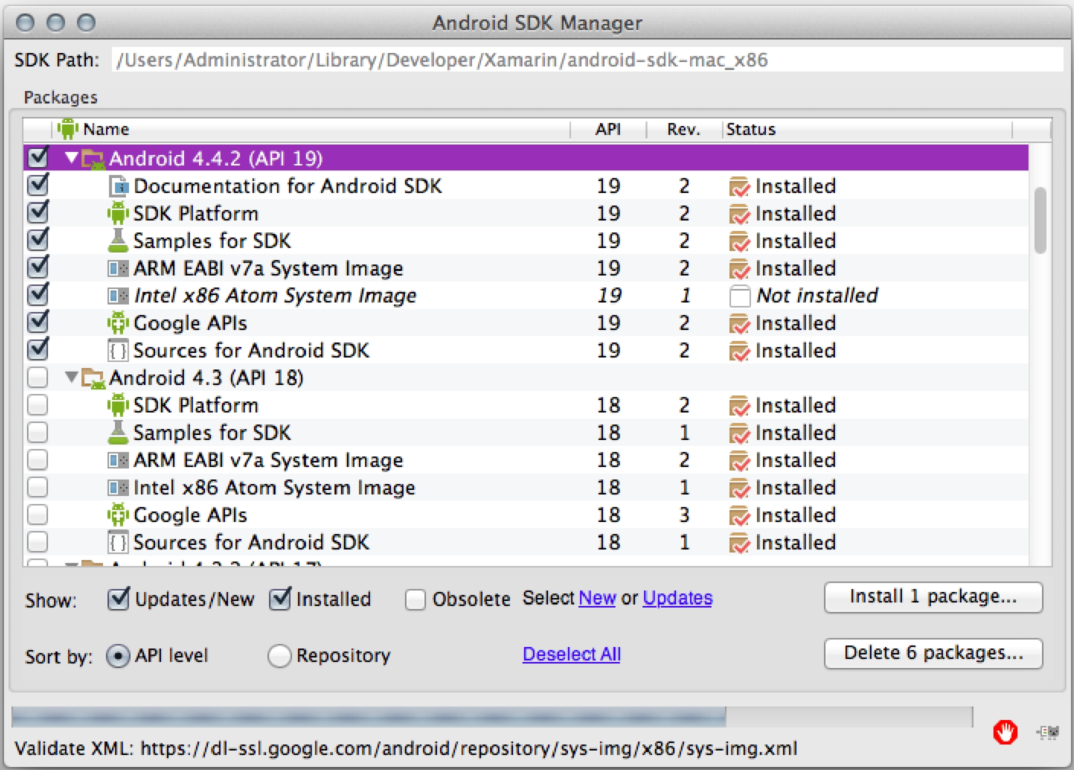
Android 4.4 (API Level 19), also known as "KitKat", was released in late 2013. KitKat offers a variety of new features and improvements, including:

- User Experience – Easy animations with transition framework, translucent status and navigation bars, and full-screen immersive mode help create a better experience for the user.
- User Content – User file management simplified with storage access framework; printing pictures, web sites, and other content is easier with improved printing APIs.
- Hardware – Turn any app into an NFC card with NFC Host-Based Card Emulation; run low-power sensors with the `SensorManager` .
- Developer Tools – Screencast applications in action with the Android Debug Bridge client, available as part of the Android SDK.

This guide provides guidance for migrating an existing Xamarin.Android application to KitKat, as well as a high-level overview of KitKat for Xamarin.Android developers.

Requirements

To develop Xamarin.Android applications using KitKat, you need *Xamarin.Android 4.11.0* or higher and Android 4.4 (API Level 19) installed via the Android SDK Manager, as illustrated by the following screenshot:



Migrating Your App to KitKat

This section provides some first-response items to help transition existing applications to Android 4.4.

Check System Version

If an application needs to be compatible with older versions of Android, be sure to wrap any KitKat-specific code in a system version check, as illustrated by the code sample below:

```
if (Build.VERSION.SdkInt >= BuildVersionCodes.Kitkat) {
    //KitKat only code here
}
```

Alarm Batching

Android uses alarm services to wake an app in the background at a specified time. KitKat takes this a step further by batching alarms to preserve power. This means that, in lieu of waking each app at an exact time, KitKat prefers to group several applications that are registered to wake during the same time interval, and wake them at the same time. To tell Android to wake an app during a specified time interval, call `SetWindow` on the `AlarmManager`, passing in the minimum and maximum time, in milliseconds, that can elapse before the app is woken, and the operation to perform at wakeup. The following code provides an example of an application that needs to be woken between a half hour and an hour from the time the window is set:

```
AlarmManager alarmManager = (AlarmManager)GetSystemService(AlarmService);
alarmManager.SetWindow (AlarmType.Rtc, AlarmManager.IntervalHalfHour, AlarmManager.IntervalHour,
pendingIntent);
```

To continue waking an app at an exact time, use `SetExact`, passing in the exact time that the app should be woken, and the operation to perform:

```
alarmManager.SetExact (AlarmType.Rtc, AlarmManager.IntervalDay, pendingIntent);
```

KitKat no longer lets you set an exact repeating alarm. Applications that use `SetRepeating` and require exact alarms to work will now need to trigger each alarm manually.

External Storage

External storage is now divided into two types - storage unique to your application, and data shared by multiple applications. Reading and writing to your app's specific location on external storage requires no special permissions. Interacting with data on shared storage now requires the `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE` permission. The two types can be classified as such:

- If you're getting a file or directory path by calling a method on `Context` - for example, `GetExternalFilesDir` or `GetExternalCacheDirs`
  - your app requires no extra permissions.
- If you're getting a file or directory path by accessing a property or calling a method on `Environment` , such as `GetExternalStorageDirectory` or `GetExternalStoragePublicDirectory` , your app requires the `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE` permission.

> **NOTE**
> `WRITE_EXTERNAL_STORAGE` implies the `READ_EXTERNAL_STORAGE` permission, so you should only ever need to set one permission.

SMS Consolidation

KitKat simplifies messaging for the user by aggregating all SMS content in one default application selected by the user. The developer is responsible for making the app selectable as the default messaging application, and behaving appropriately in code and in life if the application is not selected. For more information on transitioning your SMS app to KitKat, refer to the Getting Your SMS Apps Ready for KitKat guide from Google.

WebView Apps

WebView got a makeover in KitKat. The biggest change is added security for loading content into a `WebView`. While most applications targeting older API versions should work as expected, testing applications that use the `WebView` class is highly recommended. For more information about affected WebView APIs refer to the Android Migrating to WebView in Android 4.4 documentation.

User Experience

KitKat comes with several new APIs to enhance user experience, including the new transition framework for handling property animations and a translucent UI option for theming. These changes are covered below.

Transition Framework

The transition framework makes animations easier to implement. KitKat lets you perform a simple property animation with just one line of code, or customize transitions using *Scenes*.

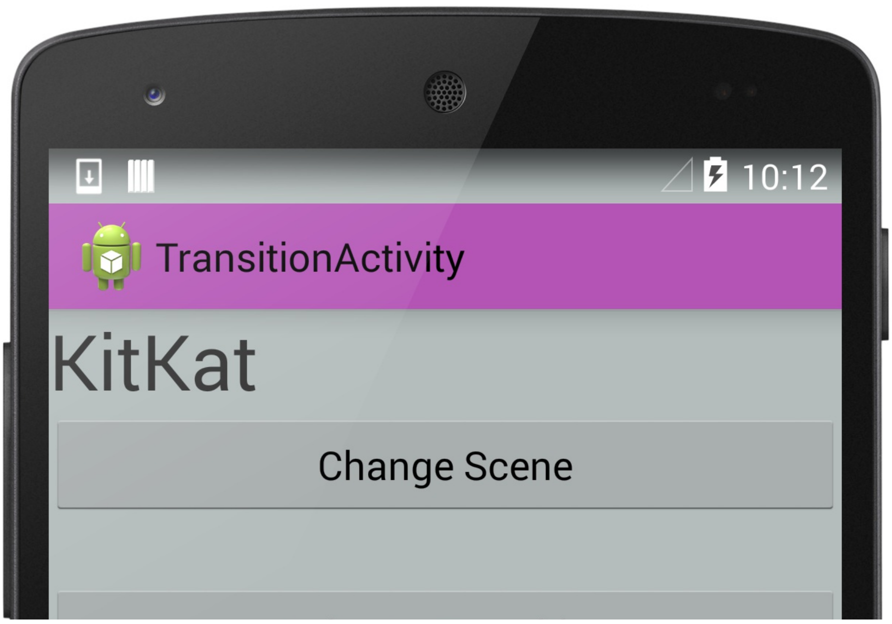
Simple Property Animation

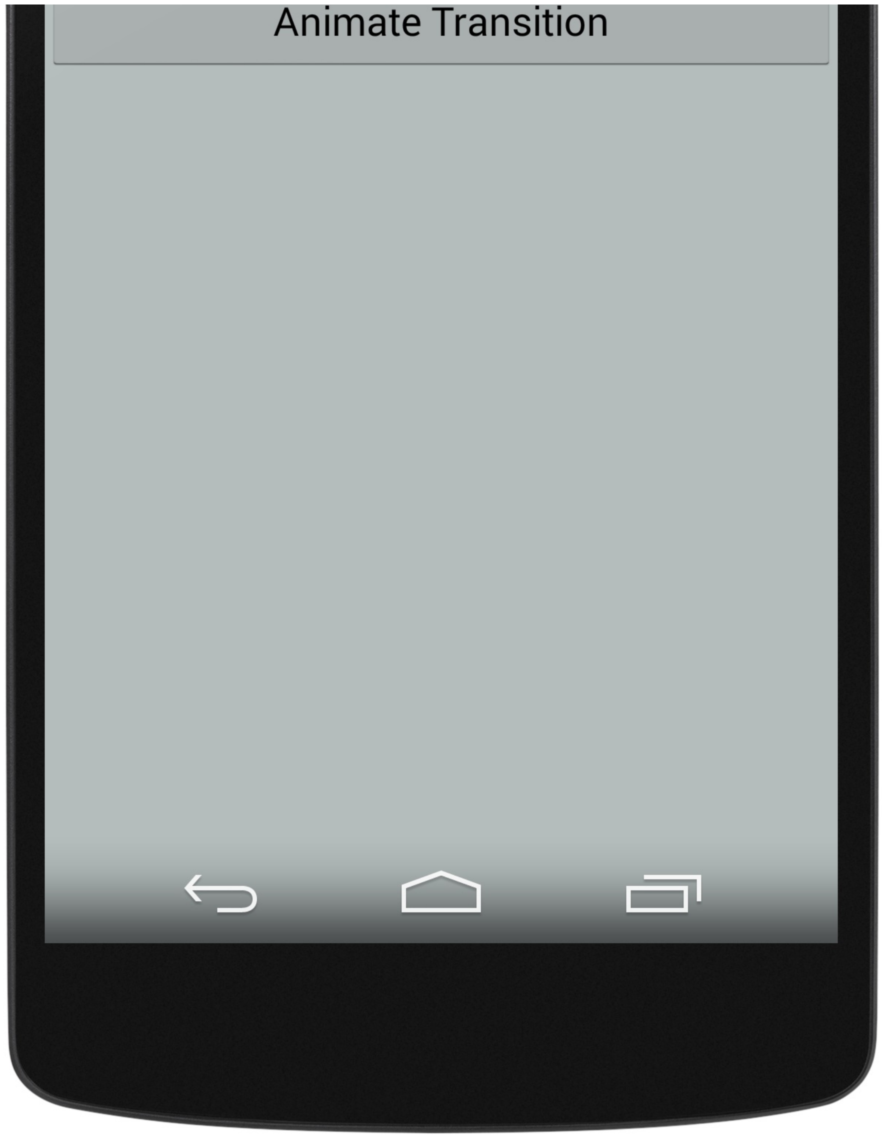
The new Android Transitions library simplifies the code behind property animations. The framework allows you to perform simple animations with minimal code. For example, the following code sample uses `TransitionManager.BeginDelayedTransition` to animate showing and hiding a `TextView`:

```
using Android.Transitions;

public class MainActivity : Activity
{
    LinearLayout linear;
    Button button;
    TextView text;

    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);
        SetContentView (Resource.Layout.Main);

        linear = FindViewById<LinearLayout> (Resource.Id.linearLayout);
        button = FindViewById<Button> (Resource.Id.button);
        text = FindViewById<TextView> (Resource.Id.textView);

        button.Click += (o, e) => {

            TransitionManager.BeginDelayedTransition (linear);

            if(text.Visibility != ViewStates.Visible)
            {
                text.Visibility = ViewStates.Visible;
            }
            else
            {
                text.Visibility = ViewStates.Invisible;
            }
        };
    }
}
```

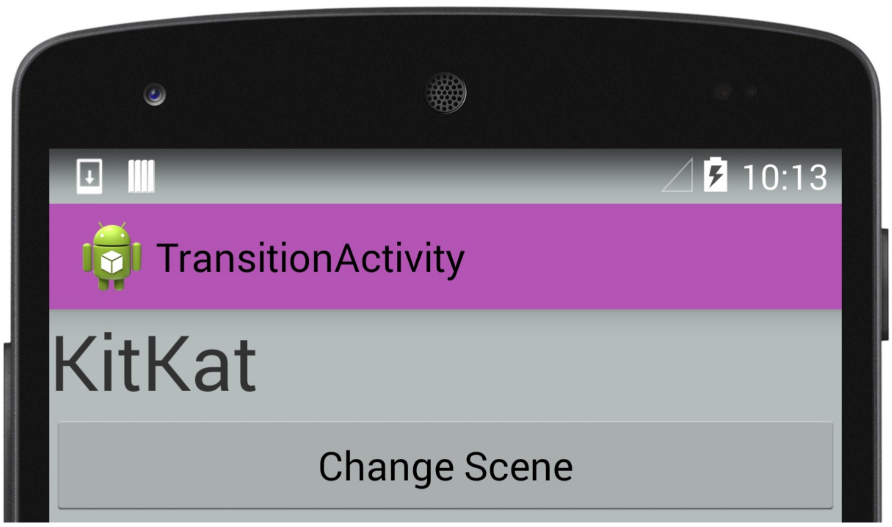
The example above uses the transition framework to create an automatic, default transition between the changing property values. Because the animation is handled by a single line of code, you can easily make this compatible with older versions of Android by wrapping the `BeginDelayedTransition` call in a system version check. See the Migrating Your App To KitKat section for more.

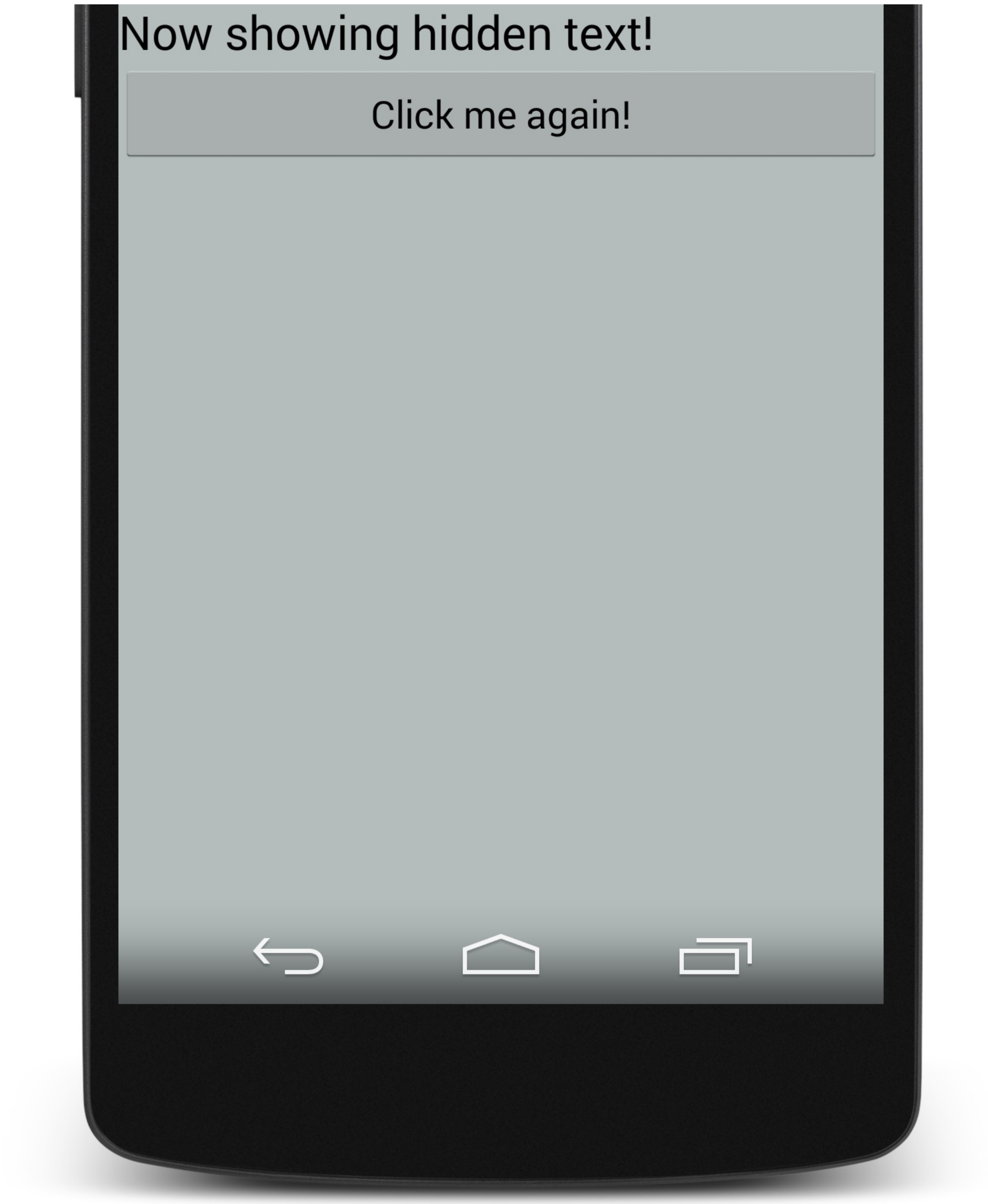
The screenshot below shows the app before the animation:





The screenshot below shows the app after the animation:



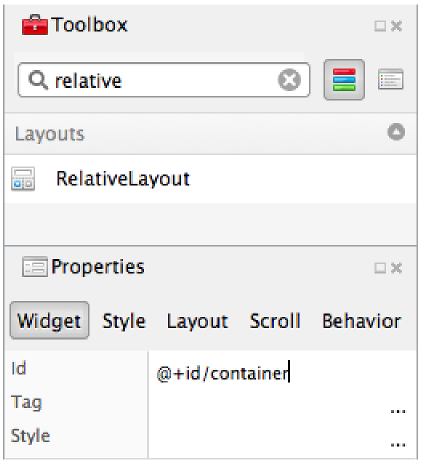


You can get more control over the transition with Scenes, which are covered in the next section.

Android Scenes

Scenes were introduced as part of the transition framework to give the developer more control over animations. Scenes create a dynamic area in the UI: you specify a container and several versions, or "scenes", for the XML content inside the container, and Android does the rest of the work to animate the transitions between the scenes. Android Scenes let you build complex animations with minimal work on the development side.

The static UI element housing the dynamic content is a called a *container* or *scene base*. The example below uses the Android Designer to create a `RelativeLayout` called `container`:



The sample layout also defines a button called `sceneButton` below the `container`. This button will trigger the transition.

The dynamic content inside the container requires two new Android layouts. These layouts specify only the code *inside* the container. The example code below defines a layout called *Scene1* that contains two text fields reading "Kit" and "Kat" respectively, and a second layout called *Scene2* that contains the same text fields reversed. The XML is as follows:

Scene1.axml:

```
<?xml version="1.0" encoding="utf-8"?>
<merge xmlns:android="http://schemas.android.com/apk/res/android">
    <TextView
        android:id="@+id/textA"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Kit"
        android:textSize="35sp" />
    <TextView
        android:id="@+id/textB"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toRightOf="@id/textA"
        android:text="Kat"
        android:textSize="35sp" />
</merge>
```

Scene2.axml:

```
<?xml version="1.0" encoding="utf-8"?>
<merge xmlns:android="http://schemas.android.com/apk/res/android">
    <TextView
        android:id="@+id/textB"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Kat"
        android:textSize="35sp" />
    <TextView
        android:id="@+id/textA"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toRightOf="@id/textB"
        android:text="Kit"
        android:textSize="35sp" />
</merge>
```

The example above uses `merge` to make the view code shorter and simplify the view hierarchy. You can read more about `merge` layouts here.

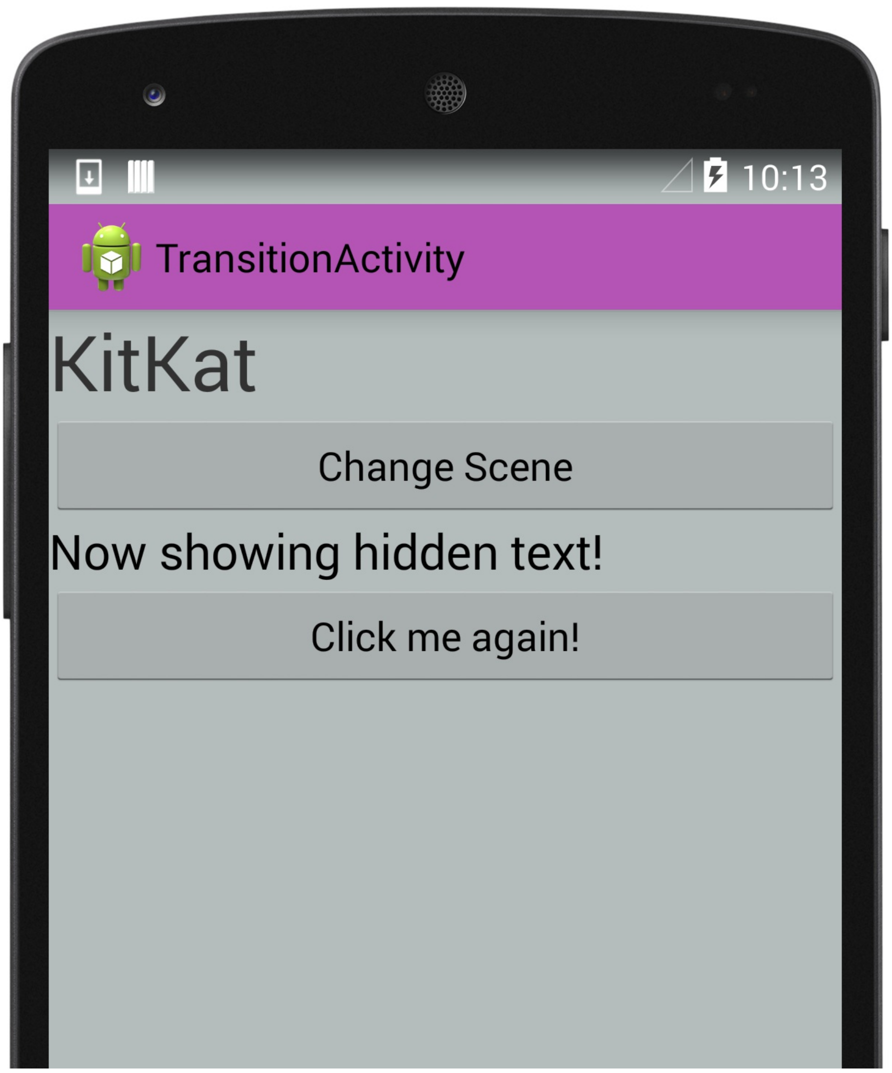
A Scene is created by calling `Scene.GetSceneForLayout`, passing in the container object, the Resource ID of the Scene's layout file, and the current `Context`, as illustrated by the code example below:

```
RelativeLayout container = FindViewById<RelativeLayout> (Resource.Id.container);

Scene scene1 = Scene.GetSceneForLayout(container, Resource.Layout.Scene1, this);
Scene scene2 = Scene.GetSceneForLayout(container, Resource.Layout.Scene2, this);

scene1.Enter();
```

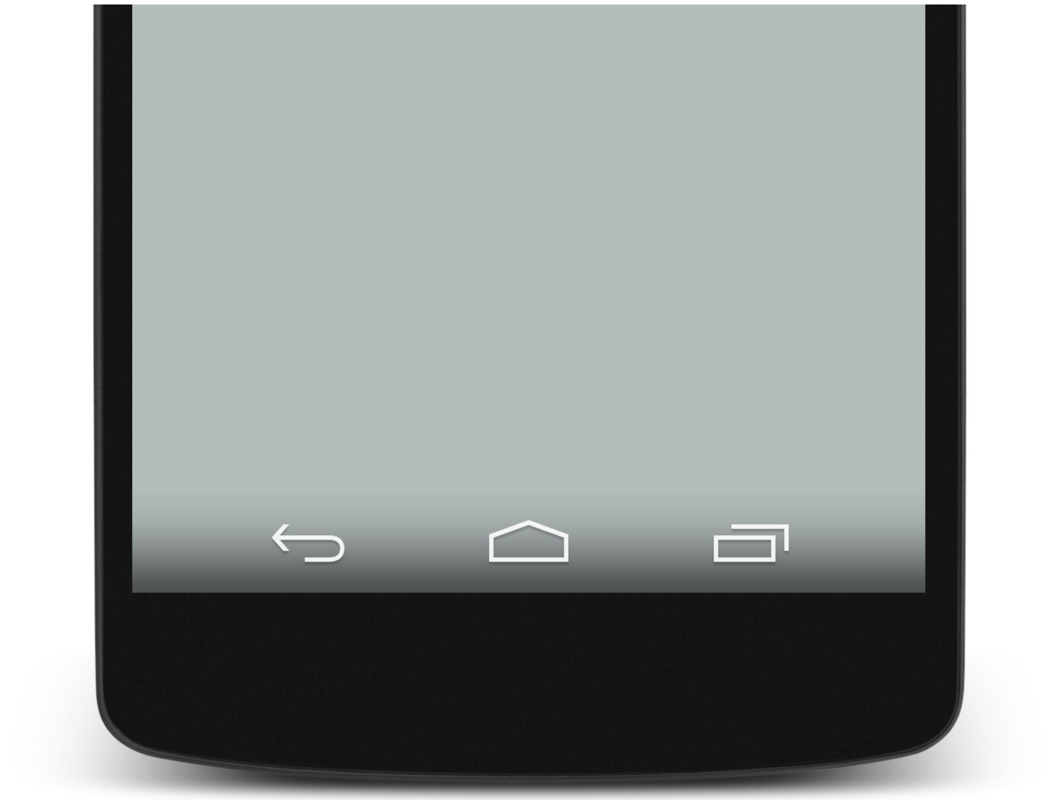
Clicking on the button flips between the two Scenes, which Android animates with the default transition values:

```
sceneButton.Click += (o, e) => {
    Scene temp = scene2;
    scene2 = scene1;
    scene1 = temp;

    TransitionManager.Go (scene1);
};
```

The screenshot below illustrates the scene before the animation:



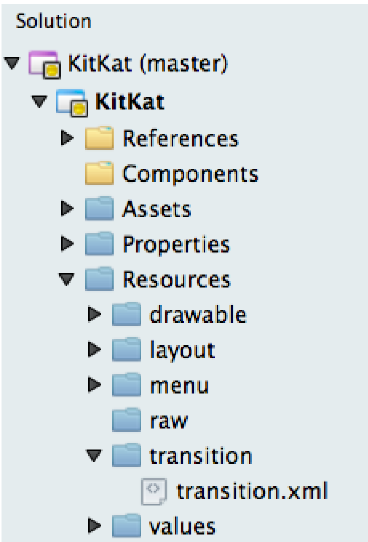
The screenshot below illustrates the scene after the animation:



> **NOTE**
>
> There is a known bug in the Android Transitions library that causes Scenes created using `GetSceneForLayout` to break when a user navigates through an Activity the second time. A java workaround is described here.

Custom Transitions in Scenes

A custom transition can be defined in an xml resource file in the `transition` directory under `Resources`, as illustrated by the screenshot below:



The following code sample defines a transition that animates for 5 seconds and uses the overshoot interpolator:

```
<changeBounds
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:duration="5000"
  android:interpolator="@android:anim/overshoot_interpolator" />
```

The transition is created in the Activity using the TransitionInflater, as illustrated by the code below:

```
Transition transition = TransitionInflater.From(this).InflateTransition(Resource.Transition.transition);
```

The new transition is then added to the `Go` call that begins the animation:

```
TransitionManager.Go (scene1, transition);
```

Translucent UI

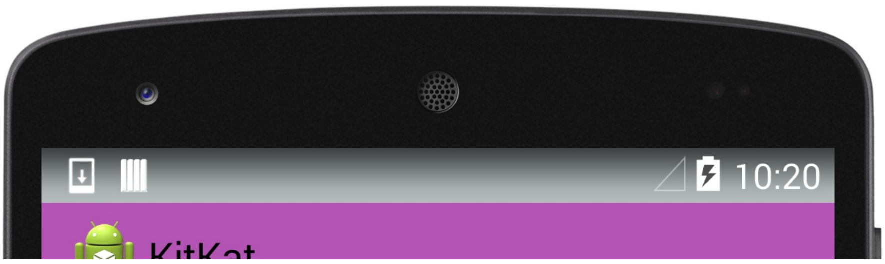
KitKat gives you more control over theming your app with optional translucent status and navigation bars. You can change the translucency of system UI elements in the same XML file you use to define your Android theme. KitKat introduces the following properties:

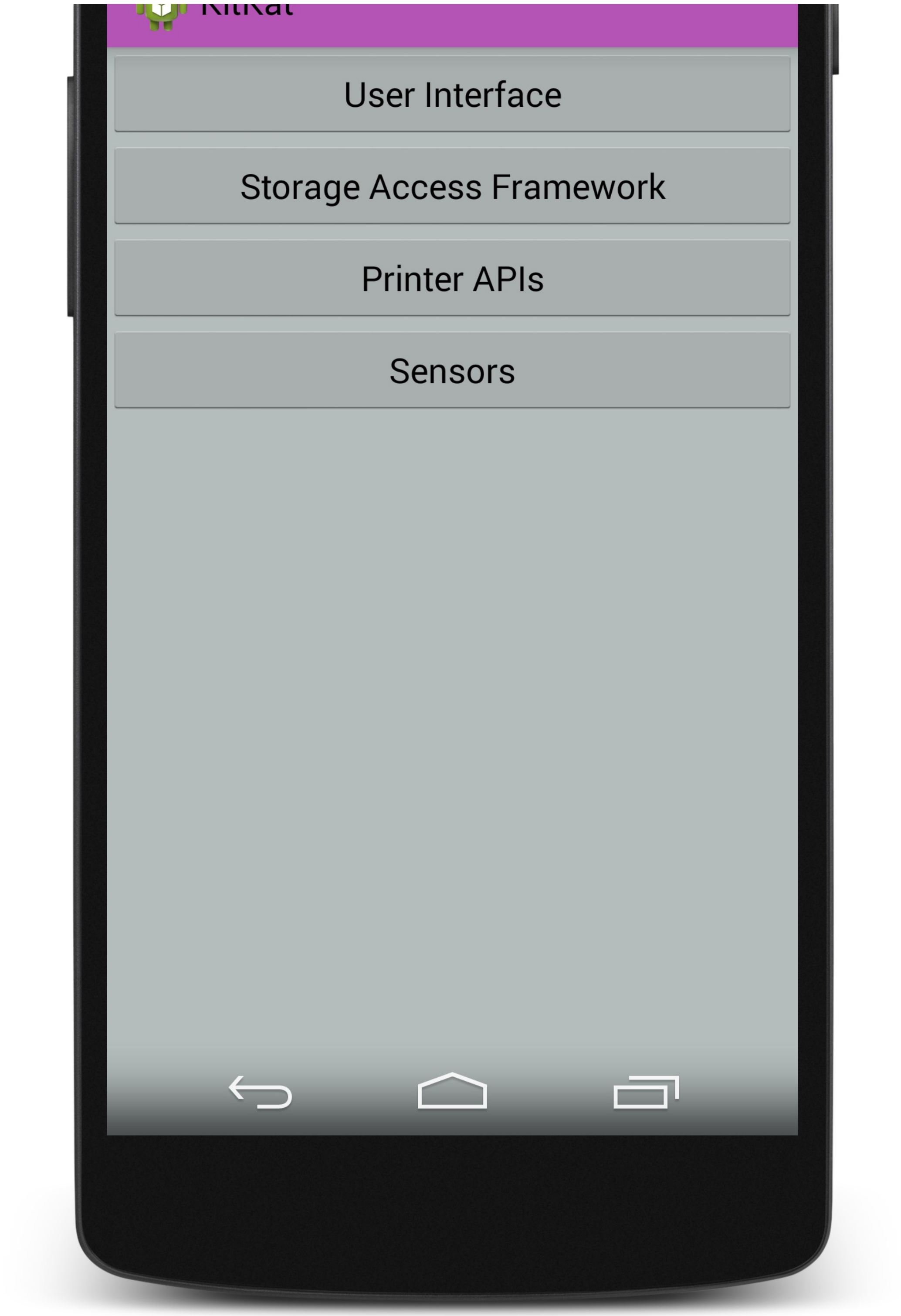
- `windowTranslucentStatus` - When set to true, makes the top status bar translucent.
- `windowTranslucentNavigation` - When set to true, makes the bottom navigation bar translucent.
- `fitsSystemWindows` - Setting the top or bottom bar to transcluent shifts content under the transparent UI elements by default. Setting this property to `true` is a simple way to prevent content from overlapping with the translucent system UI elements.

The following code defines a theme with translucent status and navigation bars:

```
<?xml version="1.0" encoding="UTF-8" ?>
<resources>
    <style name="KitKatTheme" parent="android:Theme.Holo.Light">
        <item name="android:windowBackground">@color/xamgray</item>
        <item name="android:windowTranslucentStatus">true</item>
        <item name="android:windowTranslucentNavigation">true</item>
        <item name="android:fitsSystemWindows">true</item>
        <item name="android:actionBarStyle">@style/ActionBar.Solid.KitKat</item>
    </style>

    <style name="ActionBar.Solid.KitKat" parent="@android:style/Widget.Holo.Light.ActionBar.Solid">
        <item name="android:background">@color/xampurple</item>
    </style>
</resources>
```

The screenshot below shows the theme above with translucent status and navigation bars:





User Content

Storage-Access Framework

The Storage Access Framework (SAF) is a new way for users to interact with stored content such as images, videos, and documents. Instead of presenting users with a dialog to choose an application to handle content, KitKat opens a new UI that allows users to access their data in one aggregate location. Once content has been chosen, the user will return to the application that requested the content, and the app experience will continue as normal.

This change requires two actions on the developer side: first, apps that require content from providers need to be updated to a new way of requesting content. Second, applications that write data to a `ContentProvider` need to be modified to use the new framework. Both scenarios depend on the new `DocumentsProvider` API.

DocumentsProvider

In KitKat, interactions with `ContentProviders` are abstracted with the `DocumentsProvider` class. This means that SAF doesn't care where the data is physically, as long as it is accessible through the `DocumentsProvider` API. Local providers, cloud services, and external storage devices all use the same interface, and are treated the same way, providing the user and the developer with one place to interact with the user's content.

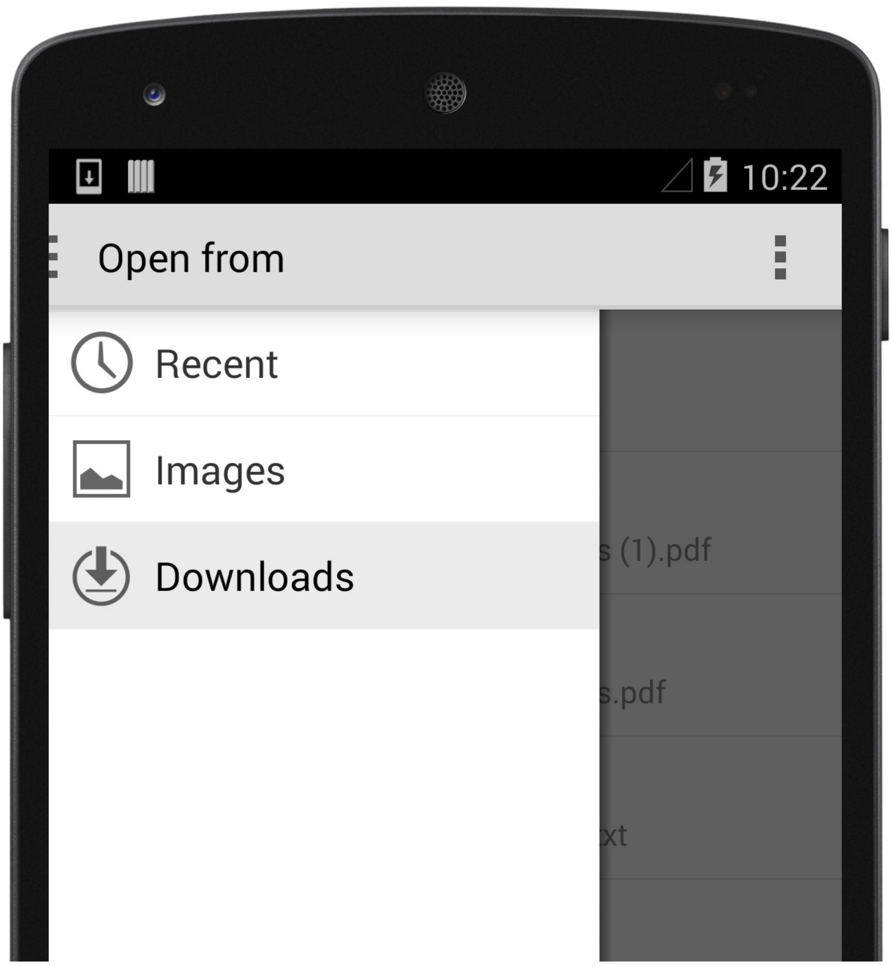
This section covers how to load and save content with the Storage Access Framework.

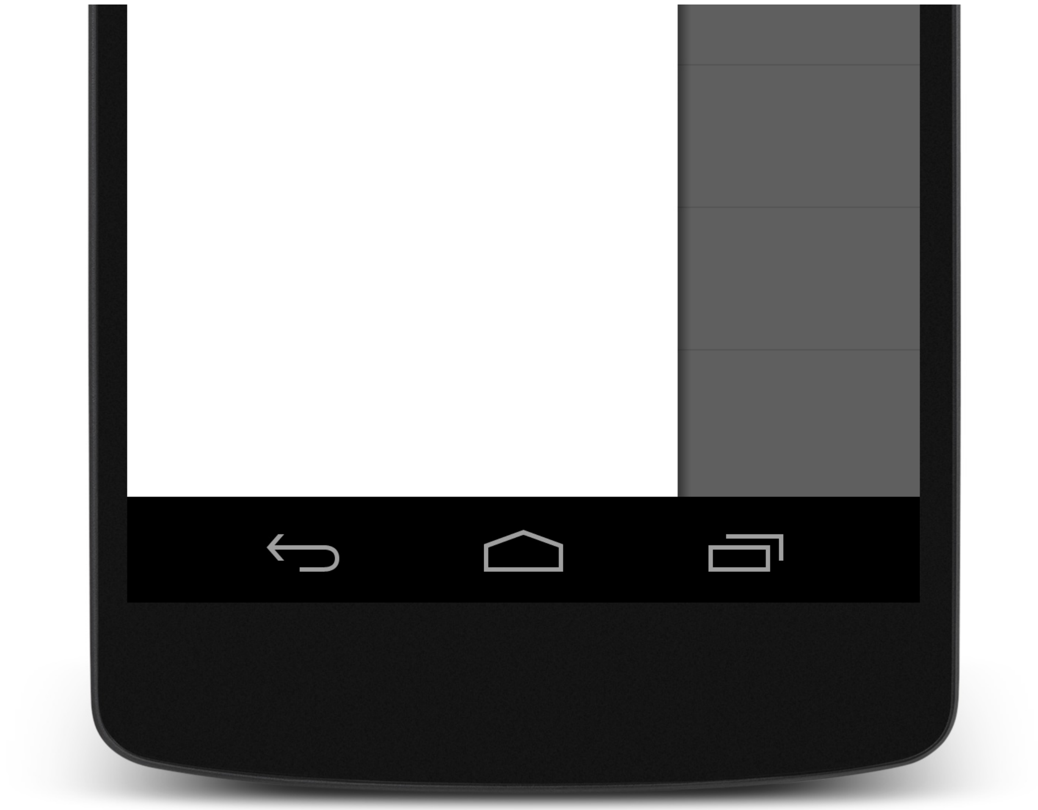
Request Content From a Provider

We can tell KitKat that we want to pick content using the SAF UI with the `ActionOpenDocument` Intent, which signifies that we want to connect to all content providers available to the device. You can add some filtering to this Intent by specifying `CategoryOpenable`, which means only content that can be opened (i.e. accessible, usable content) will be returned. KitKat also allows filtering of content with the `MimeType`. For example, the code below filters for image results by specifying the image `MimeType`:

```
Intent intent = new Intent (Intent.ActionOpenDocument);
intent.AddCategory (Intent.CategoryOpenable);
intent.SetType ("image/*");
StartActivityForResult (intent, save_request_code);
```

Calling `StartActivityForResult` launches the SAF UI, which the user can then browse to choose an image:





After the user has chosen an image, `OnActivityResult` returns the `Android.Net.Uri` of the chosen file. The code sample below displays the user's image selection:

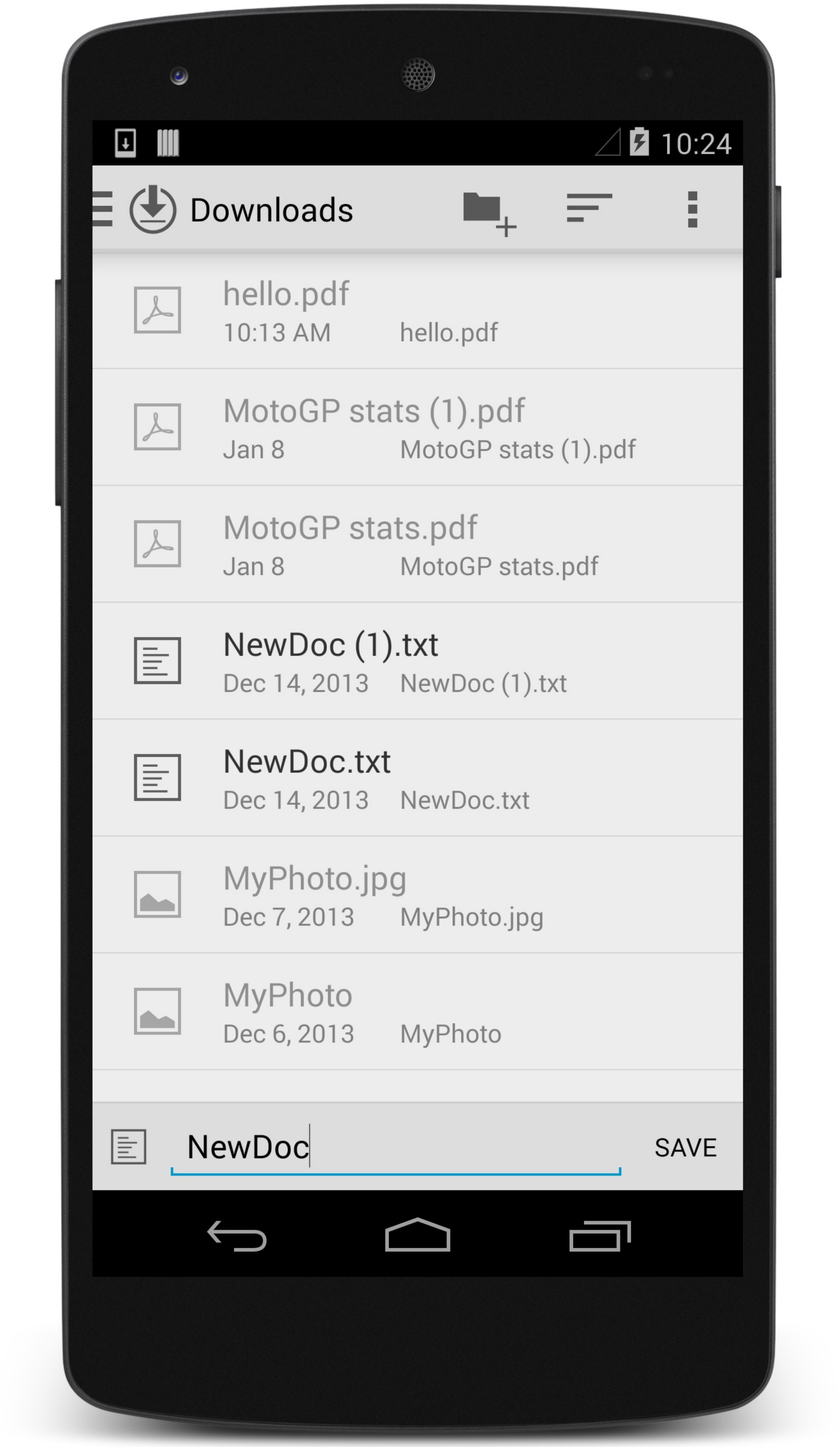
```
protected override void OnActivityResult(int requestCode, Result resultCode, Intent data)
{
    base.OnActivityResult(requestCode, resultCode, data);

    if (resultCode == Result.Ok && data != null && requestCode == save_request_code) {
        imageView = FindViewById<ImageView> (Resource.Id.imageView);
        imageView.SetImageURI (data.Data);
    }
}
```

Write Content To a Provider

In addition to loading content from the SAF UI, KitKat also lets you save content to any `ContentProvider` that implements the `DocumentProvider` API. Saving content uses an `Intent` with `ActionCreateDocument`:

```
Intent intentCreate = new Intent (Intent.ActionCreateDocument);
intentCreate.AddCategory (Intent.CategoryOpenable);
intentCreate.SetType ("text/plain");
intentCreate.PutExtra (Intent.ExtraTitle, "NewDoc");
StartActivityForResult (intentCreate, write_request_code);
```

The above code sample loads the SAF UI, letting the user change the file name and select a directory to house the new file:



When the user presses **Save**, `OnActivityResult` gets passed the `Android.Net.Uri` of the newly created file, which can be accessed with `data.Data`. The uri can be used to stream data into the new file:

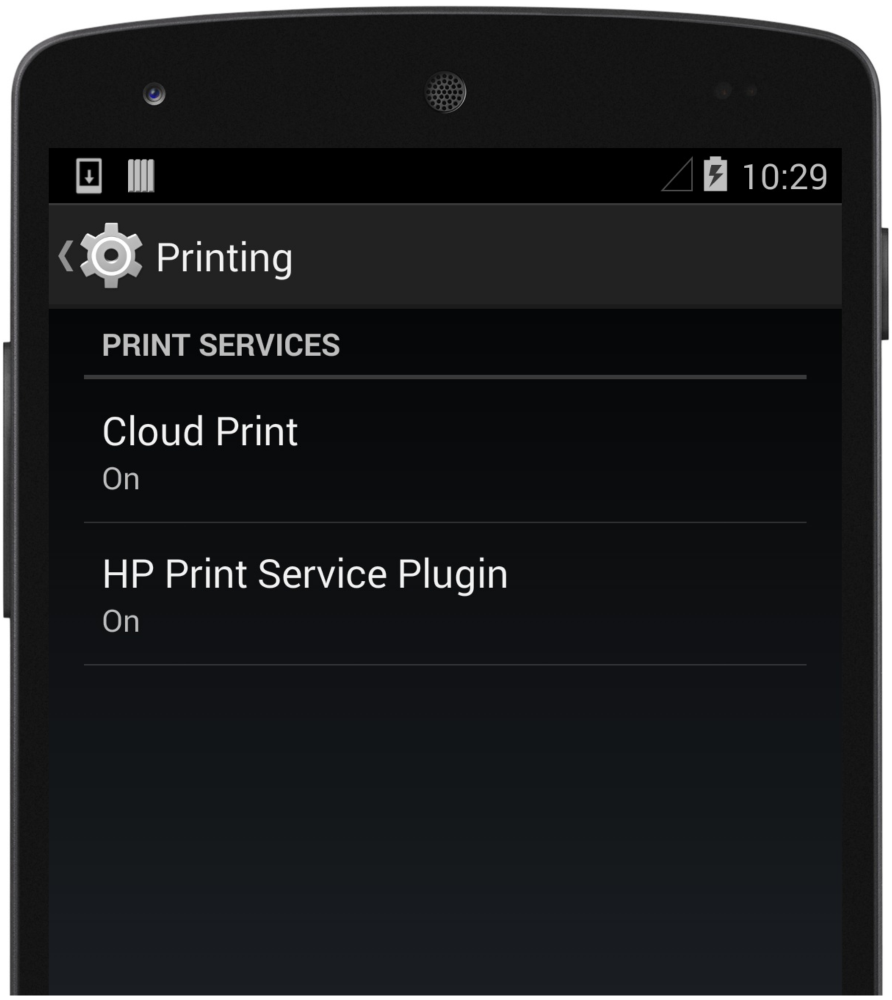
```
protected override void OnActivityResult(int requestCode, Result resultCode, Intent data)
{
    base.OnActivityResult(requestCode, resultCode, data);

    if (resultCode == Result.Ok && data != null && requestCode == write_request_code) {
        using (Stream stream = ContentResolver.OpenOutputStream(data.Data)) {
            Encoding u8 = Encoding.UTF8;
            string content = "Hello, world!";
            stream.Write (u8.GetBytes(content), 0, content.Length);
        }
    }
}
```

Note that `ContentResolver.OpenOutputStream(Android.Net.Uri)` returns a `System.IO.Stream`, so the entire streaming process can be written in .NET.

For more information on loading, creating, and editing content with the Storage Access Framework, refer to the Android documentation for the Storage Access Framework.

Printing

Printing content is simplified in KitKat with the introduction of the Print Services and `PrintManager`. KitKat is also the first API version to fully leverage the Google's Cloud Print service APIs using the Google Cloud Print application. Most devices that ship with KitKat automatically download Google Cloud Print app and the HP Print Service Pluginwhen they first connect to WiFi. A user can check his or her device's Print settings by navigating to **Settings > System > Printing**:

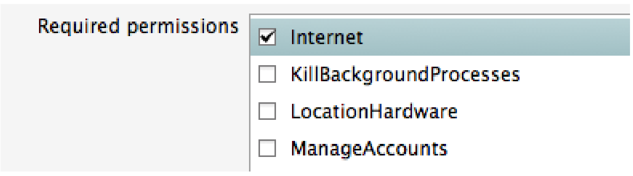


> **NOTE**
>
> Although the printing APIs are set up to work with Google Cloud Print by default, Android still lets developers prepare print content using the new APIs, and send it to other applications to handle printing.

Printing HTML Content

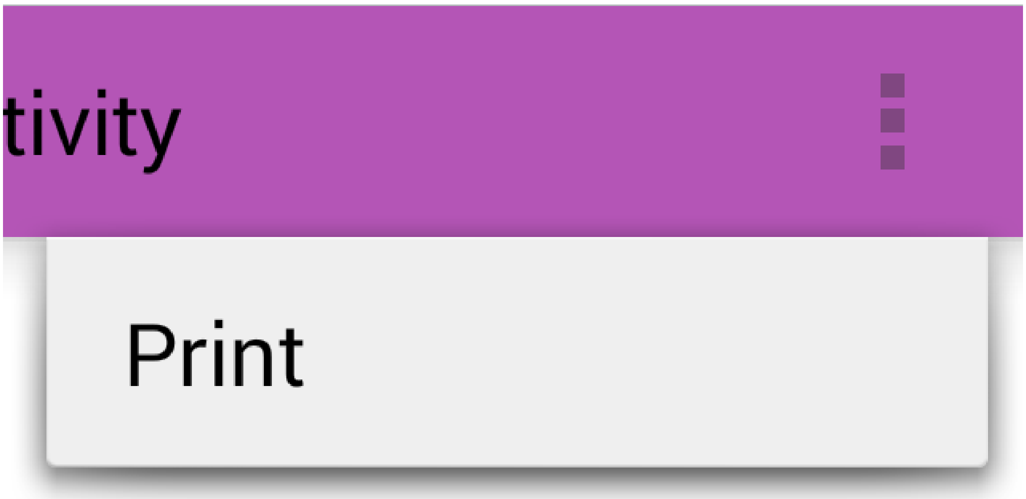
KitKat automatically creates a `PrintDocumentAdapter` for a web view with `WebView.CreatePrintDocumentAdapter`. Printing web content is a coordinated effort between a `WebViewClient` that waits for the HTML content to load and lets the Activity know to make the print option available in the options menu, and the Activity, which waits for the user to select the Print option and calls `Print` on the `PrintManager`. This section covers the basic setup required to print on-screen HTML content.

Note that loading and printing web content requires the Internet permission:



Print Menu Item

The print option will typically appear in the Activity's options menu. The options menu lets users perform actions on an Activity. It is in the top right corner of the screen, and looks like this:



Additional menu items can be defined in the *menu*directory under *Resources*. The code below defines a sample menu item called Print:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/menu_print"
        android:title="Print"
        android:showAsAction="never" />
</menu>
```

Interaction with the options menu in the Activity happens through the `OnCreateOptionsMenu` and `OnOptionsItemSelected` methods. `OnCreateOptionsMenu` is the place to add new menu items, like the Print option, from the *menu* resources directory. `OnOptionsItemSelected` listens for the user selecting the Print option from the menu, and begins printing:

```
bool dataLoaded;

public override bool OnCreateOptionsMenu (IMenu menu)
{
    base.OnCreateOptionsMenu (menu);
    if (dataLoaded) {
        MenuInflater.Inflate (Resource.Menu.print, menu);
    }
    return true;
}

public override bool OnOptionsItemSelected (IMenuItem item)
{
    if (item.ItemId == Resource.Id.menu_print) {
        PrintPage ();
        return true;
    }
    return base.OnOptionsItemSelected (item);
}
```

The code above also defines a variable called `dataLoaded` to keep track of the status of the HTML content. The `WebViewClient` will set this variable to true when all content has loaded, so the Activity knows to add the Print menu item to the options menu.

WebViewClient

The job of the `WebViewClient` is to ensure data in the `WebView` is fully loaded before the print option appears in the menu, which it does with the `OnPageFinished` method. `OnPageFinished` listens for web content to finish loading, and tells the Activity to recreate its options menu with `InvalidateOptionsMenu`:

```
class MyWebViewClient : WebViewClient
{
    PrintHtmlActivity caller;

    public MyWebViewClient (PrintHtmlActivity caller)
    {
        this.caller = caller;
    }

    public override void OnPageFinished (WebView view, string url)
    {
        caller.dataLoaded = true;
        caller.InvalidateOptionsMenu ();
    }
}
```

`OnPageFinished` also sets the `dataLoaded` value to `true`, so `OnCreateOptionsMenu` can recreate the menu with the Print option in place.

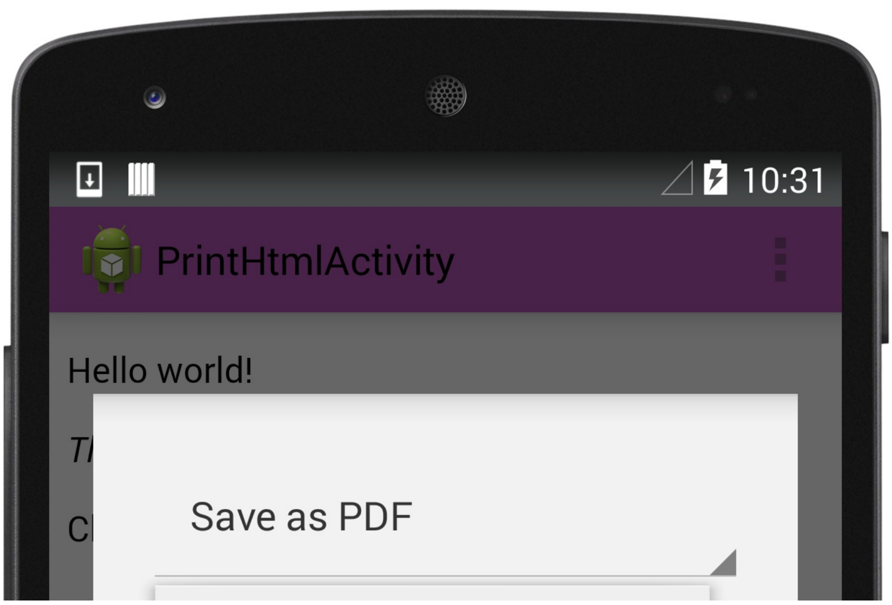
PrintManager

The following code example prints the contents of a `WebView`:

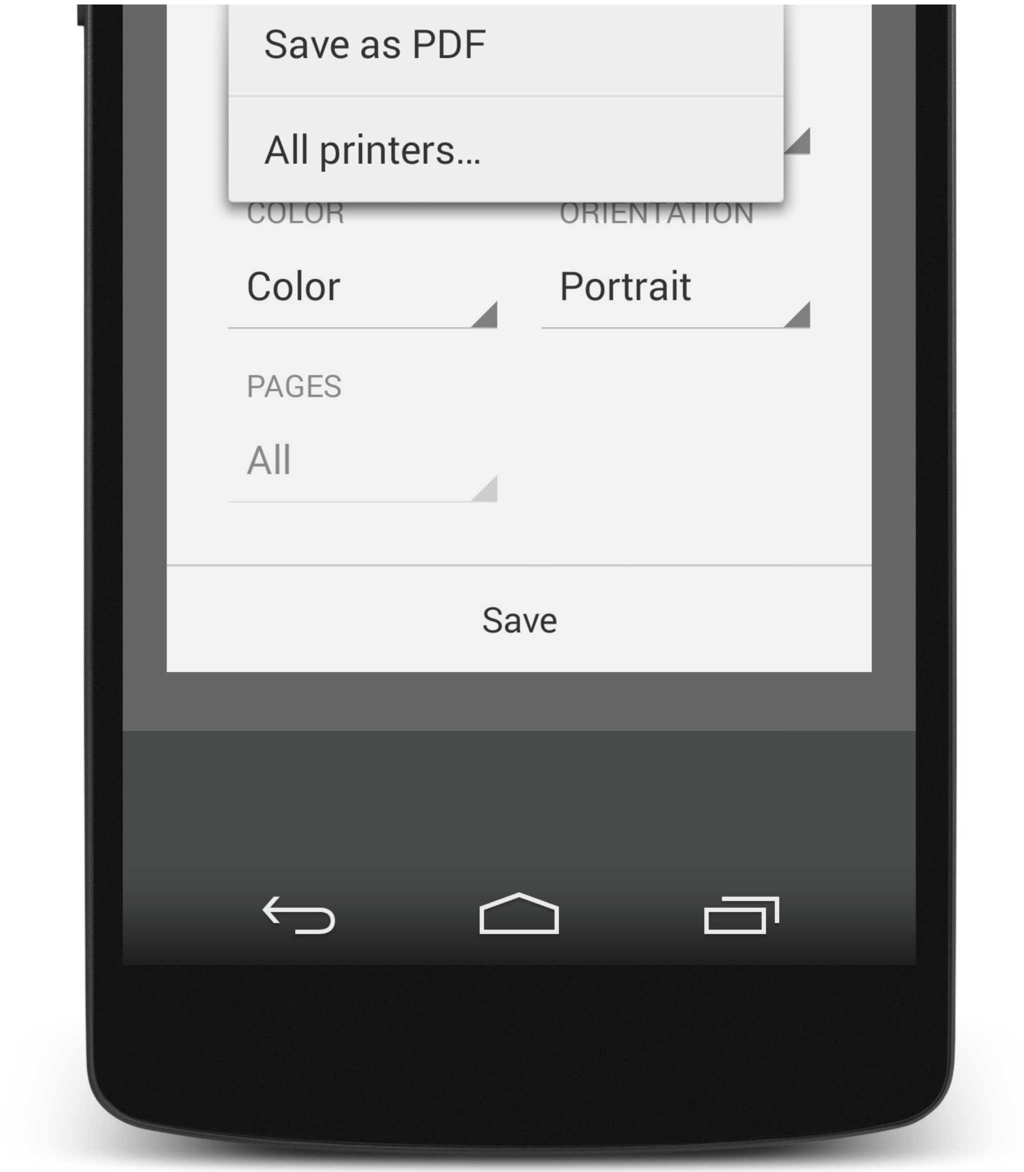
```
void PrintPage ()
{
    PrintManager printManager = (PrintManager)GetSystemService (Context.PrintService);
    PrintDocumentAdapter printDocumentAdapter = myWebView.CreatePrintDocumentAdapter ();
    printManager.Print ("MyWebPage", printDocumentAdapter, null);
}
```

`Print` takes as arguments: a name for the print job ("MyWebPage" in this example), a `PrintDocumentAdapter` that generates the print document from the content, and `PrintAttributes` (`null` in the example above). You can specify `PrintAttributes` to help lay out content on the printed page, although the default attributes should handle most scenarios.

Calling `Print` loads the print UI, which lists options for the print job. The UI gives users the option of printing or saving the HTML content to a PDF, as illustrated by the screenshots below:



10:31
PrintHtmlActivity
Hello world!
Save as PDF



Hardware

KitKat adds several APIs to accommodate new device features. The most notable of these are Host-Based Card Emulation and the new `SensorManager`.

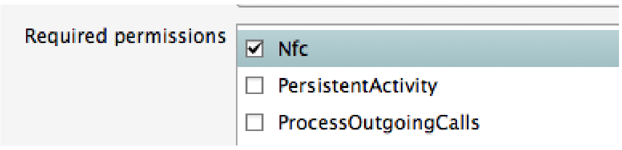
Host-Based Card Emulation in NFC

Host-Based Card Emulation (HCE) allows applications to behave like NFC cards or NFC card readers without relying on the carrier's proprietary Secure Element. Before setting up HCE, ensure HCE is available on the device with `PackageManager.HasSystemFeature`:

```
bool hceSupport = PackageManager.HasSystemFeature(PackageManager.FeatureNfcHostCardEmulation);
```

HCE requires that both the HCE feature and the `Nfc` permission be registered with the application's `AndroidManifest.xml`:

```
<uses-feature android:name="android.hardware.nfc.hce" />
```



To work, HCE has to be able to run in the background, and it has to start when the user makes an NFC transaction, even if the application using HCE is not running. We can accomplish this by writing the HCE code as a `Service`. An HCE Service implements the `HostApduService` interface, which implements the following methods:

- *ProcessCommandApdu* - An Application Protocol Data Unit (APDU) is what gets sent between the NFC Reader and the HCE Service. This method consumes an ADPU from the reader, and returns a data unit in response.
- *OnDeactivated* - The `HostAdpuService` is deactivated when the HCE Service is no longer communicating with the NFC Reader.

An HCE Service also needs to be registered with the application's manifest, and decorated with the proper permissions, intent filter, and metadata. The following code is an example of a `HostApduService` registered with the Android Manifest using the `Service` attribute (for more information on attributes, refer to the Xamarin Working with Android Manifest guide):

```
[Service(Exported=true, Permission="android.permissions.BIND_NFC_SERVICE"),
    IntentFilter(new[] {"android.nfc.cardemulation.HOST_APDU_SERVICE"}),
    MetaData("android.nfc.cardemulation.host.apdu_service",
    Resource="@xml/hceservice")]

class HceService : HostApduService
{
    public override byte[] ProcessCommandApdu(byte[] apdu, Bundle extras)
    {
        ...
    }

    public override void OnDeactivated (DeactivationReason reason)
    {
        ...
    }
}
```

The above Service provides a way for the NFC reader to interact with the application, but the NFC reader still has no way of knowing if this Service is emulating the NFC card it needs to scan. To help the NFC reader identify the Service, we can assign the Service a unique *Application ID (AID)*. We specify an AID, along with other metadata about the HCE Service, in an xml resource file registered with the `MetaData` attribute (see code example above). This resource file specifies one or more AID filters - unique identifier strings in hexadecimal format that correspond to the AIDs of one or more NFC reader devices:

```
<host-apdu-service xmlns:android="http://schemas.android.com/apk/res/android"
    android:description="@string/hce_service_description"
    android:requireDeviceUnlock="false"
    android:apduServiceBanner="@drawable/service_banner">
    <aid-group android:description="@string/aid_group_description"
                android:category="payment">
        <aid-filter android:name="1111111111111111"/>
        <aid-filter android:name="0123456789012345"/>
    </aid-group>
</host-apdu-service>
```

In addition to AID filters, the xml resource file also provides a user-facing description of the HCE Service, specifies an AID group (payment application versus "other") and, in the case of a payment application, a 260x96 dp banner to display to the user.

The setup outlined above provides the basic building blocks for an application emulating an NFC card. NFC itself requires several more steps and further testing to configure. For more information on Host-based Card Emulation, refer to the Android documentation portal. For more information on using NFC with Xamarin, check out the Xamarin NFC samples.

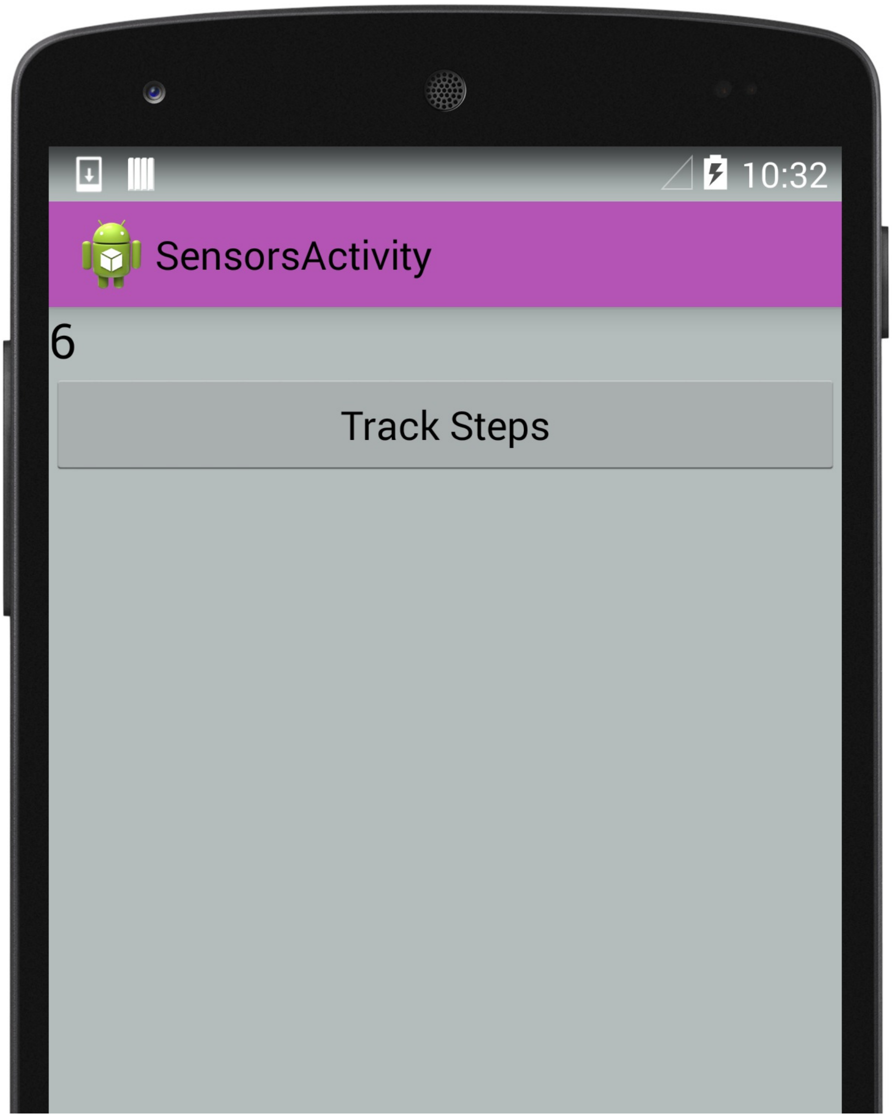
Sensors

KitKat provides access to the device's sensors through a `SensorManager`. The `SensorManager` allows the OS to schedule the delivery of sensor information to an application in batches, preserving battery life.

KitKat also ships with two new sensor types for tracking the user's steps. These are based on accelerometer and include:

- *StepDetector* - App is notified/woken when the user takes a step, and the detector provides a time value for when the step occurred.
- *StepCounter* - Keeps track of the number of steps the user has taken since the sensor was registered *until the next device reboot*.

The screenshot below depicts the step counter in action:



You can create a `SensorManager` by calling `GetSystemService(SensorService)` and casting the result as a `SensorManager`. To use the step counter, call `GetDefaultSensor` on the `SensorManager`. You can register the sensor and listen to changes in step count with the help of the `ISensorEventListener` interface, as illustrated by the code sample below:

```
public class MainActivity : Activity, ISensorEventListener
{
    float count = 0;

    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);
        SetContentView (Resource.Layout.Main);

        SensorManager senMgr = (SensorManager) GetSystemService (SensorService);
        Sensor counter = senMgr.GetDefaultSensor (SensorType.StepCounter);
        if (counter != null) {
            senMgr.RegisterListener(this, counter, SensorDelay.Normal);
        }
    }

    public void OnAccuracyChanged (Sensor sensor, SensorStatus accuracy)
    {
        Log.Info ("SensorManager", "Sensor accuracy changed");
    }

    public void OnSensorChanged (SensorEvent e)
    {
        count = e.Values [0];
    }
}
```

`OnSensorChanged` is called if the step count updates while the application is in the foreground. If the application enters the background, or the device is asleep, `OnSensorChanged` will not be called; however, the steps will continue to be counted until `UnregisterListener` is called.

Keep in mind that *the step count value is cumulative across all applications that register the sensor*. This means that even if you uninstall and reinstall your application, and initialize the `count` variable at 0 at application startup, the value reported by the sensor will remain the total number of steps taken while the sensor was registered, whether by your application or another. You can prevent your application from adding to the step counter by calling `UnregisterListener` on the `SensorManager`, as illustrated by the code below:

```
protected override void OnPause()
{
    base.OnPause ();
    senMgr.UnregisterListener(this);
}
```

Rebooting the device resets the step count to 0. Your app will require extra code to ensure it is reporting an accurate count for the application, regardless of other applications using the sensor or the state of the device.

NOTE

While the API for the step detection and counting ships with KitKat, not all phones are outfitted with the sensor. You can check if the sensor is available by running `PackageManager.HasSystemFeature(PackageManager.FeatureSensorStepCounter);`, or check to ensure the returned value of `GetDefaultSensor` isn't `null`.

Developer Tools

Screen Recording

KitKat includes new screen recording capabilities so that developers can record applications in action. Screen recording is available through the Android Debug Bridge (ADB) client, which can be downloaded as part of the Android SDK.

To record your screen, connect your device; then, locate your Android SDK installation, navigate to the **platform-tools** directory and run the **adb** client:

```
adb shell screenrecord /sdcard/screencast.mp4
```

The above command will record a default 3-minute video at the default resolution of 4Mbps. To edit the length, add the *--time-limit* flag. To change the resolution, add the *--bit-rate* flag. The following command will record a minute-long video at 8Mbps:

```
adb shell screenrecord --bit-rate 8000000 --time-limit 60 /sdcard/screencast.mp4
```

You can find your video on your device - it will appear in your Gallery when the recording is complete.

Other KitKat Additions

In addition to the changes described above, KitKat allows you to:

- *Use the Full Screen* - KitKat introduces a new Immersive mode for browsing content, playing games, and running other applications that could benefit from a full-screen experience.
- *Customize Notifications* - Get additional details about system notifications with the `NotificationListenerService`. This lets you present the information in a different way inside your app.
- *Mirror Drawable Resources* - Drawable resources have a new `autoMirrored` attribute that tells the system

create a mirrored version for images that require flipping for left-to-right layouts.

- *Pause Animations* - Pause and resume animations created with the `Animator` class.
- *Read Dynamically Changing Text* - Denote parts of UI that update dynamically with new text as "live regions" with the new `accessibilityLiveRegion` attribute so the new text will be read automatically in accessibility mode.
- *Enhance Audio Experience* - Make tracks louder with the `LoudnessEnhancer` , find the Peak and RMS of an audio stream with the `Visualizer` class, and get information from an audio timestamp to help with audio-video synchronization.
- *Sync ContentResolver at Custom Interval* - KitKat adds some variability to the time that a sync request is performed. Sync a `ContentResolver` at custom time or interval by calling `ContentResolver.RequestSync` and passing in a `SyncRequest` .
- *Distinguish Between Controllers* - In KitKat, controllers are assigned unique integer identifiers that can be accessed through the device's `ControllerNumber` property. This makes it easier to tell apart players in a game.
- *Remote Control* - With a few changes on both the hardware and software side, KitKat allows you to turn a device outfitted with an IR transmitter into a remote control using the `ConsumerIrService` , and interact with peripheral devices with the new `RemoteController` APIs.

For more information on the above API changes, please refer to the Google Android 4.4 APIs overview.

Summary

This article introduced some of the new APIs available in Android 4.4 (API Level 19), and covered best practices when transitioning an application to KitKat. It outlined changes to the APIs affecting user experience, including the *transition framework* and new options for *theming*. Next, it introduced the *Storage-Access Framework* and `DocumentsProvider` class, as well as the new *printing APIs*. It explored *NFC host-based card emulation* and how to work with *low-power sensors*, including two new sensors for tracking the user's steps. Finally, it demonstrated capturing real-time demos of applications with *screen recording*, and provided a detailed list of KitKat API changes and additions.

Related Links

- KitKat Sample
- Android 4.4 APIs
- Android KitKat

Jelly Bean Features

7/8/2021 • 11 minutes to read • Edit Online

This document will provide a high level overview of the new features for developers that were introduced in Android 4.1. These features include: enhanced notifications, updates to Android Beam to share large files, updates to multimedia, peer-to-peer network discovery, animations, new permissions.

Overview

Android 4.1 (API Level 16), also known as "Jelly Bean", was release on July 9th, 2012. This article will provide a high level introduction to some of the new features in Android 4.1 for developers using Xamarin.Android. Some of these new features introduced are enhancements to animations for launching an activity, new sounds for a camera, and improved support for application stack navigation. It is now possible to cut and paste with intents.

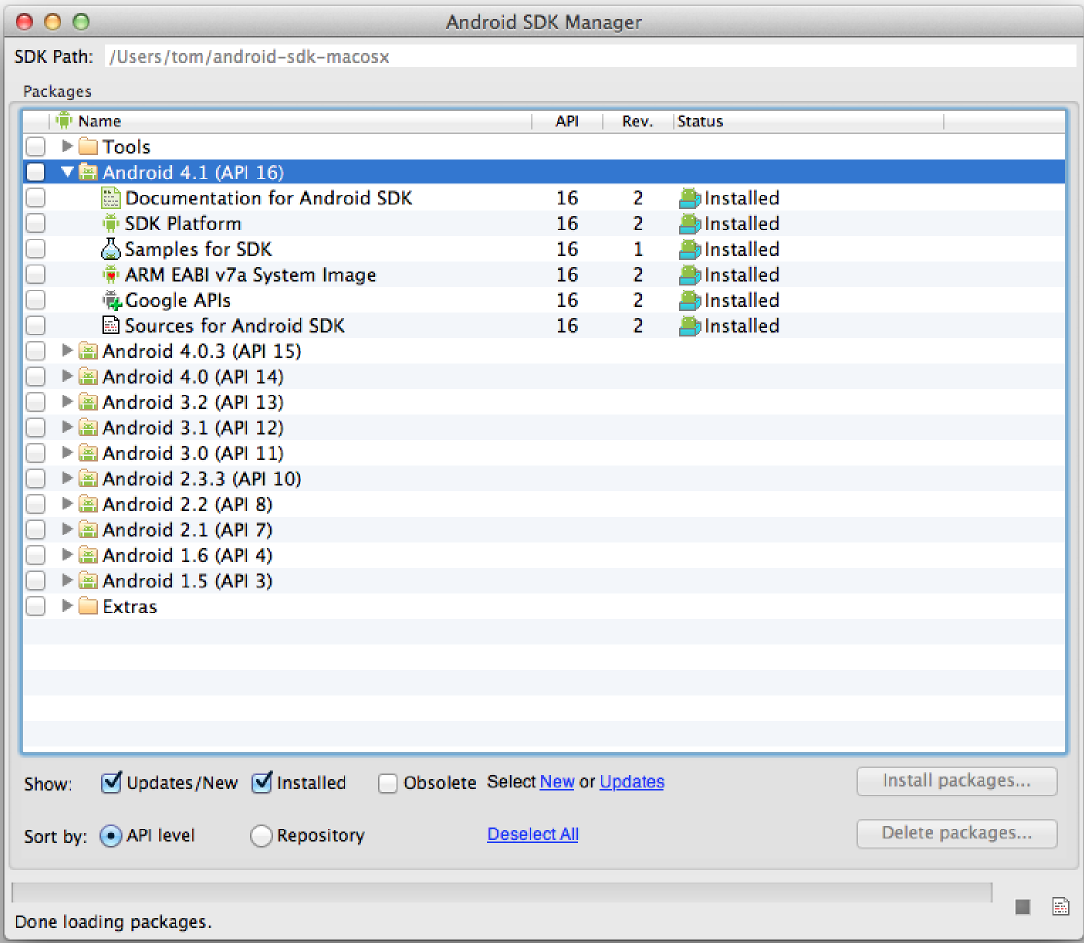
The stability of Android applications is improved with the ability to isolate the dependency on unstable content providers. Services may also be isolated so that they are accessible only by the activity that started them.

Support has been added for network service discovery using Bonjour, UPnP, or multicast DNS based services. It is now possible for richer notifications that have formatted text, action buttons and large images.

Finally several new permissions have been added in Android 4.1.

Requirements

To develop Xamarin.Android applications using Jelly Bean requires Xamarin.Android 4.2.6 or higher and Android 4.1 (API Level 16) be installed via the Android SDK Manager as shown in the following screen shot:



What's New

Animations

Activities may be launched using either zoom animations or custom animations by using the `ActivityOptions` class. The following new methods are provided to support these animations:

- `MakeScaleUpAnimation` – This will create an animation that scales up an activity window from a start position and size on the screen.
- `MakeThumbnailScaleUpAnimation` – This will create an animation that scales up from a thumbnail image from specified position on the screen.
- `MakeCustomAnimation` – This creates an animation from resources in the application. There is one animation for when the activity opens and another for when the activity stops.

The new `TimeAnimator` class provides an interface `TimeAnimator.ITimeListener` that can notify an application every time a frame changes in an animation. For example, consider the following implementation of `TimeAnimator.ITimeListener`:

```
class MyTimeListener : Java.Lang.Object,  TimeAnimator.ITimeListener
{
    public void OnTimeUpdate(TimeAnimator animation, long totalTime, long deltaTime)
    {
        Log.Debug("Activity1", "totalTime={0}, deltaTime={1}", totalTime, deltaTime);
    }
}
```

And now to use the class, an instance of `TimeAnimator` is created, and the listener is set:

```
var animator = new TimeAnimator();
animator.SetTimeListener(new MyTimeListener());
animator.Start();
```

As the `TimeAnimator` instance is running, it will invoke `ITimeAnimator.ITimeListener`, which will then log the how long the animator has been running and how long it as been since the last time the method has been invoked.

Application Stack Navigation

Android 4.1 improves on the application stack navigation that was introduced in Android 3.0. By specifying the `ParentName` property of the `ActivityAttribute`, Android can open the proper parent Activity when the user presses the Up button on the action bar - Android will instantiate the Activity specified by the `ParentName` property. This allows applications to preserve hierarchy of activities that make a given task.

For most applications setting the `ParentName` on the activity is sufficient information for Android to provide the correct behavior for navigating the application stack; Android will synthesize the necessary back stack by creating a series of Intents for each parent activity. However, because this is an artificial application stack, each synthetic activity will not have the saved state that a natural activity would have. To provide saved state to a synthetic parent activity, an Activity may override the `OnPrepareNavigationUpTaskStack` method. This method receives a `TaskStackBuilder` instance that will have a collection of Intent objects that Android will use to create the back stack. The activity may modify these Intents so that, as the synthetic activity is created, it will receive the proper state information.

For more complex scenarios, there are new methods on the Activity class that may be used to handle the behavior of Up navigation and construct the back stack:

- `OnNavigateUp` – By overriding this method it is possible to perform a custom action when the **Up** button is pressed.
- `NavigateUpTo` – Calling this method will cause the application to navigate from the current activity to the activity specified by a given intent.
- `ParentActivityIntent` – This is used to obtain an Intent that will launch the parent activity of the current activity.
- `ShouldUpRecreateTask` – This method is used to query if the synthetic back stack must be created to navigate up to a parent activity. Returns `true` if the synthetic stack must be created.
- `FinishAffinity` – Calling this method will finish the current activity and all activities below it in the current task that have the same task affinity.
- `OnCreateNavigateUpTaskStack` – This method is overridden when it is necessary to have complete control over how the synthetic stack is created.

Camera

There is a new interface, `Camera.IAutoFocusMoveCallback`, which can be used to detect when the auto focus has started or stopped moving. An example of this new interface can be seen in the following snippet:

```
public class AutoFocusCallbackActivity : Activity, Camera.IAutoFocusCallback
{
    public void OnAutoFocus(bool success, Camera camera)
    {
        // camera is an instance of the camera service object.

        if (success)
        {
            // Auto focus was successful - do something here.
        }
        else
        {
            // Auto focus didn't happen for some reason - react to that here.
        }
    }
}
```

The new class `MediaActionSound` provides a set of API's for producing sounds for the various media actions. There are several actions that can occur with a camera, these are defined by the enum `Android.Media.MediaActionSoundType`:

- `MediaActionSoundType.FocusComplete` – This sound that is played when focusing has completed.
- `MediaActionSoundType.ShutterClick` – This sound will be played when a still image picture is taken.
- `MediaActionSoundType.StartVideoRecording` – This sound is used indicate the start of video recording.
- `MediaActionSoundType.StopVideoRecording` – This sound will be played to indicate the end of video recording.

An example of how to use the `MediaActionSound` class can be seen in the following snippet:

```
var mediaActionPlayer = new MediaActionSound();

// Preload the sound for a shutter click.
mediaActionPlayer.Load(MediaActionSoundType.ShutterClick);
var button = FindViewById<Button>(Resource.Id.MyButton);

// Play the sound on a button click.
button.Click += (sender, args) => mediaActionPlayer.Play(MediaActionSoundType.ShutterClick);

// This releases the preloaded resources. Don't make any calls on
// mediaActionPlayer after this.
mediaActionPlayer.Release();
```

Connectivity

Android Beam

Android Beam is an NFC based technology that allows two Android devices to communicate with each other. Android 4.1 provides better support for the transfer of large files. When using the new method `NfcAdapter.SetBeamPushUris()` Android will switch between alternate transport mechanisms (such as Bluetooth) to achieve a fast transfer speed.

Network Services Discovery

Android 4.1 contains new API's for multicast DNS-based service discovery. This allows an application to detect and connect over Wi-Fi to other devices such as printers, cameras, and media devices. These new API's are in the `Android.Net.Nsd` package.

To create a service that may be consumed by other services, the `NsdServiceInfo` class is used to create an object that will define the properties of a service. This object is then provided to `NsdManager.RegisterService()` along with an implementation of `NsdManager.ResolveListener`. Implementations of `NsdManager.ResolveListener` are used to notify of a successful registration and to unregister the service.

To discover services on the network, and implementation of `Nsd.DiscoveryListener` passed to `NsdManager.discoverServices()`.

Network Usage

A new method, `ConnectivityManager.IsActiveNetworkMetered` allows a device to check if it is connected to a metered network. This method can be used to help manage data usage by accurately informing users that there might be expensive charges for data operations.

WiFi Direct Service Discovery

The `WifiP2pManager` class was introduced in Android 4.0 to support *zeroconf*. Zeroconf (zero configuration networking) is a set of techniques that allows devices (computers, printers, phones) to connect to networks automatically, with the intervention of human network operators or special configuration servers.

In Jelly Bean, `WifiP2pManager` can discover nearby devices using either *Bonjour* or *Upnp*. Bonjour is Apple's implementation of zeroconf. Upnp is set of networking protocols that also supports zeroconf. The following methods added to the `WiFiP2pManager` to support Wi-Fi service discovery:

- `AddLocalService()` – This method is used announce an application as a service over Wi-Fi for discovery by peers.
- `AddServiceRequest( )` – This method is to send a service discovery request to the framework. It is used to initialize the Wi-Fi service discovery.
- `SetDnsSdResponseListeners()` – This method is used to register callbacks to be invoked on receiving a response to discovery requests from Bonjour.
- `SetUpnpServiceResponseListener()` – This method is used to register callbacks to be invoked on receiving a response to discovery requests Upnp.

Content Providers

The `ContentResolver` class has received a new method, `AcquireUnstableContentProvider`. This method allows an application to acquire an "unstable" content provider. Normally, when an application acquires a content provider, and that content provider crashes, so will the application. With this method call, an application will not crash if the content provider crashes. Instead, `Android.OS.DeadObjectionException` will be thrown from calls on the content provider to inform an application that the content provider has gone away. An "unstable" content provider is useful when interacting with content providers from other applications – it is less likely that buggy code from another application will affect another application.

Copy and Paste With Intents

The `Intent` class can now have a `ClipData` object associated with it via the `Intent.ClipData` property. This method allows for extra data from the clipboard to be transmitted with the intent. An instance of `ClipData` can contain one or more `ClipData.Item`. `ClipData.Item`'s are items of the following types:

- **Text** – This is any string of text, either HTML or any string whose format is supported by the built-in Android style spans.
- **Intent** – Any `Intent` object.
- **Uri** – This can be any URI, such as an HTTP bookmark or the URI to a content provider.

Isolated Services

An isolated service is a service that runs under its own special process and has no permissions of its own. The only communication with the service is when starting up the service and binding to it via the Service API. It is possible to declare a service as isolated by setting the property `IsolatedProcess="true"` in the `ServiceAttribute` that adorns a service class.

Media

The new `Android.Media.MediaCodec` class provides an API to low-level media codecs. Applications can query the system to find out what low level codecs are available on the device.

The new `Android.Media.Audiofx.AudioEffect` subclasses have been added to support additional audio pre-processing on captured audio:

- `Android.Media.Audiofx.AcousticEchoCanceler` – This class is used for pre-processing audio to remove the signal from a remote party from a captured audio signal. For example, removing the echo from a voice communication application.
- `Android.Media.Audiofx.AutomaticGainControl` – This class is used to normalize the captured signal by boosting or lowering an input signal so that the output signal is constant.
- `Android.Media.Audiofx.NoiseSuppressor` – This class will remove background noise from the captured signal.

Not all devices will support these effects. The method `AudioEffect.IsAvailable` should be called by an application to see if the audio effect in question is supported on the device running the application.

The `MediaPlayer` class now supports gapless playback with the `SetNextMediaPlayer()` method. This new method specifies the next MediaPlayer to start when the current media player finishes its playback.

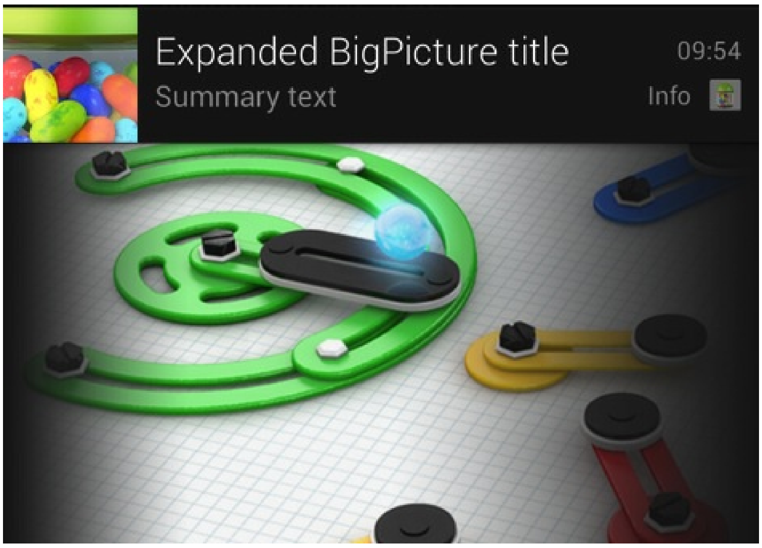
The following new classes provide standard mechanisms and UI for selecting where media will be played:

- `MediaRouter` – This class allows applications to control the routing of media channels from a device to external speakers or other devices.
- `MediaRouterActionProvider` and `MediaRouteButton` – These classes help provide a consistent UI for selecting and playing media.

Notifications

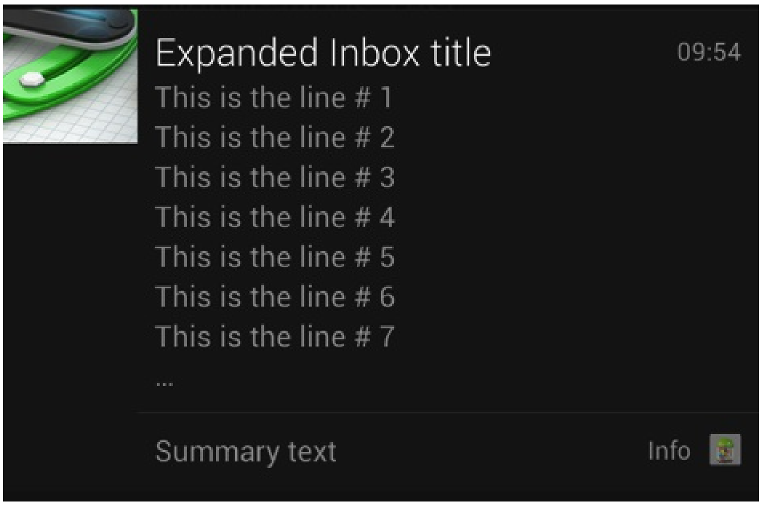
Android 4.1 allows applications more flexibility and control with displaying notifications. Applications can now show bigger and better notifications to users. A new method, `NotificationBuilder.SetStyle()` allows for one of new three new style to be set on notifications:

- `Notification.BigPictureStyle` – This is a helper class that will generate notifications that will have an image in them. The following image shows an example of a notification with a big image:

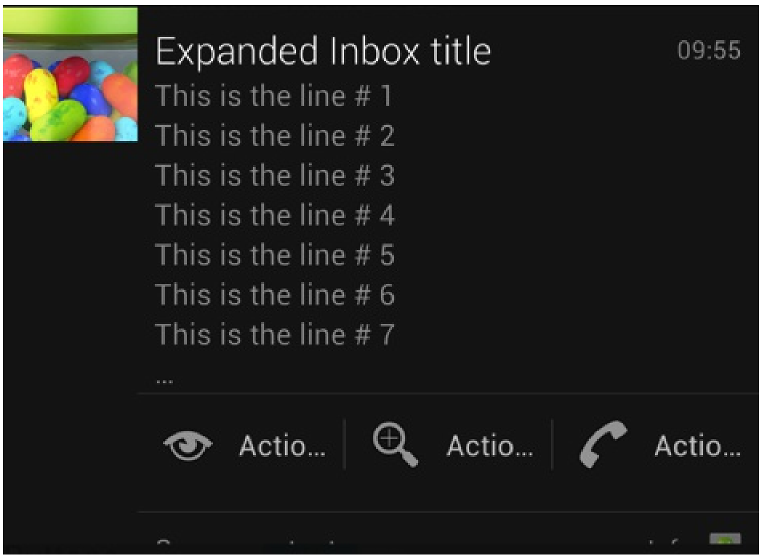


- `Notification.BigTextStyle` – This is a helper class that will generate notifications that will have multiple lines of text, such as e-mail. An example of this new notification style can be seen in the following screenshot:

- `Notification.InboxStyle` – This is a helper class that will generate notifications that contain a list of strings, such as snippets from an e-mail message, as shown in this screenshot:



It is possible to add up to two action buttons at the bottom of a notification message when the notification is using the normal or larger style. An example of this can be seen in the following screenshot, where the action buttons are visible at the bottom of the notification:



The `Notification` class has received new constants that allow a developer to specify one of five priority levels for a notification. These can be set on a notification using the `Priority` property.

Permissions

The following new permissions have been added:

- `READ_EXTERNAL_STORAGE` - The application requires read only access to external storage. Currently all applications have read access by default, but future releases of Android will require applications explicitly request read access.
- `READ_USER_DICTIONARY` - Allows a read-access to the user's word dictionary.
- `READ_CALL_LOG` - Allows an application to obtain information about incoming and outgoing calls by reading the call log.
- `WRITE_CALL_LOG` - Allows an application to write to the call log on the phone.
- `WRITE_USER_DICTIONARY` - Allows an application to write to the user's word dictionary.

An important change to note `READ_EXTERNAL_STORAGE` – currently this permission is automatically granted by Android. Future versions of Android will require an application to request this permission before granted the permission.

Summary

This article introduced some of the new API's that are available in Android 4.1 (API Level 16). It highlighted some of changes for animations and animating the launch of an activity, and introduced the new API's for network discovery of other devices using protocols such as Bonjour or UPnP. Other changes to the API were highlighted as well, such as the ability to cut and paste data via intents, the ability to use isolated services or "unstable" content providers.

This article then went on to introduce the updates to notifications, and discussed some of the new permissions that have been introduced with Android 4.1

Related Links

- Time Animation Example (sample)
- Android 4.1 APIs
- Tasks and Back Stacks
- Navigation with Back and Up

Ice Cream Sandwich Features

11/2/2020 • 2 minutes to read • Edit Online

This article describes several of the new features available to application developers with the Android 4 API - Ice Cream Sandwich. It covers several new user interface technologies and then examines a variety of new capabilities that Android 4 offers for sharing data between applications and between devices.

Overview

Android OS version 4.0 (API Level 14) represents a major reworking of the Android Operating System and includes a number of important changes and upgrades, including:

- **Updated User Interface** – Several new UI features give developers more power and flexibility when they create application user interfaces. These new features include: `GridLayout` , `PopupMenu` , `Switch` widget, and `TextureView` .
- **Better Hardware Acceleration** – 2D rendering now takes place on the GPU for all Android controls. Additionally, hardware acceleration is on, by default, in all applications developed for Android 4.0.
- **New Data APIs** – There's new access to data that was not previously officially accessible, such as calendar data and the user profile of the device owner.
- **App Data Sharing** – Sharing data between applications and devices is now easier than ever via technologies such as the `ShareActionProvider` , which makes it easy to create a sharing action from an Action Bar, and *Android Beam* for *Near Field Communications (NFC)* , which makes it a snap to share data across devices in close proximity to each other.

In this article, we're going to explore these features and other changes that have been made to the Android 4.0 API, and we'll explain how to use each feature with Xamarin.Android.

User Interface Features

A variety of new user interface technologies are available with Android 4, including:

- **GridLayout** – Supports 2D grid layout of controls.
- **Switch widget** – Allows toggling between ON or OFF.
- **TextureView** – Enables video and OpenGL content within a view.
- **Navigation Bar** – Contains virtual buttons for back, home, and multi-tasking.

Additionally, other UI elements have been enhanced, such as the `<a href"/guides/android/user_interface/popup_menus">PopupMenu</a>`, which is now easier to work with, and tabs, which have a more polished appearance.

Sharing Features

Android 4 includes several new technologies that let us share data across devices and across applications. It also provides access to various types of data that were not previously available, such as calendar information and the device owner's user profile. In this section we'll examine a variety of features offered by Android 4 that address these areas including:

- **Android Beam** – Allows data sharing via NFC.
- **ShareActionProvider** – Creates a provider that allows developers to specify sharing actions from the Action Bar.

- **User Profile** – Provides access to profile data of the device owner.
- **Calendar API** – Provides access to calendar data from the calendar provider.

x86 Emulators

ICS does not yet support development with an x86 emulator. x86 emulators are only supported with Android 2.3.3, API level 10. See Configuring the x86 Emulator for more information.

Summary

This article covered a variety of the new technologies that are now available with Android 4. We reviewed new user interface features such as the *GridLayout*, *PopupMenu*, and *Switch* widget. We also looked at some of the new support for controlling the system UI, as well as how to work with the *TextureView*. Then we discussed a variety of new sharing technologies. We covered how *Android Beam* let's you share information across devices that use *NFC*, discussed the new *Calendar API*, and also showed how to use the built in *ShareActionProvider*. Finally, we examined how to use the *ContactsContract* provider to access user profile data.

Related Links

- TextureViewDemo (sample)
- CalendarDemo (sample)
- Tab Layout Tutorial
- Ice Cream Sandwich
- Android 4.0 Platform

Intro to ContentProviders

11/2/2020 • 2 minutes to read • Edit Online

The Android operating system uses content providers to facilitate access to shared data such as media files, contacts and calendar information. This article introduces the ContentProvider class, and provides two examples of how to use it.

Content Providers Overview

A *ContentProvider* encapsulates a data repository and provides an API to access it. The provider exists as part of an Android application that usually also provides a UI for displaying/managing the data. The key benefit of using a content provider is enabling other applications to easily access the encapsulated data using a provider client object (called a *ContentResolver*). Together, a content provider and content resolver offer a consistent inter-application API for data access that is simple to build and consume. Any application can choose to use `ContentProviders` to manage data internally and also to expose it to other applications.

A `ContentProvider` is also required for your application to provide custom search suggestions, or if you want to provide the ability to copy complex data from your application to paste into other applications. This document shows how to access and build `ContentProviders` with Xamarin.Android.

The structure of this section is as follows:

- **How it works** – An overview of what the `ContentProvider` is designed for and how it works.
- **Consuming a Content Provider** – An example accessing the Contacts list.
- **Using ContentProvider to share data** – Writing and consuming a `ContentProvider` in the same application.

`ContentProviders` and the cursors that operate on their data are often used to populate ListViews. Refer to the ListViews and Adapters guide for more information on how to use those classes.

`ContentProviders` exposed by Android (or other applications) are an easy way to include data from other sources in your application. They allow you to access and present data such as the Contacts list, photos or calendar events from within your application, and let the user interact with that data.

Custom `ContentProviders` are a convenient way to package your data for use inside your own app, or for use by other applications (including special uses like custom search and copy/paste).

The topics in this section provide some simple examples of consuming and writing `ContentProvider` code.

Related Links

- ContactsAdapter Demo (sample)
- SimpleContentProvider (sample)
- Content Providers Developers Guide
- ContentProvider Class Reference
- ContentResolver Class Reference
- ListView Class Reference
- CursorAdapter Class Reference
- UriMatcher Class Reference

- Android.Provider
- ContactsContract Class Reference

How Content Providers Work

7/8/2021 • 2 minutes to read • Edit Online

There are two classes involved in a `ContentProvider` interaction:

- **ContentProvider** – Implements an API that exposes a set of data in a standard way. The main methods are Query, Insert, Update and Delete.
- **ContentResolver** – A static proxy that communicates with a `ContentProvider` to access its data, either from within the same application or from another application.

A content provider is normally backed by an SQLite database, but the API means that consuming code does not need to know anything about the underlying SQL. Queries are done via a Uri using constants to reference column names (to reduce dependencies on the underlying data structure), and an `ICursor` is returned for the consuming code to iterate over.

Consuming a ContentProvider

`ContentProviders` expose their functionality through a Uri that is registered in the **AndroidManifest.xml** of the application that publishes the data. There is a convention where the Uri and the data columns that are exposed should be available as constants to make it easy to bind to the data. Android's built-in `ContentProviders` all provide convenience classes with constants that reference the data structure in the `Android.Providers` namespace.

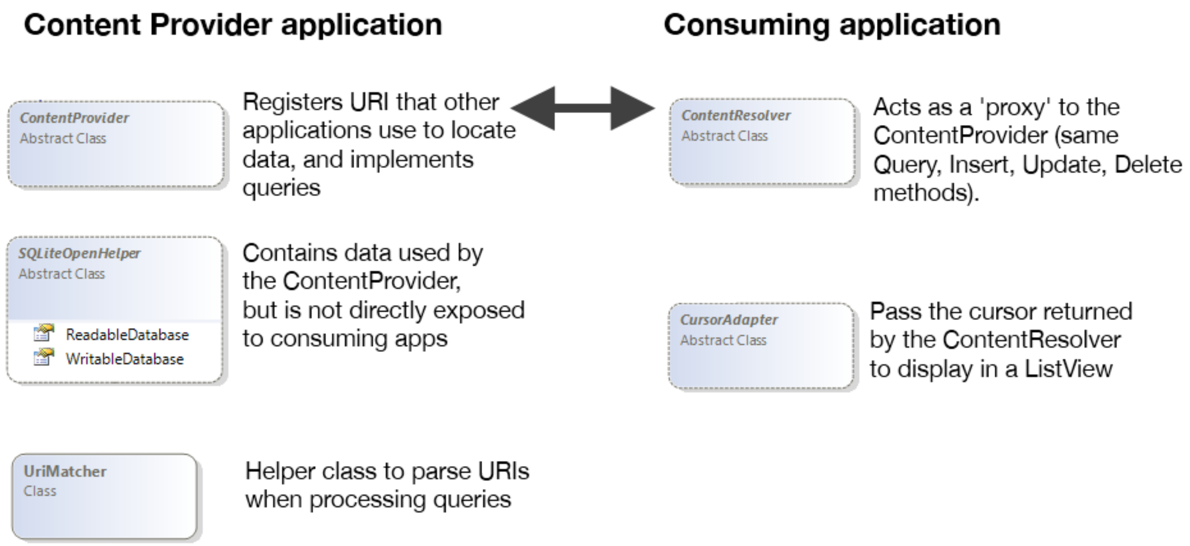
Built-In Providers

Android offers access to a wide range of system and user data using `ContentProviders`:

- *Browser* – bookmarks and browser history (requires permission `READ_HISTORY_BOOKMARKS` and/or `WRITE_HISTORY_BOOKMARKS`).
- *CallLog* – recent calls made or received with the device.
- *Contacts* – detailed information from the user's contact list, including people, phones, photos & groups.
- *MediaStore* – contents of the user's device: audio (albums, artists, genres, playlists), images (including thumbnails) & video.
- *Settings* – system-wide device settings and preferences.
- *UserDictionary* – contents of the user-defined dictionary used for predictive text input.
- *Voicemail* – history of voicemail messages.

Classes Overview

The primary classes used when working with a `ContentProvider` are shown here:



In this diagram, the `ContentProvider` implements queries and registers URI's that other applications use to locate data. The `ContentResolver` acts as a 'proxy' to the `ContentProvider` (Query, Insert, Update, and Delete methods). The `SQLiteOpenHelper` contains data used by the `ContentProvider`, but it is not directly exposed to consuming apps. The `CursorAdapter` passes the cursor returned by the `ContentResolver` to display in a `ListView`. The `UriMatcher` is a helper class that parses URIs when processing queries.

The purpose of each class is described below:

- **ContentProvider** – Implement this abstract class's methods to expose data. The API is made available to other classes and applications via the Uri attribute that is added to the class definition.
- **SQLiteOpenHelper** – Helps implement the SQLite datastore that is exposed by the `ContentProvider`.
- **UriMatcher** – Use `UriMatcher` in your `ContentProvider` implementation to help manage Uris that are used to query the content.
- **ContentResolver** – Consuming code uses a `ContentResolver` to access a `ContentProvider` instance. The two classes together take care of the inter-process communication issues, allowing data to be easily shared between applications. Consuming code never creates a `ContentProvider` class explicity; instead, the data is accessed by creating a cursor based on a Uri exposed by the `ContentProvider` application.
- **CursorAdapter** – Use `CursorAdapter` or `SimpleCursorAdapter` to display data accessed via a `ContentProvider`.

The `ContentProvider` API allows consumers to perform a variety of operations on the data, such as:

- Query data to return lists or individual records.
- Modify individual records.
- Add new records.
- Delete records.

This document contains an example that uses a system-provided `ContentProvider`, as well as a simple read-only example that implements a custom `ContentProvider`.

Using the Contacts ContentProvider

7/8/2021 • 5 minutes to read • Edit Online

Code that uses access data exposed by a `ContentProvider` doesn't require a reference to the `ContentProvider` class at all. Instead, a Uri is used to create a cursor over the data exposed by the `ContentProvider`. Android uses the Uri to search the system for the application that exposes a `ContentProvider` with that identifier. The Uri is a string, typically in a reverse-DNS format such as `com.android.contacts/data`.

Rather than making developers remember this string, the Android *Contacts* provider exposes its metadata in the `android.provider.ContactsContract` class. This class is used to determine the Uri of the `ContentProvider` as well as the names of the tables and columns that can be queried.

Some data types also require special permission to access. The built-in contacts list requires the `android.permission.READ_CONTACTS` permission in the **AndroidManifest.xml** file.

There are three ways to create a cursor from the Uri:

1. **ManagedQuery()** – The preferred approach in Android 2.3 (API Level 10) and earlier, a `ManagedQuery` returns a cursor and also automatically manages refreshing the data and closing the cursor. This method is deprecated in Android 3.0 (API Level 11).

2. **ContentResolver.Query()** – Returns an unmanaged cursor, which means it must be refreshed and closed explicitly in code.

3. **CursorLoader().LoadInBackground()** – Introduced in Android 3.0 (API Level 11), `CursorLoader` is now the preferred way to consume a `ContentProvider`. `CursorLoader` queries a `ContentResolver` on a background thread so the UI isn't blocked. This class can be accessed in older versions of Android using the v4 compatibility library.

Each of these methods has the same basic set of inputs:

- **Uri** – The fully qualified name of the `ContentProvider`.
- **Projection** – Specification of which columns to select for the cursor.
- **Selection** – Similar to a SQL `WHERE` clause.
- **SelectionArgs** – Parameters to be substituted in the Selection.
- **SortOrder** – Columns to sort by.

Creating Inputs for a Query

The `ContactsProvider` sample code performs a very simple query against Android's built-in Contacts provider. You do not need to know the actual Uri or column names - all the information required to query the Contacts `ContentProvider` is available as constants exposed by the `ContactsContract` class.

Regardless of which method is used to retrieve the cursor, these same objects are used as parameters as shown in the *ContactsProvider/ContactsAdapter.cs* file:

```
var uri = ContactsContract.Contacts.ContentUri;
string[] projection = {
   ContactsContract.Contacts.InterfaceConsts.Id,
   ContactsContract.Contacts.InterfaceConsts.DisplayName,
   ContactsContract.Contacts.InterfaceConsts.PhotoId,
};
```

For this example, the `selection`, `selectionArgs` and `sortOrder` will be ignored by setting them to `null`.

Creating a Cursor from a Content Provider Uri

Once the parameter objects have been created, they can be used in one of the following three ways:

Using a Managed Query

Applications targeting Android 2.3 (API Level 10) or earlier should use this method:

```
var cursor = activity.ManagedQuery(uri, projection, null, null, null);
```

This cursor will be managed by Android so you do not need to close it.

Using ContentResolver

Accessing `ContentResolver` directly to get a cursor against a `ContentProvider` can be done like this:

```
var cursor = activity.ContentResolver(uri, projection, null, null, null);
```

This cursor is unmanaged, so it must be closed when no longer required. Ensure that the code closes a cursor that is open, otherwise an error will occur.

```
cursor.Close();
```

Alternatively, you can call `StartManagingCursor()` and `StopManagingCursor()` to 'manage' the cursor. Managed cursors are automatically deactivated and re-queried when Activities are stopped and restarted.

Using CursorLoader

Applications built for Android 3.0 (API Level 11) or newer should use this method:

```
var loader = new CursorLoader (activity, uri, projection, null, null, null);
var cursor = (ICursor)loader.LoadInBackground();
```

The `CursorLoader` ensures that all cursor operations are done on a background thread, and can intelligently re-use an existing cursor across activity instances when an activity is restarted (e.g. due to a configuration change) rather that reload the data again.

Earlier Android versions can also use the `CursorLoader` class by using the v4 support libraries.

Displaying the Cursor Data with a Custom Adapter

To display the contact image we'll use a custom adapter, so that we can manually resolve the `PhotoId` reference to an image file path.

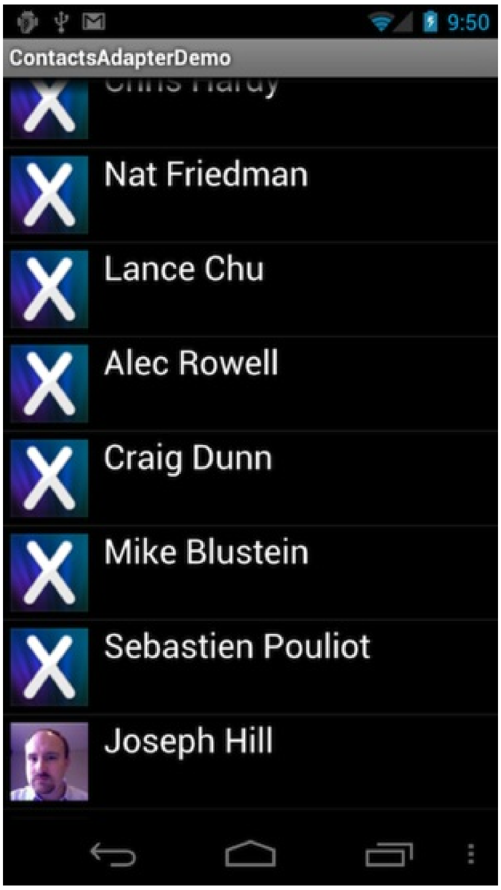
To display data with a custom adapter, the example uses a `CursorLoader` to retrieve all the Contact data into a local collection in the **FillContacts** method from **ContactsProvider/ContactsAdapter.cs**:

```
void FillContacts ()
{
   var uri = ContactsContract.Contacts.ContentUri;
   string[] projection = {
       ContactsContract.Contacts.InterfaceConsts.Id,
       ContactsContract.Contacts.InterfaceConsts.DisplayName,
       ContactsContract.Contacts.InterfaceConsts.PhotoId
  };
   // CursorLoader introduced in Honeycomb (3.0, API11)
   var loader = new CursorLoader(activity, uri, projection, null, null, null);
   var cursor = (ICursor)loader.LoadInBackground();
   contactList = new List<Contact> ();
   if (cursor.MoveToFirst ()) {
      do {
          contactList.Add (new Contact{
              Id = cursor.GetLong (cursor.GetColumnIndex (projection [0])),
              DisplayName = cursor.GetString (cursor.GetColumnIndex (projection [1])),
              PhotoId = cursor.GetString (cursor.GetColumnIndex (projection [2]))
          });
       } while (cursor.MoveToNext());
   }
}
```

Then implement the BaseAdapter's methods using the `contactList` collection. The adapter is implemented just as it would be with any other collection – there is no special handling here because the data is sourced from a `ContentProvider`:

```
Activity activity;
public ContactsAdapter (Activity activity)
{
   this.activity = activity;
   FillContacts ();
}
public override int Count {
   get { return contactList.Count; }
}
public override Java.Lang.Object GetItem (int position)
{
  return null; // could wrap a Contact in a Java.Lang.Object to return it here if needed
}
public override long GetItemId (int position)
{
   return contactList [position].Id;
}
public override View GetView (int position, View convertView, ViewGroup parent)
{
   var view = convertView ?? activity.LayoutInflater.Inflate (Resource.Layout.ContactListItem, parent,
false);
   var contactName = view.FindViewById<TextView> (Resource.Id.ContactName);
   var contactImage = view.FindViewById<ImageView> (Resource.Id.ContactImage);
   contactName.Text = contactList [position].DisplayName;
   if (contactList [position].PhotoId == null) {
       contactImage = view.FindViewById<ImageView> (Resource.Id.ContactImage);
       contactImage.SetImageResource (Resource.Drawable.ContactImage);
   } else {
       var contactUri = ContentUris.WithAppendedId (ContactsContract.Contacts.ContentUri, contactList
[position].Id);
       var contactPhotoUri = Android.Net.Uri.WithAppendedPath (contactUri,
Contacts.Photos.ContentDirectory);
       contactImage.SetImageURI (contactPhotoUri);
   }
   return view;
}
```

The image is displayed (if it exists) using the Uri to the image file on the device. The application looks like this:



Using a similar code pattern, your application can access a wide variety of system data including the user's photos, videos and music. Some data types require special permissions to be requested in the project's **AndroidManifest.xml**.

Displaying the Cursor Data with a SimpleCursorAdapter

The cursor could also be displayed with a `SimpleCursorAdapter` (although only the name will be displayed, not the photo). This code demonstrates how to use a `ContentProvider` with `SimpleCursorAdapter` (this code does not appear in the sample):

```
var uri = ContactsContract.Contacts.ContentUri;
string[] projection = {
   ContactsContract.Contacts.InterfaceConsts.Id,
   ContactsContract.Contacts.InterfaceConsts.DisplayName
};
var loader = new CursorLoader (this, uri, projection, null, null, null);
var cursor = (ICursor)loader.LoadInBackground();
var fromColumns = new string[] {ContactsContract.Contacts.InterfaceConsts.DisplayName};
var toControlIds = new int[] {Android.Resource.Id.Text1};
adapter = new SimpleCursorAdapter (this, Android.Resource.Layout.SimpleListItem1, cursor, fromColumns,
toControlsIds);
listView.Adapter = adapter;
```

Refer to the ListViews and Adapters for further information on implementing `SimpleCursorAdapter`.

Related Links

- ContactsAdapter Demo (sample)

Creating a Custom ContentProvider

7/8/2021 • 9 minutes to read • Edit Online

The previous section demonstrated how to consume data from a built-in ContentProvider implementation. This section will explain how to build a custom ContentProvider and then consume its data.

About ContentProviders

A content provider class must inherit from `ContentProvider`. It should consist of an internal data store that is used to respond to queries and it should expose Uris and MIME Types as constants to help consuming code make valid requests for data.

URI (Authority)

`ContentProviders` are accessed in Android using a Uri. An application that exposes a `ContentProvider` sets the Uris that it will respond to in its **AndroidManifest.xml** file. When the application is installed, these Uris are registered so that other applications can access them.

In Mono for Android, the content provider class should have a `[ContentProvider]` attribute to specify the Uri (or Uris) that should be added to **AndroidManifest.xml**.

Mime Type

The typical format for MIME Types consists of two parts. Android `ContentProviders` commonly use these two strings for the first part of the MIME Type:

1. `vnd.android.cursor.item` – to represent a single row, use the `ContentResolver.CursorItemBaseType` constant in code.
2. `vnd.android.cursor.dir` – for multiple rows, use the `ContentResolver.CursorDirBaseType` constant in code.

The second part of the MIME Type is specific to your application, and should use a reverse-DNS standard with a `vnd.` prefix. The sample code uses `vnd.com.xamarin.sample.Vegetables`.

Data Model Metadata

Consuming applications need to construct Uri queries to access different types of data. The base Uri can be expanded to refer to a particular table of data and may also include parameters to filter the results. The columns and clauses used with the resulting cursor to display data must also be declared.

To ensure that only valid Uri queries are constructed, it is customary to provide the valid strings as constant values. This makes it easier to access the `ContentProvider` because it makes the values discoverable via code-completion, and prevents typos in the strings.

In the previous example the `android.provider.ContactsContract` class exposed the metadata for the Contacts data. For our custom `ContentProvider` we will just expose the constants on the class itself.

Implementation

There are three steps to creating and consuming a custom `ContentProvider`:

1. **Create a database class** – Implement `SQLiteOpenHelper`.
2. **Create a `ContentProvider` class** – Implement `ContentProvider` with an instance of the database, metadata exposed as constant values and methods to access the data.

3. **Access the `ContentProvider` via its Uri** – Populate a `CursorAdapter` using the `ContentProvider`, accessed via its Uri.

As previously discussed, `ContentProviders` can be consumed from applications other than where they are defined. In this example the data is consumed in the same application, but keep in mind that other applications can also access it as long as they know the Uri and information about the schema (which is usually exposed as constant values).

Create a Database

Most `ContentProvider` implementations will be based on a `SQLite` database. The example database code in **SimpleContentProvider/VegetableDatabase.cs** creates a very simple two-column database, as shown:

```
class VegetableDatabase  : SQLiteOpenHelper {
  const string create_table_sql =
    "CREATE TABLE [vegetables] ([_id] INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE, [name] TEXT NOT
NULL UNIQUE)";
  const string DatabaseName = "vegetables.db";
  const int DatabaseVersion = 1;

  public VegetableDatabase(Context context) : base(context, DatabaseName, null, DatabaseVersion) { }
  public override void OnCreate(SQLiteDatabase db)
  {
    db.ExecSQL(create_table_sql);
    // seed with data
    db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Vegetables')");
    db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Fruits')");
    db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Flower Buds')");
    db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Legumes')");
    db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Bulbs')");
    db.ExecSQL("INSERT INTO vegetables (name) VALUES ('Tubers')");
  }
  public override void OnUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
  {
    throw new NotImplementedException();
  }
}
```

The database implementation itself does not need any special considerations to be exposed with a `ContentProvider`, however if you intend to bind the `ContentProvider's` data to a `ListView` control then a unique integer column named `_id` must be part of the result set. See the ListViews and Adapters document for more details on using the `ListView` control.

Create the ContentProvider

The rest of this section gives step-by-step instructions on how the **SimpleContentProvider/VegetableProvider.cs** example class was built.

Initialize the Database

The first step is to subclass `ContentProvider` and add the database that it will use.

```
public class VegetableProvider : ContentProvider
{
    VegetableDatabase vegeDB;
    public override bool OnCreate()
    {
       vegeDB = new VegetableDatabase(Context);
        return true;
    }
}
```

The rest of the code will form the actual content provider implementation that allows the data to be discovered and queried.

Add Metadata for Consumers

There are four different types of metadata that we are going to expose on the `ContentProvider` class. Only the authority is required, the rest are done by convention.

- **Authority** – The `ContentProvider` attribute *must* be added to the class so that it is registered with the Android when the application is installed.
- **Uri** – The `CONTENT_URI` is exposed as a constant so that it is easy to use in code. It should match the Authority, but include the scheme and base path.
- **MIME Types** – Lists of results and single results are treated as different content types, so we define two MIME Types to represent them.
- **InterfaceConsts** – Provide a constant value for each data column name, so that consuming code can easily discover and refer to them without risking typographical errors.

This code shows how each of these items is implemented, adding to the database definition from the previous step:

```
[ContentProvider(new string[] { CursorTableAdapter.VegetableProvider.AUTHORITY })]
public class VegetableProvider : ContentProvider
{
   public const string AUTHORITY = "com.xamarin.sample.VegetableProvider";
   static string BASE_PATH = "vegetables";
   public static readonly Android.Net.Uri CONTENT_URI = Android.Net.Uri.Parse("content://" + AUTHORITY + "/"
+ BASE_PATH);
   // MIME types used for getting a list, or a single vegetable
   public const string VEGETABLES_MIME_TYPE = ContentResolver.CursorDirBaseType +
"/vnd.com.xamarin.sample.Vegetables";
   public const string VEGETABLE_MIME_TYPE = ContentResolver.CursorItemBaseType +
"/vnd.com.xamarin.sample.Vegetables";
   // Column names
   public static class InterfaceConsts {
       public const string Id = "_id";
       public const string Name = "name";
   }
   VegetableDatabase vegeDB;
   public override bool OnCreate()
   {
       vegeDB = new VegetableDatabase(Context);
       return true;
   }
}
```

Implement the URI Parsing Helper

Because consuming code uses Uris to make requests of a `ContentProvider`, we need to be able to parse those requests to determine what data to return. The `UriMatcher` class can help to parse Uris, once it has been initialized with the Uri patterns that the `ContentProvider` supports.

The `UriMatcher` in the example will be initialized with two Uris:

1. *"com.xamarin.sample.VegetableProvider/vegetables"* – request to return the full list of vegetables.
2. *"com.xamarin.sample.VegetableProvider/vegetables/#"* – where the # is a placeholder for a numeric parameter (the `_id` of the row in the database). An asterisk placeholder ("*") can also be used to match a text parameter.

In the code we use the constants to refer to metadata values like the AUTHORITY and BASE_PATH. The return codes will be used in methods that do Uri parsing, to determine what data to return.

```
const int GET_ALL = 0; // return code when list of Vegetables requested
const int GET_ONE = 1; // return code when a single Vegetable is requested by ID
static UriMatcher uriMatcher = BuildUriMatcher();
static UriMatcher BuildUriMatcher()
{
  var matcher = new UriMatcher(UriMatcher.NoMatch);
  // Uris to match, and the code to return when matched
  matcher.AddURI(AUTHORITY, BASE_PATH, GET_ALL); // all vegetables
  matcher.AddURI(AUTHORITY, BASE_PATH + "/#", GET_ONE); // specific vegetable by numeric ID
  return matcher;
}
```

This code is all private to the `ContentProvider` class. Refer to Google's UriMatcher documentation for further information.

Implement the QueryMethod

The simplest `ContentProvider` method to implement is the `Query` method. The implementation below uses the `UriMatcher` to parse the `uri` parameter and call the correct database method. If the `uri` contains an ID parameter then the integer is parsed out (using `LastPathSegment`) and used in the database query.

```
public override Android.Database.ICursor Query(Android.Net.Uri uri, string[] projection, string selection,
string[] selectionArgs, string sortOrder)
{
  switch (uriMatcher.Match(uri)) {
  case GET_ALL:
    return GetFromDatabase();
  case GET_ONE:
    var id = uri.LastPathSegment;
    return GetFromDatabase(id); // the ID is the last part of the Uri
  default:
    throw new Java.Lang.IllegalArgumentException("Unknown Uri: " + uri);
  }
}
Android.Database.ICursor GetFromDatabase()
{
  return vegeDB.ReadableDatabase.RawQuery("SELECT _id, name FROM vegetables", null);
}
Android.Database.ICursor GetFromDatabase(string id)
{
  return vegeDB.ReadableDatabase.RawQuery("SELECT _id, name FROM vegetables WHERE _id = " + id, null);
}
```

The `GetType` method must also be overridden. This method may be called to determine the content type that will be returned for a given Uri. This might tell the consuming application how to handle that data.

```
public override String GetType(Android.Net.Uri uri)
{
  switch (uriMatcher.Match(uri)) {
  case GET_ALL:
    return VEGETABLES_MIME_TYPE; // list
  case GET_ONE:
    return VEGETABLE_MIME_TYPE; // single item
  default:
    throw new Java.Lang.IllegalArgumentExceptoin ("Unknown Uri: " + uri);
   }
}
```

Implement the Other Overrides

Our simple example does not allow for editing or deletion of data, but the Insert, Update and Delete methods must be implemented so add them without an implementation:

```
public override int Delete(Android.Net.Uri uri, string selection, string[] selectionArgs)
{
  throw new Java.Lang.UnsupportedOperationException();
}
public override Android.Net.Uri Insert(Android.Net.Uri uri, ContentValues values)
{
  throw new Java.Lang.UnsupportedOperationException();
}
public override int Update(Android.Net.Uri uri, ContentValues values, string selection, string[]
selectionArgs)
{
  throw new Java.Lang.UnsupportedOperationException();
}
```

That completes the basic `ContentProvider` implementation. Once the application has been installed, the data it exposes will be available both inside the application but also to any other application that knows the Uri to reference it.

Access the ContentProvider

Once the `VegetableProvider` has been implemented, accessing it is done the same way as the Contacts provider at the start of this document: obtain a cursor using the specified Uri and then use an adapter to access the data.

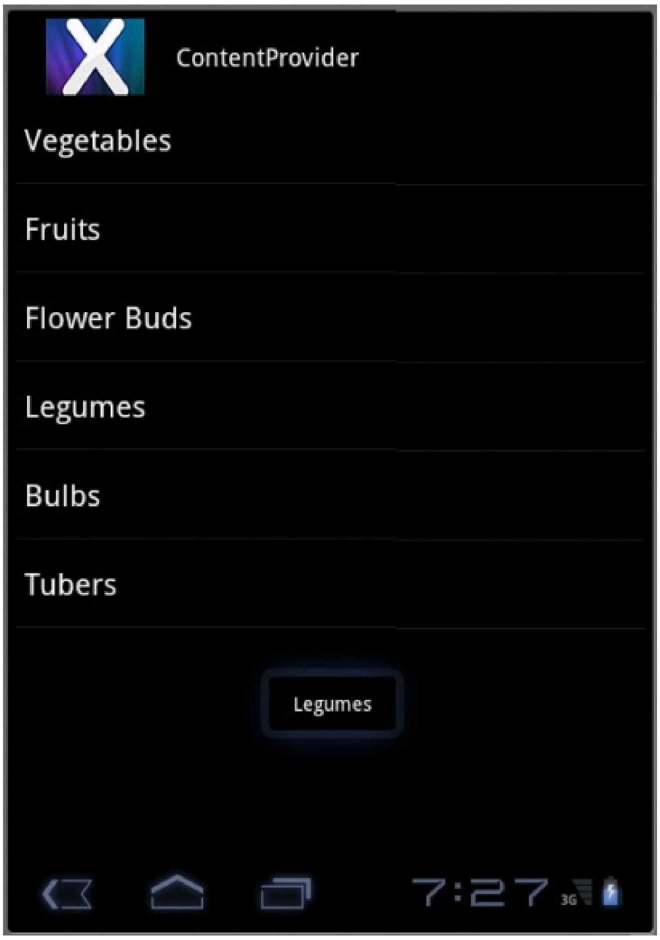
Bind a ListView to a ContentProvider

To populate a `ListView` with data we use the Uri that corresponds to the unfiltered list of vegetables. In the code we use the constant value `VegetableProvider.CONTENT_URI`, which we know resolves to `com.xamarin.sample.vegetableprovider/vegetables`. Our `VegetableProvider.Query` implementation will return a cursor that can then be bound to the `ListView`.

The code in `SimpleContentProvider/HomeScreen.cs` shows how simple it is to display data from a `ContentProvider`:

```
listView = FindViewById<ListView>(Resource.Id.List);
string[] projection = new string[] { VegetableProvider.InterfaceConsts.Id,
VegetableProvider.InterfaceConsts.Name} ;
string[] fromColumns = new string[] { VegetableProvider.InterfaceConsts.Name };
int[] toControlIds = new int[] { Android.Resource.Id.Text1 };

// CursorLoader introduced in Honeycomb (3.0, API_11)
var loader = new CursorLoader(this,
   VegetableProvider.CONTENT_URI, projection, null, null, null);
cursor = (ICursor)loader.LoadInBackground();

// Create a SimpleCursorAdapter
adapter = new SimpleCursorAdapter(this, Android.Resource.Layout.SimpleListItem1, cursor, fromColumns,
toControlIds);
listView.Adapter = adapter;
```

The resulting application looks like this:



Retrieve a Single Item from a ContentProvider

A consuming application might also want to access single rows of data, which can be done by constructing a different Uri that refers to a specific row (for example).

Use `ContentResolver` directly to access a single item, by building up a Uri with the required `Id`.

```
Uri.WithAppendedPath(VegetableProvider.CONTENT_URI, id.ToString());
```

The complete method looks like this:

```
protected void OnListItemClick(object sender, AdapterView.ItemClickEventArgs e)
{
  var id = e.Id;
  string[] projection = new string[] { "name" };
  var uri = Uri.WithAppendedPath(VegetableProvider.CONTENT_URI, id.ToString());
  ICursor vegeCursor = ContentResolver.Query(uri, projection, null, new string[] { id.ToString() }, null);
  string text = "";
  if (vegeCursor.MoveToFirst()) {
    text = vegeCursor.GetInt(0) + " " + vegeCursor.GetString(1);
    Android.Widget.Toast.MakeText(this, text, Android.Widget.ToastLength.Short).Show();
  }
  vegeCursor.Close();
}
```

Related Links

- SimpleContentProvider (sample)

Maps and Location on Android

10/28/2019 • 2 minutes to read • Edit Online

Location Services

This guide introduces location-awareness in Android applications, and illustrates how to get the user's location using the Android Location Service API, as well as the Fused Location Provider available with the Google Location Services API.

Maps

This article discusses how to use maps and location with Xamarin.Android. It covers everything from leveraging the built-in maps application to using the Google Maps Android API V2 directly. Additionally, it explains how to use a single API to work with location services, which allows an application to obtain location fixes via cell tower location, Wi-Fi or GPS.

Location services on Android

7/8/2021 • 13 minutes to read • Edit Online

This guide introduces location-awareness in Android applications and illustrates how to get the user's location using the Android Location Service API, as well as the fused location provider available with the Google Location Services API.

Android provides access to various location technologies such as cell tower location, Wi-Fi, and GPS. The details of each location technology are abstracted through *location providers*, allowing applications to obtain locations in the same way regardless of the provider used. This guide introduces the fused location provider, a part of the Google Play Services, which intelligently determines the best way to obtain the location of the devices based on what providers are available and how the device is being used. Android Location Service API and shows how to communicate with the system location Service using a `LocationManager`. The second part of the guide explores the Android Location Services API using the `LocationManager`.

As a general rule of thumb, applications should prefer to use the fused location provider, falling back the older Android Location Service API only when necessary.

Location fundamentals

In Android, no matter what API you choose for working with location data, several concepts remain the same. This section introduces Location Providers and location-related permissions.

Location providers

Several technologies are used internally to pinpoint the user's location. The hardware used depends on the type of *location provider* selected for the job of collecting data. Android uses three location providers:

- **GPS Provider** – GPS gives the most accurate location, uses the most power, and works best outdoors. This provider uses a combination of GPS and assisted GPS (aGPS), which returns GPS data collected by cellular towers.
- **Network Provider** – Provides a combination of WiFi and Cellular data, including aGPS data collected by cell towers. It uses less power than the GPS Provider, but returns location data of varying accuracy.
- **Passive Provider** – A "piggyback" option using providers requested by other applications or Services to generate location data in an application. This is a less reliable but power-saving option ideal for applications that don't require constant location updates to work.

Location providers are not always available. For example, we might want to use GPS for our application, but GPS might be turned off in Settings, or the device might not have GPS at all. If a specific provider is not available, choosing that provider might return `null`.

Location permissions

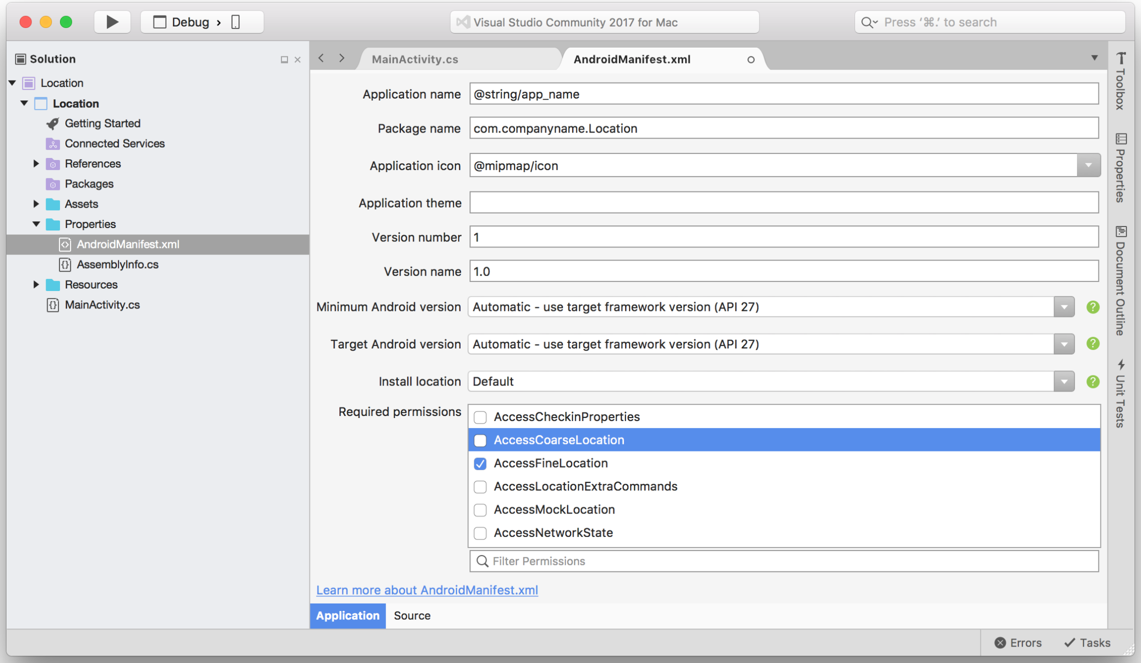
A location-aware application needs access a device's hardware sensors to receive GPS, Wi-Fi, and cellular data. Access is controlled through appropriate permissions in the application's Android Manifest. There are two permissions available – depending on your application's requirements and your choice of API, you will want to allow one:

- `ACCESS_FINE_LOCATION` – Allows an application access to GPS. Required for the *GPS Provider* and *Passive Provider* options (*Passive Provider needs permission to access GPS data collected by another application or Service*). Optional permission for the *Network Provider*.

- `ACCESS_COARSE_LOCATION` – Allows an application access to Cellular and Wi-Fi location. Required for *Network Provider* if `ACCESS_FINE_LOCATION` is not set.

For apps that target API version 21 (Android 5.0 Lollipop) or higher, you can enable `ACCESS_FINE_LOCATION` and still run on devices that do not have GPS hardware. If your app requires GPS hardware, you should explicitly add an `android.hardware.location.gps` `uses-feature` element to the Android Manifest. For more information, see the Android uses-feature element reference.

To set the permissions, expand the **Properties** folder in the **Solution Pad** and double-click **AndroidManifest.xml**. The permissions will be listed under **Required Permissions**:



Setting either of these permissions tells Android that your application needs permission from the user in order to access to the location providers. Devices that run API level 22 (Android 5.1) or lower will ask the user to grant these permissions each time the app is installed. On devices running API level 23 (Android 6.0) or higher, the app should perform a run-time permission check before making a request of the location provider.

> **NOTE**
>
> Note: Setting `ACCESS_FINE_LOCATION` implies access to both coarse and fine location data. You should never have to set both permissions, only the *minimal* permission your app requires to work.

This snippet is an example of how to check that an app has permission for the `ACCESS_FINE_LOCATION` permission:

```
 if (ContextCompat.CheckSelfPermission(this, Manifest.Permission.AccessFineLocation) == Permission.Granted)
{
    StartRequestingLocationUpdates();
    isRequestingLocationUpdates = true;
}
else
{
    // The app does not have permission ACCESS_FINE_LOCATION
}
```

Apps must be tolerant of the scenario where the user will not grant permission (or has revoked the permission) and have a way to gracefully deal with that situation. Please see the Permissions guide for more details on implementing run-time permission checks in Xamarin.Android.

Using the fused location provider

The fused location provider is the preferred way for Android applications to receive location updates from the device because it will efficiently select the location provider during run time to provide the best location information in a battery-efficient fashion. For example, a user walking around outdoors gets the best location reading with GPS. If the user then walks indoors, where GPS works poorly (if at all), the fused location provider may automatically switch to WiFi, which works better indoors.

The fused location provider API provides a variety of other tools to empower location-aware applications, including geofencing and activity monitoring. In this section, we are going to focus on the basics of setting up the `LocationClient`, establishing providers, and getting the user's location.

The fused location provider is part of Google Play Services. The Google Play Services package must be installed and configured properly in the application for the fused location provider API to work, and the device must have the Google Play Services APK installed.

Before a Xamarin.Android application can use the fused location provider, it must add the **Xamarin.GooglePlayServices.Location** package to the project. In addition, the following `using` statements should be added to any source files that reference the classes described below:

```
using Android.Gms.Common;
using Android.Gms.Location;
```

Checking if Google Play Services is installed

A Xamarin.Android will crash if it tries to use the fused location provider when Google Play Services is not installed (or out of date) then a runtime exception would occur. If Google Play Services is not installed, then the application should fall back to the Android Location Service discussed above. If Google Play Services is out of date, then the app could display a message to the user asking them to update the installed version of Google Play Services.

This snippet is an example of how an Android Activity can programmatically check if Google Play Services is installed:

```
bool IsGooglePlayServicesInstalled()
{
    var queryResult = GoogleApiAvailability.Instance.IsGooglePlayServicesAvailable(this);
    if (queryResult == ConnectionResult.Success)
    {
        Log.Info("MainActivity", "Google Play Services is installed on this device.");
        return true;
    }

    if (GoogleApiAvailability.Instance.IsUserResolvableError(queryResult))
    {
        // Check if there is a way the user can resolve the issue
        var errorString = GoogleApiAvailability.Instance.GetErrorString(queryResult);
        Log.Error("MainActivity", "There is a problem with Google Play Services on this device: {0} - {1}",
                  queryResult, errorString);

        // Alternately, display the error to the user.
    }

    return false;
}
```

FusedLocationProviderClient

To interact with the fused location provider, a Xamarin.Android application must have an instance of the `FusedLocationProviderClient`. This class exposes the necessary methods to subscribe to location updates and to retrieve the last known location of the device.

The `OnCreate` method of an Activity is a suitable place to get a reference to the `FusedLocationProviderClient`, as demonstrated in the following code snippet:

```
public class MainActivity: AppCompatActivity
{
    FusedLocationProviderClient fusedLocationProviderClient;

    protected override void OnCreate(Bundle bundle)
    {
        fusedLocationProviderClient = LocationServices.GetFusedLocationProviderClient(this);
    }
}
```

Getting the last known location

The `FusedLocationProviderClient.GetLastLocationAsync()` method provides a simple, non-blocking way for a Xamarin.Android application to quickly obtain the last known location of the device with minimal coding overhead.

This snippet shows how to use the `GetLastLocationAsync` method to retrieve the location of the device:

```
async Task GetLastLocationFromDevice()
{
    // This method assumes that the necessary run-time permission checks have succeeded.
    getLastLocationButton.SetText(Resource.String.getting_last_location);
    Android.Locations.Location location = await fusedLocationProviderClient.GetLastLocationAsync();

    if (location == null)
    {
        // Seldom happens, but should code that handles this scenario
    }
    else
    {
        // Do something with the location
        Log.Debug("Sample", "The latitude is " + location.Latitude);
    }
}
```

Subscribing to location updates

A Xamarin.Android application can also subscribe to location updates from the fused location provider using the `FusedLocationProviderClient.RequestLocationUpdatesAsync` method, as shown in this code snippet:

```
await fusedLocationProviderClient.RequestLocationUpdatesAsync(locationRequest, locationCallback);
```

This method takes two parameters:

- `Android.Gms.Location.LocationRequest` – A `LocationRequest` object is how a Xamarin.Android application passes the parameters on how the fused location provider should work. The `LocationRequest` holds information such as how frequent requests should be made or how important an accurate location update should be. For example, an important location request will cause the device to use the GPS, and consequently more power, when determining the location. This code snippet shows how to create a `LocationRequest` for a location with high accuracy, checking approximately every five minutes for a location update (but not sooner than two minutes between requests). The fused location provider will use a `LocationRequest` as guidance for which location provider to use when trying to determine the device location:

```
LocationRequest locationRequest = new LocationRequest()
                                  .SetPriority(LocationRequest.PriorityHighAccuracy)
                                  .SetInterval(60 * 1000 * 5)
                                  .SetFastestInterval(60 * 1000 * 2);
```

- `Android.Gms.Location.LocationCallback` – In order to receive location updates, a Xamarin.Android application must subclass the `LocationProvider` abstract class. This class exposed two methods which maybe invoked by the fused location provider to update the app with location information. This will be discussed in more detail below.

To notify a Xamarin.Android application of a location update, the fused location provider will invoke the `LocationCallBack.OnLocationResult(LocationResult result)`. The `Android.Gms.Location.LocationResult` parameter will contain the update location information.

When the fused location provider detects a change in the availability of location data, it will call the `LocationProvider.OnLocationAvailability(LocationAvailability locationAvailability)` method. If the `LocationAvailability.IsLocationAvailable` property returns `true`, then it can be assumed that the device location results reported by `OnLocationResult` are as accurate and as up to date as required by the `LocationRequest`. If `IsLocationAvailable` is false, then no location results will be return by `OnLocationResult`.

This code snippet is a sample implementation of the `LocationCallback` object:

```
public class FusedLocationProviderCallback : LocationCallback
{
    readonly MainActivity activity;

    public FusedLocationProviderCallback(MainActivity activity)
    {
        this.activity = activity;
    }

    public override void OnLocationAvailability(LocationAvailability locationAvailability)
    {
        Log.Debug("FusedLocationProviderSample", "IsLocationAvailable:
{0}",locationAvailability.IsLocationAvailable);
    }

    public override void OnLocationResult(LocationResult result)
    {
        if (result.Locations.Any())
        {
            var location = result.Locations.First();
            Log.Debug("Sample", "The latitude is :" + location.Latitude);
        }
        else
        {
            // No locations to work with.
        }
    }
}
```

Using the Android Location Service API

The Android Location Service is an older API for using location information on Android. Location data is collected by hardware sensors and collected by a system service, which is accessed in the application with a `LocationManager` class and an `ILocationListener`.

The Location Service is best suited for applications that must run on devices that do not have Google Play Services installed.

The Location Service is a special type of Service managed by the System. A System Service interacts with the device hardware and is always running. To tap into location updates in our application, we will subscribe to location updates from the system Location Service using a `LocationManager` and a `RequestLocationUpdates` call.

To obtain the user's location using Android Location Service involves several steps:

1. Get a reference to the `LocationManager` service.
2. Implement the `ILocationListener` interface and handle events when the location changes.
3. Use the `LocationManager` to request location updates for a specified provider. The `ILocationListener` from the previous step will be used to receive callbacks from the `LocationManager`.
4. Stop location updates when the application it is no longer appropriate to receive updates.

Location Manager

We can access the system location service with an instance of the `LocationManager` class. `LocationManager` is a special class that lets us interact with the system location Service and call methods on it. An application can get a reference to the `LocationManager` by calling `GetSystemService` and passing in a Service type, as shown below:

```
LocationManager locationManager = (LocationManager) GetSystemService(Context.LocationService);
```

`OnCreate` is a good place to get a reference to the `LocationManager`. It's a good idea to keep the `LocationManager` as a class variable, so that we can call it at various points in the Activity lifecycle.

Request location updates from the LocationManager

Once the application has a reference to the `LocationManager`, it needs to tell the `LocationManager` what type of location information that are required, and how often that information is to be updated. Do this by calling `RequestLocationUpdates` on the `LocationManager` object, and passing in some criteria for updates and a callback that will receive the location updates. This callback is a type that must implement the `ILocationListener` interface (described in more detail later in this guide).

The `RequestLocationUpdates` method tells the system location Service that your application would like to start receiving location updates. This method allows you to specify the provider, as well as time and distance thresholds to control update frequency. For example, the method below requests location updates from the GPS location provider every 2000 milliseconds, and only when the location changes more than 1 metre:

```
// For this example, this method is part of a class that implements ILocationListener, described below
locationManager.RequestLocationUpdates(LocationManager.GpsProvider, 2000, 1, this);
```

An application should request location updates only as often as required for the application to perform well. This preserves battery life and creates a better experience for the user.

Responding to updates from the LocationManager

Once an application has requested updates from the `LocationManager`, it can receive information from the Service by implementing the `ILocationListener` interface. This interface provides four methods for listening to the location Service and the location provider, `OnLocationChanged`. The System will call `OnLocationChanged` when the user's location changes enough to qualify as a location change according to the Criteria set when requesting location updates.

The following code shows the methods in the `ILocationListener` interface:

```
public class MainActivity : AppCompatActivity, ILocationListener
{
    TextView latitude;
    TextView longitude;

    public void OnLocationChanged (Location location)
    {
        // called when the location has been updated.
    }

    public OnProviderDisabled(string locationProvider)
    {
        // called when the user disables the provider
    }

    public OnProviderEnabled(string locationProvider)
    {
        // called when the user enables the provider
    }

    public OnStatusChanged(string locationProvider, Availability status, Bundle extras)
    {
        // called when the status of the provider changes (there are a variety of reasons for this)
    }
}
```

Unsubscribing to LocationManager updates

In order to conserve system resources, an application should unsubscribe to location updates as soon as possible. The `RemoveUpdates` method tells the `LocationManager` to stop sending updates to our application. As an example, an Activity may call `RemoveUpdates` in the `OnPause` method so that we are able to conserve power if an application doesn't need location updates while its Activity is not on the screen:

```
protected override void OnPause ()
{
    base.OnPause ();
    locationManager.RemoveUpdates (this);
}
```

If your application needs to get location updates while in the background, you'll want to create a custom Service that subscribes to the system Location Service. Refer to the Backgrounding with Android Services guide for more information.

Determining the best location provider for the LocationManager

The application above sets GPS as the location provider. However, GPS may not be available in all cases, such as if the device is indoors or does not have a GPS receiver. If this is the case, the result is a `null` return for the Provider.

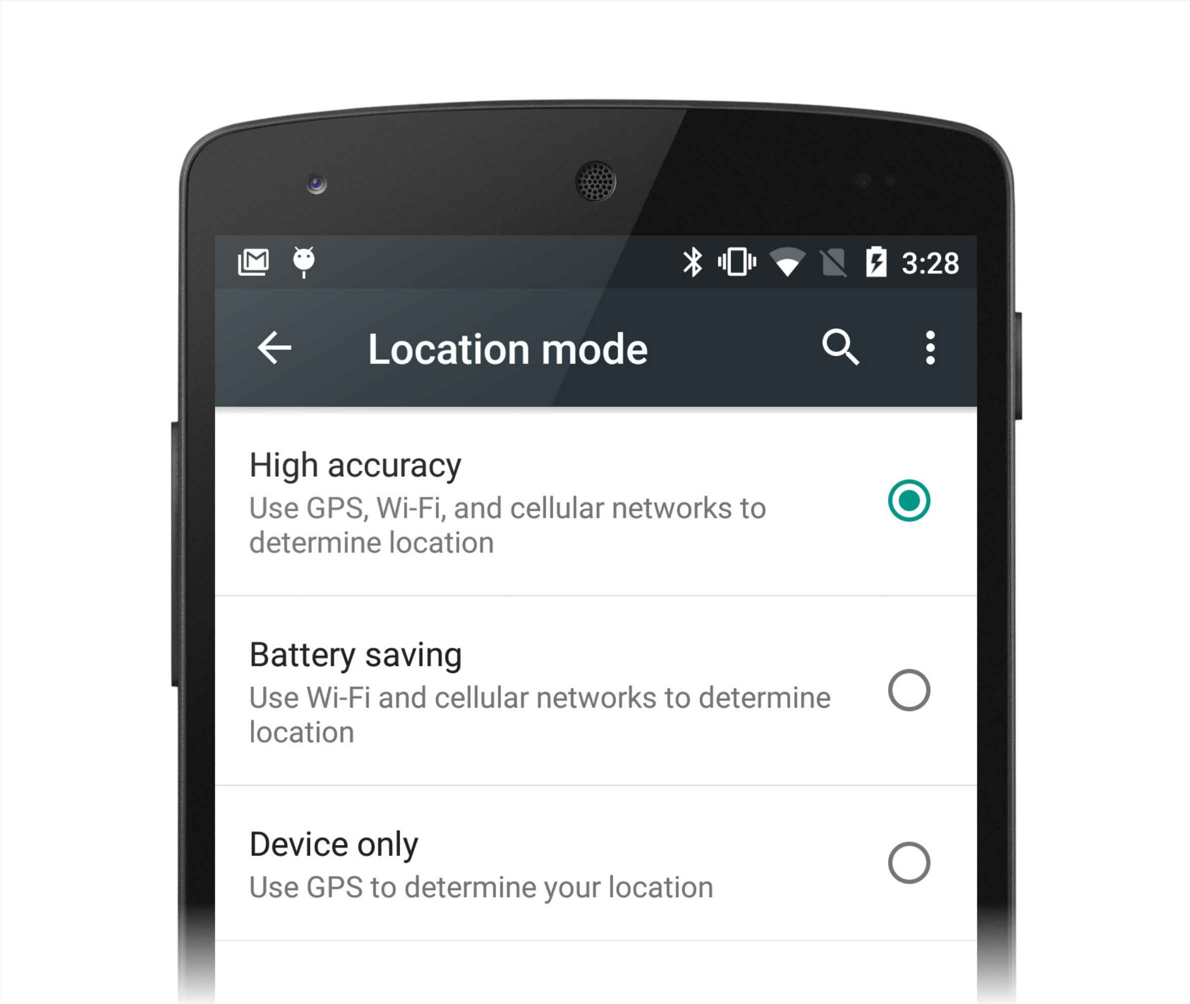
To get your app to work when GPS is not available, you use the `GetBestProvider` method to ask for the best available (device-supported and user-enabled) location provider at application launch. Instead of passing in a specific provider, you can tell `GetBestProvider` the requirements for the provider - such as accuracy and power - with a `Criteria` object. `GetBestProvider` returns the best provider for the given Criteria.

The following code shows how to get the best available provider and use it when requesting location updates:

```
Criteria locationCriteria = new Criteria();
locationCriteria.Accuracy = Accuracy.Coarse;
locationCriteria.PowerRequirement = Power.Medium;

locationProvider = locationManager.GetBestProvider(locationCriteria, true);

if(locationProvider != null)
{
    locationManager.RequestLocationUpdates (locationProvider, 2000, 1, this);
}
else
{
    Log.Info(tag, "No location providers available");
}
```

NOTE

If the user has disabled all location providers, `GetBestProvider` will return `null`. To see how this code works on a real device, be sure to enable GPS, Wi-Fi, and cellular networks under **Google Settings** > **Location** > **Mode** as shown in this screenshot:



The screenshot below demonstrates the location application running using `GetBestProvider`:

Keep in mind that `GetBestProvider` does not change the provider dynamically. Rather, it determines the best available provider once during the Activity lifecycle. If the provider status changes after it has been set, the application will require additional code in the `ILocationListener` methods – `OnProviderEnabled`, `OnProviderDisabled`, and `OnStatusChanged` – to handle every possibility related to the provider switch.

Summary

This guide covered obtaining the user's location using both the Android Location Service and the fused location provider from Google Location Services API.

Related links

- Location (sample)
- FusedLocationProvider (sample)
- Google Play Services
- Criteria Class
- LocationManager Class
- LocationListener Class
- LocationClient API
- LocationListener API
- LocationRequest API

How to use Google Maps and Location with Xamarin.Android

11/2/2020 • 2 minutes to read • Edit Online

This article discusses how to use maps and location with Xamarin.Android. It covers everything from leveraging the built-in maps application to using the Google Maps Android API V2 directly.

Maps Overview

Mapping technologies are a ubiquitous complement to mobile devices. Desktop computers and laptops don't tend to have location awareness built-in. On the other hand, mobile devices use such applications to locate devices and to display changing location information. Android has powerful, built-in technology that displays location data on maps using location hardware that may be available on the device. This article covers a spectrum of what the maps applications under Xamarin.Android have to offer, including:

- Using the built-in maps application to quickly add mapping functionality.
- Working with the Maps API to control a map's display.
- Using a variety of techniques to add graphical overlays.

The topics in this section cover a wide range of mapping features. First, they explain how to leverage Android's built-in maps application and how to display a panoramic street view of a location. Then they discuss how to use the Maps API to incorporate mapping features directly within an application, covering both how to control the position and display of a map, as well as how to add graphical overlays.

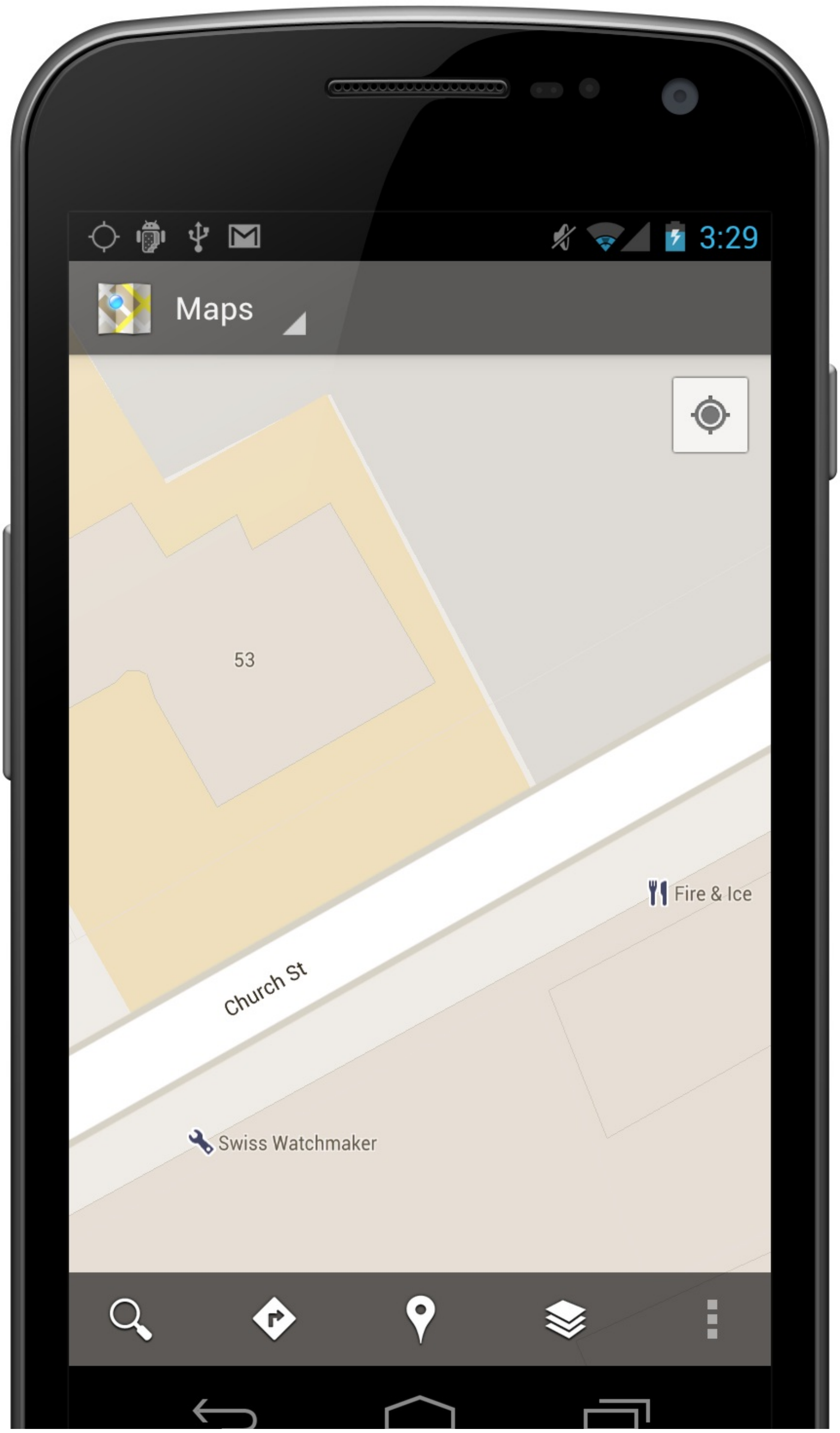
Related Links

- MapsAndLocationDemo_v3 (sample)
- Activity Lifecycle
- Obtaining a Google Maps API Key
- Intents List: Invoking Google Applications on Android Devices
- Location and Maps

Launching the Maps Application

7/8/2021 • 2 minutes to read • Edit Online

The simplest way to work with maps in Xamarin.Android is to leverage the built-in maps application shown below:



When you use the maps application, the map will not be part of your application. Instead, your application will launch the maps application and load the map externally. The next section examines how to use Xamarin.Android to launch maps like the one above.

Creating the Intent

Working with the maps application is as easy as creating an Intent with an appropriate URI, setting the action to ActionView, and calling the StartActivity method. For example, the following code launches the maps application centered at a given latitude and longitude:

```
var geoUri = Android.Net.Uri.Parse ("geo:42.374260,-71.120824");
var mapIntent = new Intent (Intent.ActionView, geoUri);
StartActivity (mapIntent);
```

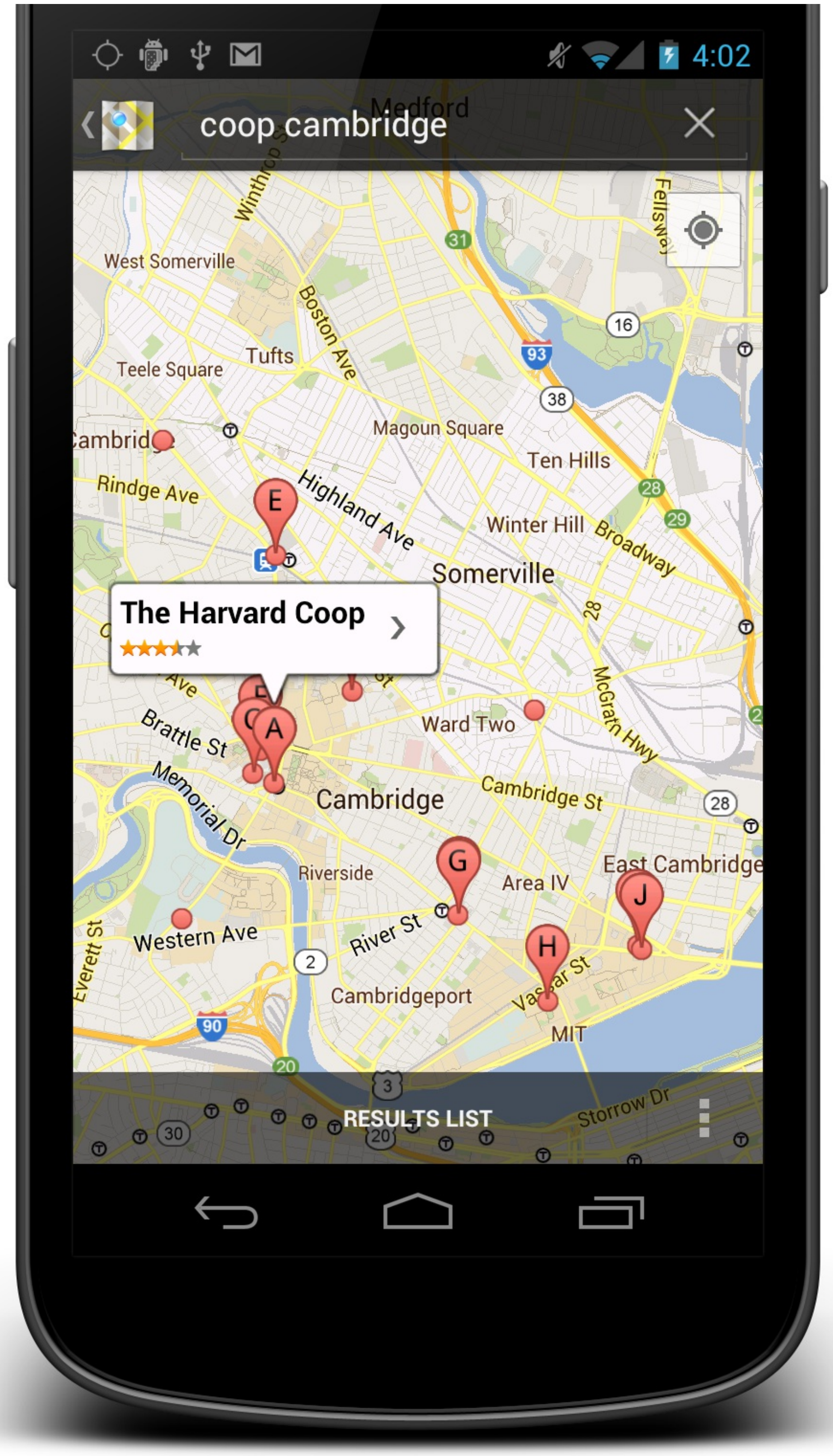
This code is all that is needed to launch the map shown in the previous screenshot. In addition to specifying latitude and longitude, the URI scheme for maps supports several other options.

Geo URI Scheme

The code above used the geo scheme to create a URI. This URI scheme supports several formats, as listed below:

- `geo:latitude,longitude` – Opens the maps application centered at a lat/lon.
- `geo:latitude,longitude?z=zoom` – Opens the maps application centered at a lat/lon and zoomed to the specified level. The zoom level can range from 1 to 23: 1 displays the entire Earth and 23 is the closest zoom level.
- `geo:0,0?q=my+street+address` – Opens the maps application to the location of a street address.
- `geo:0,0?q=business+near+city` – Opens the maps application and displays the annotated search results.

The versions of the URI that take a query (namely the street address or search terms) use Google's geocoder service to retrieve the location that is then displayed on the map. For example, the URI `geo:0,0?q=coop+Cambridge` results in the map shown below:



For more information about geo URI schemes, see Show a location on a map.

Street View

In addition to the geo scheme, Android also supports loading street views from an Intent. An example of the street view application launched from Xamarin.Android is shown below:


4:42
53 Church Street
Church Street (NE)
Google
© 2011 Google

To launch a street view, simply use the `google.streetview` URI scheme, as demonstrated in the following code:

```
var streetViewUri = Android.Net.Uri.Parse (
       "google.streetview:cbll=42.374260,-71.120824&cbp=1,90,,0,1.0&mz=20");
var streetViewIntent = new Intent (Intent.ActionView, streetViewUri);
StartActivity (streetViewIntent);
```

The google.streetview URI scheme used above takes the following form:

```
google.streetview:cbll=lat,lng&cbp=1,yaw,,pitch,zoom&mz=mapZoom
```

As you can see, there are several parameters supported, as listed below:

- `lat` – The latitude of the location to be shown in the street view.
- `lng` – The longitude of the location to be shown in the street view.
- `pitch` – Angle of street view panorama, measured from the center in degrees where 90 degrees is straight down and -90 degrees is straight up.
- `yaw` – Center-of-view of street view panorama, measured clockwise in degrees from North.
- `zoom` – Zoom multiplier for street view panorama, where 1.0 = normal zoom, 2.0 = zoomed 2x, 3.0 = zoomed 4x, etc.
- `mz` – The map zoom level that will be used when going to the maps application from the street view.

Working with the built-in maps application or the street view is an easy way to quickly add mapping support. However, Android's Maps API offers finer control over the mapping experience.

Using the Google Maps API in your application

7/8/2021 • 18 minutes to read • Edit Online

Using the Maps application is great, but sometimes you want to include maps directly in your application. In addition to the built-in maps application, Google also offers a native mapping API for Android. The Maps API is suitable for cases where you want to maintain more control over the mapping experience. Things that are possible with the Maps API include:

- Programmatically changing the viewpoint of the map.
- Adding and customizing markers.
- Annotating a map with overlays.

Unlike the now-deprecated Google Maps Android API v1, Google Maps Android API v2 is part of Google Play Services. A Xamarin.Android app must meet some mandatory prerequisites before it is possible to use the Google Maps Android API.

Google Maps API prerequisites

Several steps need to be taken before you can use the Maps API, including:

- Obtain a Maps API key
- Install the Google Play Services SDK
- Install the Xamarin.GooglePlayServices.Maps package from NuGet
- Specify the required permissions
- Optionally, Create an emulator with the Google APIs

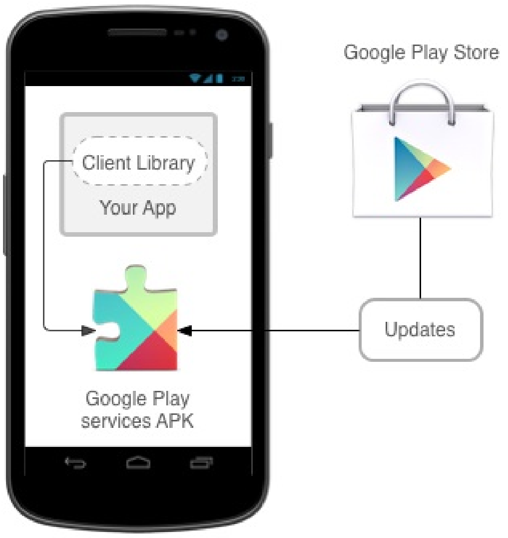
Obtain a Google Maps API Key

The first step is to get a Google Maps API key (note that you cannot reuse an API key from the legacy Google Maps v1 API). For information about how to obtain and use the API key with Xamarin.Android, see Obtaining A Google Maps API Key.

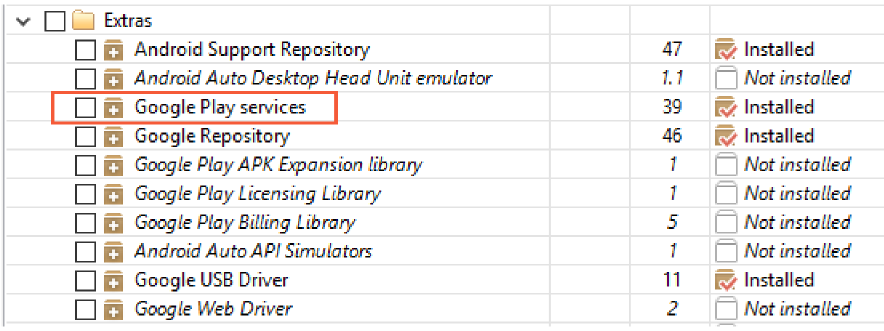
Install the Google Play Services SDK

Google Play Services is a technology from Google that allows Android applications to take advantage of various Google features such as Google+, In-App Billing, and Maps. These features are accessible on Android devices as background services, which are contained in the Google Play Services APK.

Android applications interact with Google Play Services through the Google Play Services client library. This library contains the interfaces and classes for the individual services such as Maps. The following diagram shows the relationship between an Android application and Google Play Services:



The Android Maps API is provided as a part of Google Play Services. Before a Xamarin.Android application can use the Maps API, the Google Play Services SDK must be installed using the Android SDK Manager. The following screenshot shows where in the Android SDK Manager the Google Play services client can be found:

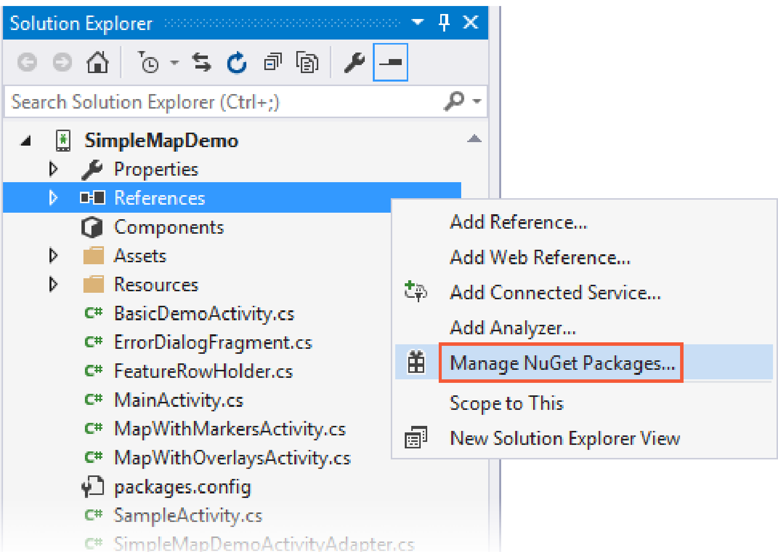


NOTE

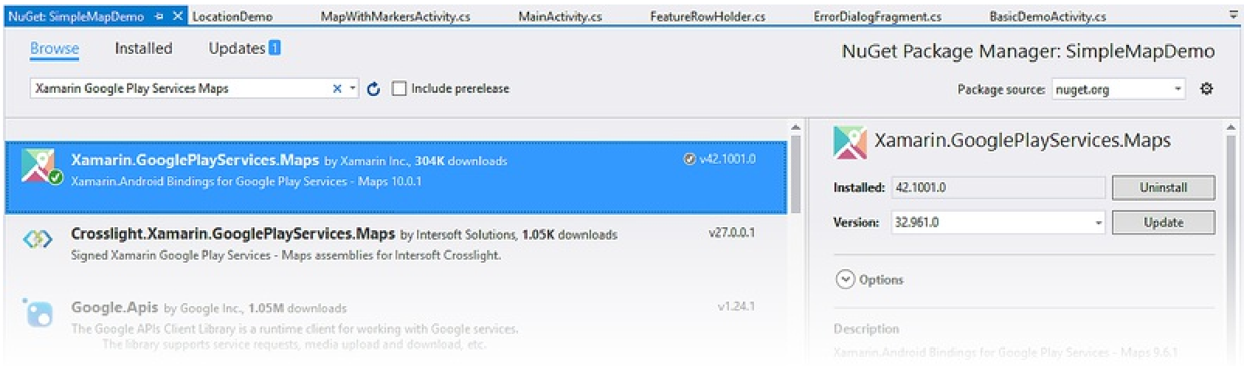
The Google Play services APK is a licensed product that may not be present on all devices. If it is not installed, then Google Maps will not work on the device.

Install the Xamarin.GooglePlayServices.Maps package from NuGet

The Xamarin.GooglePlayServices.Maps package contains the Xamarin.Android bindings for the Google Play Services Maps API. To add the Google Play Services Map package, right-click the **References** folder of your project in the Solution Explorer and click **Manage NuGet Packages...**:



This opens the **NuGet Package Manager**. Click **Browse** and enter **Xamarin Google Play Services Maps** in the search field. Select **Xamarin.GooglePlayServices.Maps** and click **Install**. (If this package had been installed previously, click **Update**.):



Notice that the following dependency packages are also installed:

- **Xamarin.GooglePlayServices.Base**
- **Xamarin.GooglePlayServices.Basement**
- **Xamarin.GooglePlayServices.Tasks**

Specify the required permissions

Apps must identify the hardware and permission requirements in order to use the Google Maps API. Some permissions are automatically granted by the Google Play Services SDK, and it is not necessary for a developer to explicitly add them to **AndroidManfest.XML**:

- **Access to the Network State** – The Maps API must be able to check if it can download the map tiles.

- **Internet Access** – Internet access is necessary to download the map tiles and communicate with the Google Play Servers for API access.

The following permissions and features must be specified in the **AndroidManifest.XML** for the Google Maps Android API:

- **OpenGL ES v2** – The application must declare the requirement for OpenGL ES v2.

- **Google Maps API Key** – The API key is used to confirm that the application is registered and authorized to use Google Play Services. See Obtaining a Google Maps API Key for details about this key.

- **Request the legacy Apache HTTP client** – Apps that target Android 9.0 (API level 28) or above must specify that the legacy Apache HTTP client is an optional library to use.

- **Access to the Google Web-based Services** – The application needs permissions to access Google's web services that back the Android Maps API.

- **Permissions for Google Play Services Notifications** – The application must be granted permission to receive remote notifications from Google Play Services.

- **Access to Location Providers** – These are optional permissions. They will allow the `GoogleMap` class to display the location of the device on the map.

In addition, Android 9 has removed the Apache HTTP client library from the bootclasspath, and so it isn't available to applications that target API 28 or higher. The following line must be added to the `application` node of your **AndroidManifest.xml** file to continue using the Apache HTTP client in applications that target API 28 or higher:

```
<application ...>
   ...
   <uses-library android:name="org.apache.http.legacy" android:required="false" />
</application>
```

NOTE

Very old versions of the Google Play SDK required an app to request the `WRITE_EXTERNAL_STORAGE` permission. This requirement is no longer necessary with the recent Xamarin bindings for Google Play Services.

The following snippet is an example of the settings that must be added to **AndroidManifest.XML**:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionName="4.5"
package="com.xamarin.docs.android.mapsandlocationdemo2" android:versionCode="6">
    <uses-sdk android:minSdkVersion="23" android:targetSdkVersion="28" />

    <!-- Google Maps for Android v2 requires OpenGL ES v2 -->
    <uses-feature android:glEsVersion="0x00020000" android:required="true" />

    <!-- Necessary for apps that target Android 9.0 or higher -->
    <uses-library android:name="org.apache.http.legacy" android:required="false" />

    <!-- Permission to receive remote notifications from Google Play Services -->
    <!-- Notice here that we have the package name of our application as a prefix on the permissions. -->
    <uses-permission android:name="<PACKAGE NAME>.permission.MAPS_RECEIVE" />
    <permission android:name="<PACKAGE NAME>.permission.MAPS_RECEIVE" android:protectionLevel="signature" />

    <!-- These are optional, but recommended. They will allow Maps to use the My Location provider. -->
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />

    <application android:label="@string/app_name">
        <!-- Put your Google Maps V2 API Key here. -->
        <meta-data android:name="com.google.android.maps.v2.API_KEY" android:value="YOUR_API_KEY" />
        <meta-data android:name="com.google.android.gms.version"
android:value="@integer/google_play_services_version" />
        <!-- Necessary for apps that target Android 9.0 or higher -->
        <uses-library android:name="org.apache.http.legacy" android:required="false" />
    </application>
</manifest>
```

In addition to requesting the permissions **AndroidManifest.XML**, an app must also perform runtime permission checks for the `ACCESS_COARSE_LOCATION` and the `ACCESS_FINE_LOCATION` permissions. See the Xamarin.Android Permissions guide for more information about performing run-time permission checks.

Create an Emulator with Google APIs

In the event that a physical Android device with Google Play services is not installed, it is possible to create an emulator image for development. For more information see the Device Manager.

The GoogleMap Class

Once the prerequisites are satisfied, it is time to start developing the application and use the Android Maps API. The GoogleMap class is the main API that a Xamarin.Android application will use to display and interact with a Google Maps for Android. This class has the following responsibilities:

- Interacting with Google Play services to authorize the application with the Google web service.
- Downloading, caching, and displaying the map tiles.

- Displaying UI controls such as pan and zoom to the user.
- Drawing markers and geometric shapes on maps.

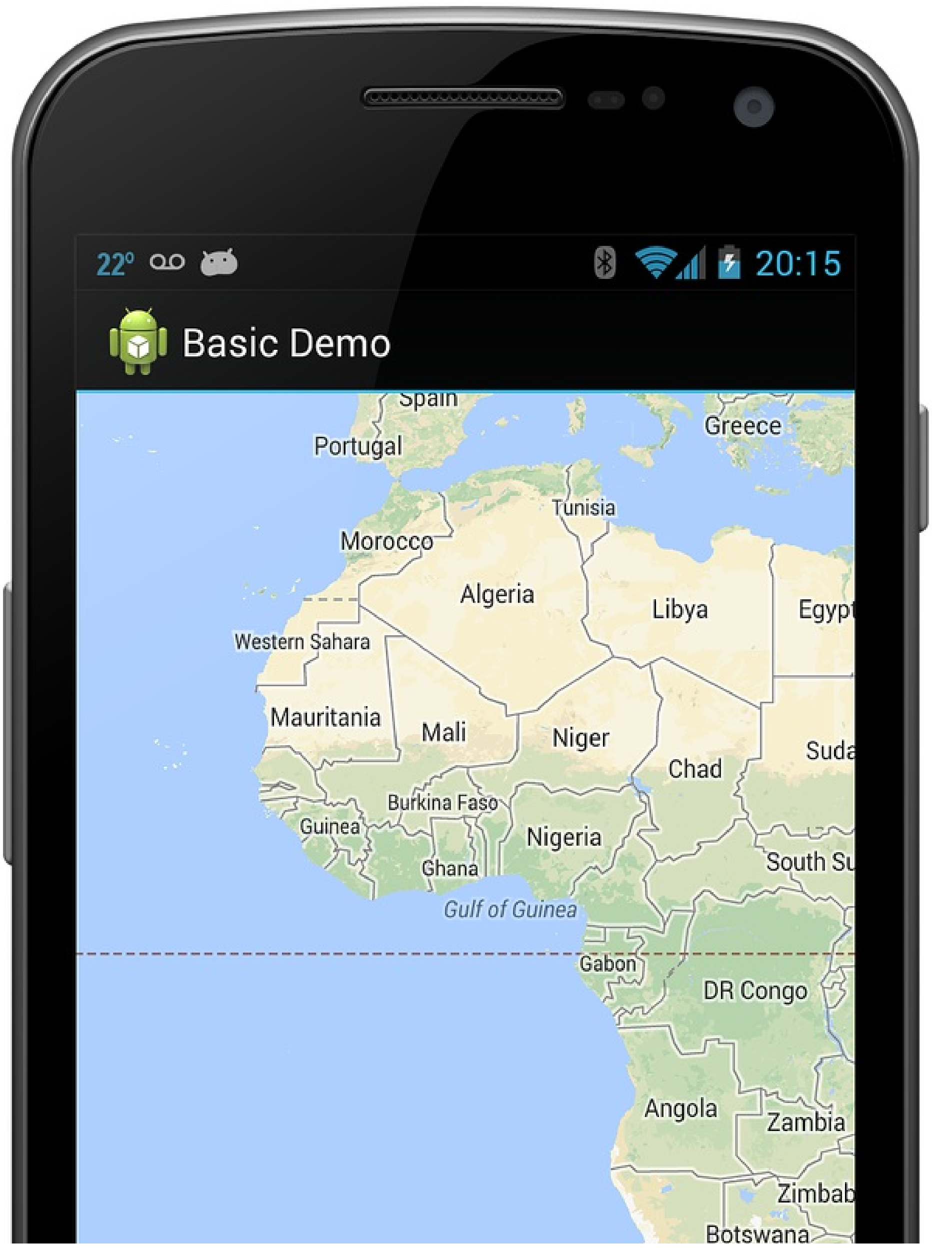
The `GoogleMap` is added to an Activity in one of two ways:

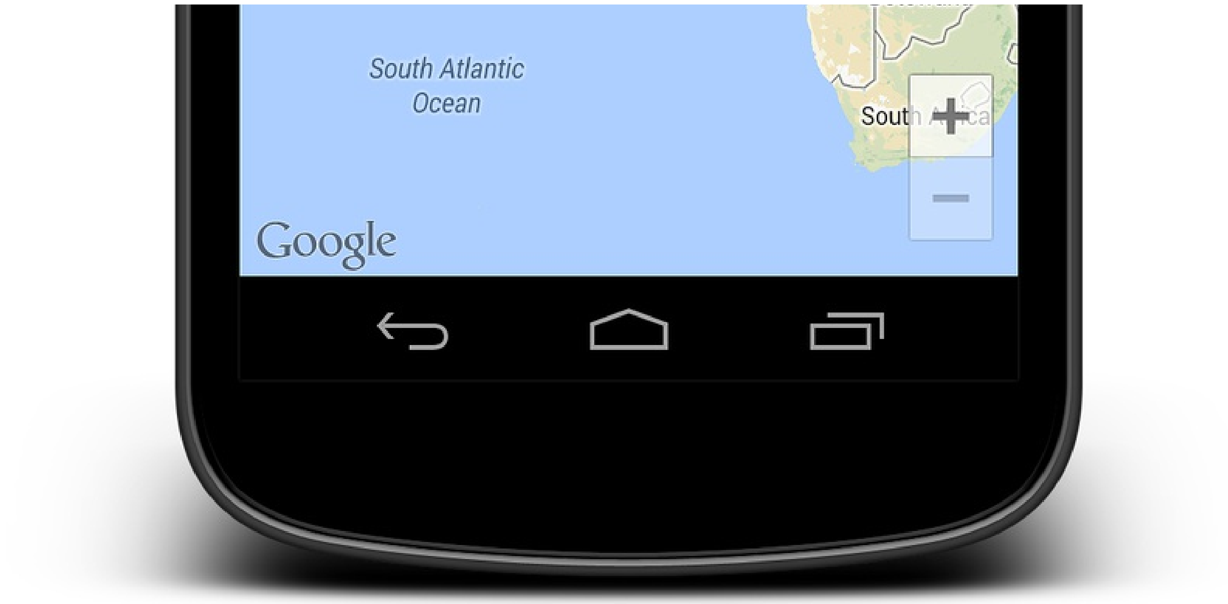
- **MapFragment** - The MapFragment is a specialized Fragment that acts as host for the `GoogleMap` object. The `MapFragment` requires Android API level 12 or higher. Older versions of Android can use the SupportMapFragment. This guide will focus on using the `MapFragment` class.
- **MapView** - The MapView is a specialized View subclass, which can act as a host for a `GoogleMap` object. Users of this class must forward all of the Activity lifecycle methods to the `MapView` class.

Each of these containers exposes a `Map` property that returns an instance of `GoogleMap`. Preference should be given to the MapFragment class as it is a simpler API that reduces the amount boilerplate code that a developer must manually implement.

Adding a MapFragment to an Activity

The following screenshot is an example of a simple `MapFragment`:





Similar to other Fragment classes, there are two ways to add a `MapFragment` to an Activity:

- **Declaratively** - The `MapFragment` can be added via the XML layout file for the Activity. The following XML snippet shows an example of how to use the `fragment` element:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
          android:id="@+id/map"
          android:layout_width="match_parent"
          android:layout_height="match_parent"
          class="com.google.android.gms.maps.MapFragment" />
```

- **Programmatically** - The `MapFragment` can be programmatically instantiated using the `MapFragment.NewInstance` method and then added to an Activity. This snippet shows the simplest way to instantiate a `MapFragment` object and add to an Activity:

```
    var mapFrag = MapFragment.NewInstance();
    activity.FragmentManager.BeginTransaction()
                            .Add(Resource.Id.map_container, mapFrag, "map_fragment")
                            .Commit();
```

  It is possible to configure the `MapFragment` object by passing a `GoogleMapOptions` object to `NewInstance`. This is discussed in the section GoogleMap properties that appears later on in this guide.

The `MapFragment.GetMapAsync` method is used to initialize the `GoogleMap` that is hosted by the fragment and obtain a reference to the map object that is hosted by the `MapFragment`. This method takes an object that implements the `IOnMapReadyCallback` interface.

This interface has a single method, `IMapReadyCallback.OnMapReady(MapFragment map)` that will be invoked when it is possible for the app to interact with the `GoogleMap` object. The following code snippet shows how an Android Activity can initialize a `MapFragment` and implement the `IOnMapReadyCallback` interface:

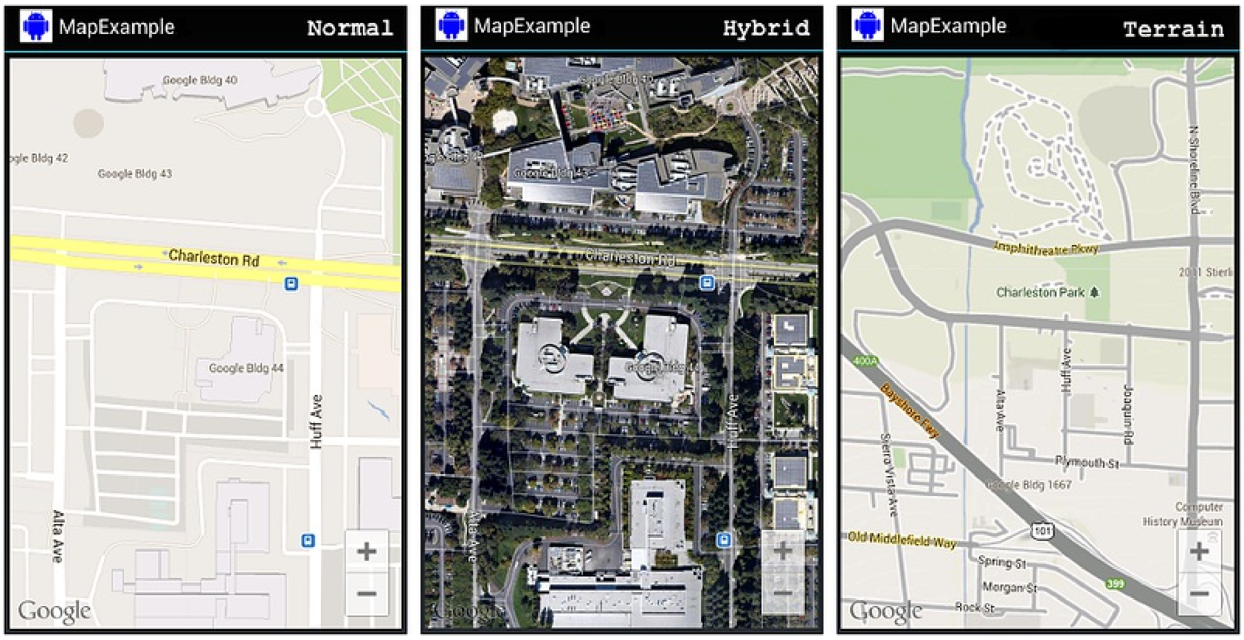
```
public class MapWithMarkersActivity : AppCompatActivity, IOnMapReadyCallback
{
    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);
        SetContentView(Resource.Layout.MapLayout);

        var mapFragment = (MapFragment) FragmentManager.FindFragmentById(Resource.Id.map);
        mapFragment.GetMapAsync(this);

        // remainder of code omitted
    }

    public void OnMapReady(GoogleMap map)
    {
        // Do something with the map, i.e. add markers, move to a specific location, etc.
    }
}
```

Map types

There are five different types of maps available from the Google Maps API:

- **Normal** - This is the default map type. It shows roads and important natural features along with some artificial points of interest (such as buildings and bridges).
- **Satellite** - This map shows satellite photography.
- **Hybrid** - This map shows satellite photography and road maps.
- **Terrain** - This primarily shows topographical features with some roads.
- **None** - This map does not load any tiles, it is rendered as an empty grid.

The image below shows three of the different types of maps, from left-to-right (normal, hybrid, terrain):



The `GoogleMap.MapType` property is used to set or change which type of map is displayed. The following code snippet shows how to display a satellite map.

```
public void OnMapReady(GoogleMap map)
{
    map.MapType = GoogleMap.MapTypeHybrid;
}
```

GoogleMap properties

`GoogleMap` defines several properties that can control the functionality and the appearance of the map. One way to configure the initial state of a `GoogleMap` is to pass a GoogleMapOptions object when creating a `MapFragment`. The following code snippet is one example of using a `GoogleMapOptions` object when creating a `MapFragment`:

```
GoogleMapOptions mapOptions = new GoogleMapOptions()
    .InvokeMapType(GoogleMap.MapTypeSatellite)
    .InvokeZoomControlsEnabled(false)
    .InvokeCompassEnabled(true);

FragmentTransaction fragTx = FragmentManager.BeginTransaction();
mapFragment = MapFragment.NewInstance(mapOptions);
fragTx.Add(Resource.Id.map, mapFragment, "map");
fragTx.Commit();
```

The other way to configure a `GoogleMap` is by manipulating properties on the UiSettings of the map object. The next code sample shows how to configure a `GoogleMap` to display the zoom controls and a compass:

```
public void OnMapReady(GoogleMap map)
{
    map.UiSettings.ZoomControlsEnabled = true;
    map.UiSettings.CompassEnabled = true;
}
```

Interacting with the GoogleMap

The Android Maps API provides APIs that allow an Activity to change the viewpoint, add markers, place custom overlays, or draw geometric shapes. This section will discuss how to accomplish some of these tasks in Xamarin.Android.

Changing the Viewpoint

Maps are modelled as a flat plane on the screen, based on the Mercator projection. The map view is that of a *camera* looking straight down on this plane. The position of the camera can be controlled by changing the location, zoom, tilt, and bearing. The CameraUpdate class is used to move the camera location. `CameraUpdate` objects are not directly instantiated, instead the Maps API provides the CameraUpdateFactory class.

Once a `CameraUpdate` object has been created, it is passed as a parameter to either the GoogleMap.MoveCamera or GoogleMap.AnimateCamera methods. The `MoveCamera` method updates the map instantly while the `AnimateCamera` method provides a smooth, animated transition.

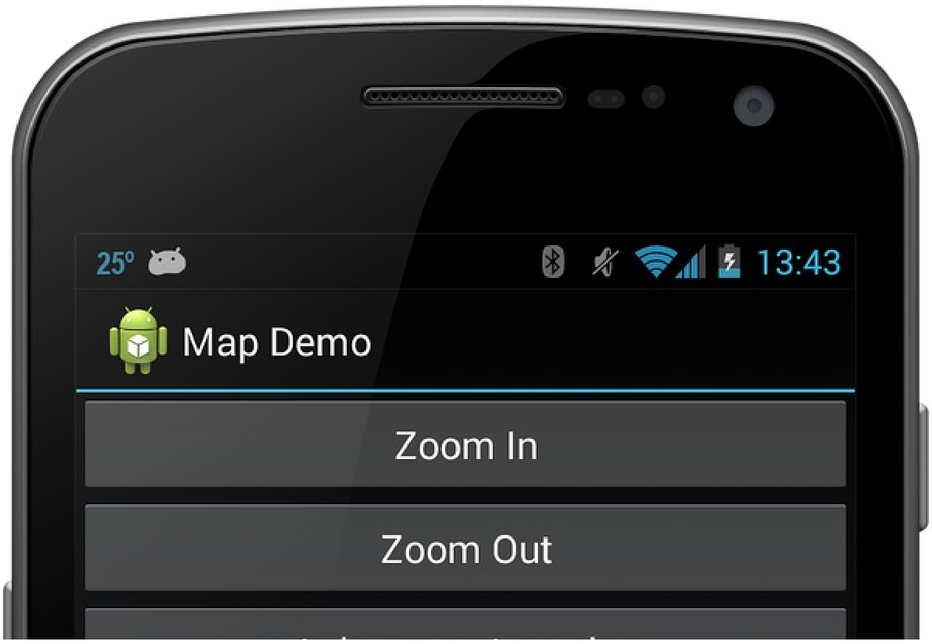
This code snippet is a simple example of how to use the `CameraUpdateFactory` to create a `CameraUpdate` that will increment the zoom level of the map by one zoom level:

```
MapFragment mapFrag = (MapFragment) FragmentManager.FindFragmentById(Resource.Id.my_mapfragment_container);
mapFrag.GetMapAsync(this);
...

public void OnMapReady(GoogleMap map)
{
    map.MoveCamera(CameraUpdateFactory.ZoomIn());
}
```

The Maps API provides a CameraPosition which will aggregate all of the possible values for the camera position. An instance of this class can be provided to the CameraUpdateFactory.NewCameraPosition method which will return a `CameraUpdate` object. The Maps API also includes the CameraPosition.Builder class that provides a fluent API for creating `CameraPosition` objects. The following code snippet shows an example of creating a `CameraUpdate` from a `CameraPosition` and using that to change the camera position on a `GoogleMap`:

```
public void OnMapReady(GoogleMap map)
{
    LatLng location = new LatLng(50.897778, 3.013333);

    CameraPosition.Builder builder = CameraPosition.InvokeBuilder();
    builder.Target(location);
    builder.Zoom(18);
    builder.Bearing(155);
    builder.Tilt(65);

    CameraPosition cameraPosition = builder.Build();

    CameraUpdate cameraUpdate = CameraUpdateFactory.NewCameraPosition(cameraPosition);

    map.MoveCamera(cameraUpdate);
}
```

In the previous code snippet, a specific location on the map is represented by the LatLng class. The zoom level is set to 18, which is an arbitrary measure of zoom used by Google Maps. The bearing is the compass measurement clockwise from North. The Tilt property controls the viewing angle and specifies an angle of 25 degrees from the vertical. The following screenshot shows the `GoogleMap` after executing the preceding code:



Drawing on the Map

The Android Maps API provides API's for drawing the following items on a map:

- **Markers** - These are special icons that are used to identify a single location on a map.
- **Overlays** - This is an image that can be used to identify a collection of locations or area on the map.
- **Lines, Polygons, and Circles** - These are APIs that allow Activities to add shapes to a map.

Markers

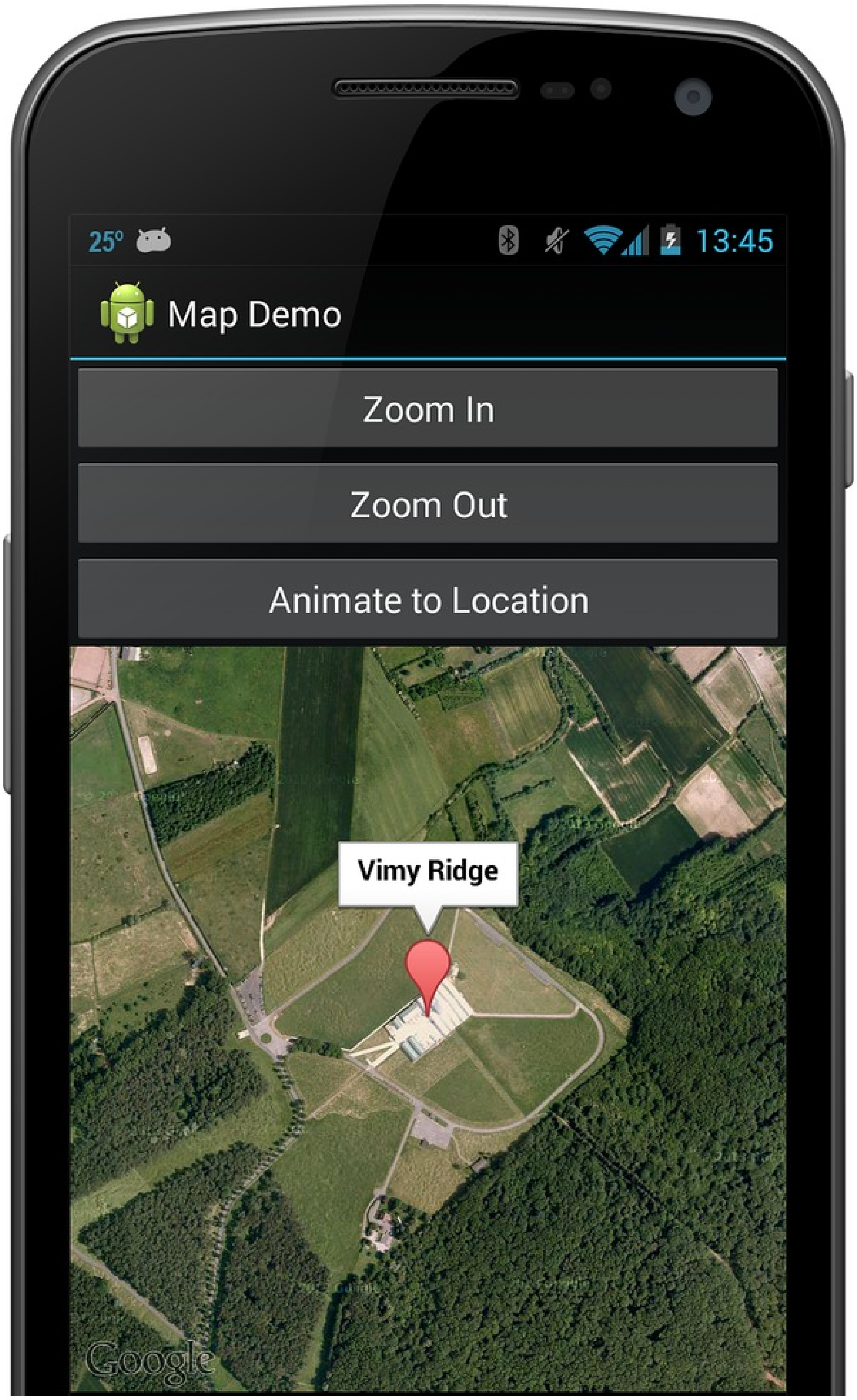
The Maps API provides a Marker class which encapsulates all of the data about a single location on a map. By default the Marker class uses a standard icon provided by Google Maps. It is possible to customize the appearance of a marker and to respond to user clicks.

Adding a Marker

To add a marker to a map, it is necessary create a new MarkerOptions object and then call the AddMarker method on a `GoogleMap` instance. This method will return a Marker object.

```
public void OnMapReady(GoogleMap map)
{
    MarkerOptions markerOpt1 = new MarkerOptions();
    markerOpt1.SetPosition(new LatLng(50.379444, 2.773611));
    markerOpt1.SetTitle("Vimy Ridge");

    map.AddMarker(markerOpt1);
}
```

The title of the marker will be displayed in an *info window* when the user taps on the marker. The following screenshot shows what this marker looks like:



Customizing A Marker

It is possible to customize the icon used by the marker by calling the `MarkerOptions.InvokeIcon` method when adding the marker to the map. This method takes a BitmapDescriptor object containing the data necessary to render the icon. The BitmapDescriptorFactory class provides some helper methods to simplify the creation of a `BitmapDescriptor`. The following list introduces some of these methods:

- `DefaultMarker(float colour)` – Use the default Google Maps marker, but change the colour.
- `FromAsset(string assetName)` – Use a custom icon from the specified file in the Assets folder.
- `FromBitmap(Bitmap image)` – Use the specified bitmap as the icon.
- `FromFile(string fileName)` – Create the custom icon from the file at the specified path.
- `FromResource(int resourceId)` – Create a custom icon from the specified resource.

The following code snippet shows an example of creating a cyan coloured default marker:

```
public void OnMapReady(GoogleMap map)
{
    MarkerOptions markerOpt1 = new MarkerOptions();
    markerOpt1.SetPosition(new LatLng(50.379444, 2.773611));
    markerOpt1.SetTitle("Vimy Ridge");

    var bmDescriptor = BitmapDescriptorFactory.DefaultMarker (BitmapDescriptorFactory.HueCyan);
    markerOpt1.InvokeIcon(bmDescriptor);

    map.AddMarker(markerOpt1);
}
```

Info windows

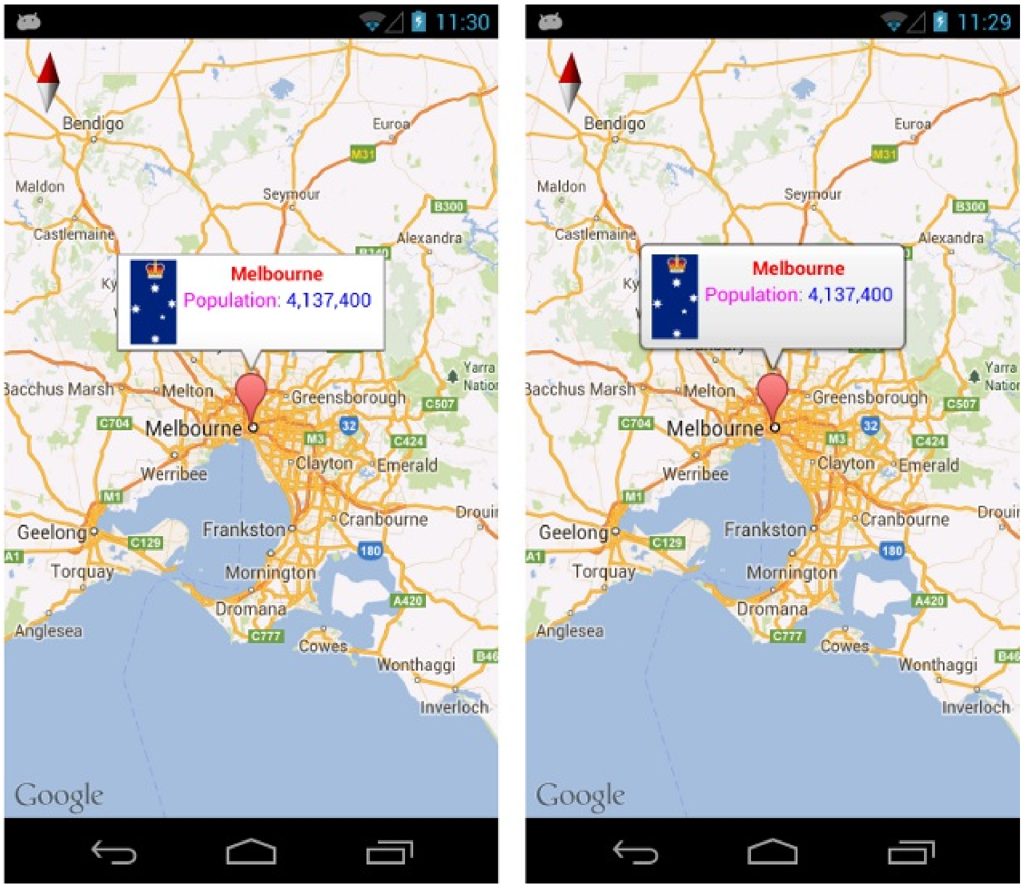
Info windows are special windows that popup to display information to the user when they tap a specific marker. By default the info window will display the contents of the marker's title. If the title has not been assigned, then no info window will appear. Only one info window may be shown at a time.

It is possible to customize the info window by implementing the GoogleMap.IInfoWindowAdapter interface. There are two important methods on this interface:

- `public View GetInfoWindow(Marker marker)` – This method is called to get a custom info window for a marker. If it returns `null`, then the default window rendering will be used. If this method returns a View, then that View will be placed inside the info window frame.
- `public View GetInfoContents(Marker marker)` – This method will only be called if GetInfoWindow returns `null`. This method can return a `null` value if the default rendering of the info window contents is to be used. Otherwise, this method should return a View with the contents of the info window.

An info window is not a live view - instead Android will convert the View to a static bitmap and display that on the image. This means that an info window cannot respond to any touch events or gestures, nor will it automatically update itself. To update an info window, it is necessary to call the GoogleMap.ShowInfoWindow method.

The following image shows some examples of some customized info windows. The image on the left has its contents customized, while the image on the right has its window and contents customized with rounded corners:



GroundOverlays

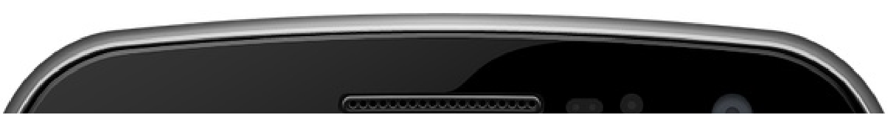
Unlike markers, which identify a specific location on a map, a GroundOverlay is an image that is used to identify a collection of locations or an area on the map.

Adding a GroundOverlay

Adding a ground overlay to a map is similar to adding a marker to a map. First, a GroundOverlayOptions object is created. This object is then passed as a parameter to the `GoogleMap.AddGroundOverlay` method, which will return a `GroundOverlay` object. This code snippet is an example of adding a ground overlay to a map:

```
BitmapDescriptor image = BitmapDescriptorFactory.FromResource(Resource.Drawable.polarbear);
GroundOverlayOptions groundOverlayOptions = new GroundOverlayOptions()
    .Position(position, 150, 200)
    .InvokeImage(image);
GroundOverlay myOverlay = googleMap.AddGroundOverlay(groundOverlayOptions);
```

The following screenshot shows this overlay on a map:



Lines, Circles, and Polygons

There are three simple types of geometric figures that can be added to a map:

- **Polyline** - This is a series of connected line segments. It can mark a path on a map or create a geometric shape.

- **Circle** - This will draw a circle on the map.
- **Polygon** - This is a closed shape for marking areas on a map.

Polylines

A Polyline is a list of consecutive `LatLng` objects which specify the vertices of each line segment. A polyline is created by first creating a `PolylineOptions` object and adding the points to it. The `PolylineOption` object is then passed to a `GoogleMap` object by calling the `AddPolyline` method.

```
PolylineOption rectOptions = new PolylineOption();
rectOptions.Add(new LatLng(37.35, -122.0));
rectOptions.Add(new LatLng(37.45, -122.0));
rectOptions.Add(new LatLng(37.45, -122.2));
rectOptions.Add(new LatLng(37.35, -122.2));
rectOptions.Add(new LatLng(37.35, -122.0)); // close the polyline - this makes a rectangle.

googleMap.AddPolyline(rectOptions);
```

Circles

Circles are created by first instantiating a CircleOption object which will specify the center and the radius of the circle in metres. The circle is drawn on the map by calling GoogleMap.AddCircle. The following code snippet shows how to draw a circle:

```
CircleOptions circleOptions = new CircleOptions ();
circleOptions.InvokeCenter (new LatLng(37.4, -122.1));
circleOptions.InvokeRadius (1000);

googleMap.AddCircle (circleOptions);
```

Polygons

`Polygon`s are similar to `Polyline`s, however they are not open ended. `Polygon`s are a closed loop and have their interior filled in. `Polygon`s are created in the exact same manner as a `Polyline`, except the GoogleMap.AddPolygon method invoked.

Unlike a `Polyline`, a `Polygon` is self-closing. The polygon will be closed off by the `AddPolygon` method by drawing a line which connects the first and last points. The following code snippet will create a solid rectangle over the same area as the previous code snippet in the `Polyline` example.

```
PolygonOptions rectOptions = new PolygonOptions();
rectOptions.Add(new LatLng(37.35, -122.0));
rectOptions.Add(new LatLng(37.45, -122.0));
rectOptions.Add(new LatLng(37.45, -122.2));
rectOptions.Add(new LatLng(37.35, -122.2));
// notice we don't need to close off the polygon

googleMap.AddPolygon(rectOptions);
```

Responding to user events

There are three types of interactions a user may have with a map:

- **Marker Click** - The user clicks on a marker.
- **Marker Drag** - The user has long-clicked on a mparger
- **Info Window Click** - The user has clicked on an info window.

Each of these events will be discussed in more detail below.

Marker click events

The `MarkerClicked` event is raised when the user taps on a marker. This event accepts a `GoogleMap.MarkerClickEventArgs` object as a parameter. This class contains two properties:

- `GoogleMap.MarkerClickEventArgs.Handled` – This property should be set to `true` to indicate that the event handler has consumed the event. If this is set to `false` then the default behaviour will occur in addition to the custom behaviour of the event handler.
- `Marker` – This property is a reference to the marker that raised the `MarkerClick` event.

This code snippet shows an example of a `MarkerClick` that will change the camera position to a new location on the map:

```
void MapOnMarkerClick(object sender, GoogleMap.MarkerClickEventArgs markerClickEventArgs)
{
    markerClickEventArgs.Handled = true;

    var marker = markerClickEventArgs.Marker;
    if (marker.Id.Equals(gotMauiMarkerId))
    {
        LatLng InMaui = new LatLng(20.72110, -156.44776);

        // Move the camera to look at Maui.
        PositionPolarBearGroundOverlay(InMaui);
        googleMap.AnimateCamera(CameraUpdateFactory.NewLatLngZoom(InMaui, 13));
        gotMauiMarkerId = null;
        polarBearMarker.Remove();
        polarBearMarker = null;
    }
    else
    {
        Toast.MakeText(this, $"You clicked on Marker ID {marker.Id}", ToastLength.Short).Show();
    }
}
```

Marker Drag events

This event is raised when the user wishes to drag the marker. By default, markers are not draggable. A marker can be set as draggable by setting the `Marker.Draggable` property to `true` or by invoking the `MarkerOptions.Draggable` method with `true` as a parameter.

To drag the marker, the user must first long-click on the marker and then their finger must remain on the map. When the user's finger is dragged around on the screen, the marker will move. When the user's finger lifts off the screen, the marker will remain in place.

The following list describes the various events that will be raised for a draggable marker:

- `GoogleMap.MarkerDragStart(object sender, GoogleMap.MarkerDragStartEventArgs e)` – This event is raised when the user first drags the marker.
- `GoogleMap.MarkerDrag(object sender, GoogleMap.MarkerDragEventArgs e)` – This event is raised as the marker is being dragged.
- `GoogleMap.MarkerDragEnd(object sender, GoogleMap.MarkerDragEndEventArgs e)` – This event is raised when the user is finished dragging the marker.

Each of the `EventArgs` contains a single property called `P0` that is a reference to the `Marker` object being dragged.

Info Window Click events

Only one info window can be displayed at a time. When the user clicks on an info window in a map, the map object will raise an `InfoWindowClick` event. The following code snippet shows how to wire up a handler to the event:

```
public void OnMapReady(GoogleMap map)
{
    map.InfoWindowClick += MapOnInfoWindowClick;
}

private void MapOnInfoWindowClick (object sender, GoogleMap.InfoWindowClickEventArgs e)
{
    Marker myMarker = e.Marker;
    // Do something with marker.
}
```

Recall that an info window is a static `View` which is rendered as an image on the map. Any widgets such as buttons, check boxes, or text views that are placed inside the info window will be inert and cannot respond to any of their integral user events.

Related Links

- SimpleMapDemo
- Google Play Services
- Google Maps Android API v2
- Google Play Services APK
- Obtaining a Google Maps API key
- uses-library
- uses-feature

Obtaining a Google Maps API Key

7/8/2021 • 5 minutes to read • Edit Online

To use the Google Maps functionality in Android, you need to register for a Maps API key with Google. Until you do this, you will just see a blank grid instead of a map in your applications. You must obtain a Google Maps Android API v2 key - keys from the older Google Maps Android API key v1 will not work.

Obtaining a Maps API v2 key involves the following steps:

1. Retrieve the SHA-1 fingerprint of the keystore that is used to sign the application.
2. Create a project in the Google APIs console.
3. Obtaining the API key.

Obtaining your Signing Key Fingerprint

To request a Maps API key from Google, you need to know the SHA-1 fingerprint of the keystore that is used to sign the application. Typically, this means you will have to determine the SHA-1 fingerprint for the debug keystore, and then the SHA-1 fingerprint for the keystore that is used to sign your application for release.

- Visual Studio
- Visual Studio for Mac

By default the keystore that is used to sign debug versions of a Xamarin.Android application can be found at the following location:

C:\Users\[USERNAME]\AppData\Local\Xamarin\Mono for Android\debug.keystore

Information about a keystore is obtained by running the `keytool` command from the JDK. This tool is typically found in the Java bin directory:

C:\Program Files\Android\jdk\microsoft_dist_openjdk_[VERSION]\bin\keytool.exe

Run keytool using the following command (using the file paths shown above):

```
keytool -list -v -keystore [STORE FILENAME] -alias [KEY NAME] -storepass [STORE PASSWORD] -keypass [KEY
PASSWORD]
```

Debug.keystore Example

For the default debug key (which is automatically created for you for debugging), use this command:

- Visual Studio
- Visual Studio for Mac

```
keytool.exe -list -v -keystore "C:\Users\[USERNAME]\AppData\Local\Xamarin\Mono for Android\debug.keystore" -
alias androiddebugkey -storepass android -keypass android
```

Production Keys

When deploying an app to Google Play, it must be signed with a private key. The `keytool` will need to be run with the private key details, and the resulting SHA-1 fingerprint used to create a production Google Maps API key. Remember to update the **AndroidManifest.xml** file with the correct Google Maps API key before deployment.

Keytool Output

You should see something like the following output in your console window:

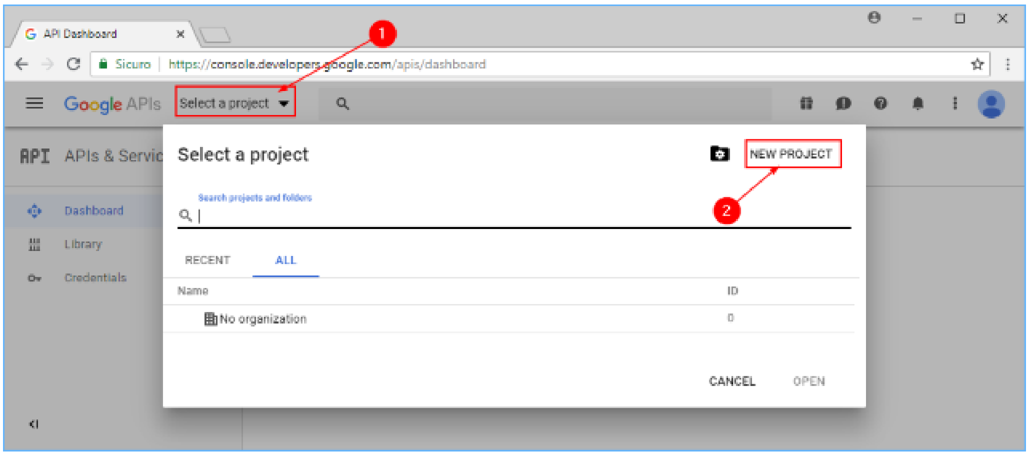
```
Alias name: androiddebugkey
Creation date: Jan 01, 2016
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=Android Debug, O=Android, C=US
Issuer: CN=Android Debug, O=Android, C=US
Serial number: 4aa9b300
Valid from: Mon Jan 01 08:04:04 UTC 2013 until: Mon Jan 01 18:04:04 PST 2033
Certificate fingerprints:
    MD5:  AE:9F:95:D0:A6:86:89:BC:A8:70:BA:34:FF:6A:AC:F9
    SHA1: BB:0D:AC:74:D3:21:E1:43:07:71:9B:62:90:AF:A1:66:6E:44:5D:75
    Signature algorithm name: SHA1withRSA
    Version: 3
```

You will use the SHA-1 fingerprint (listed after **SHA1**) later in this guide.

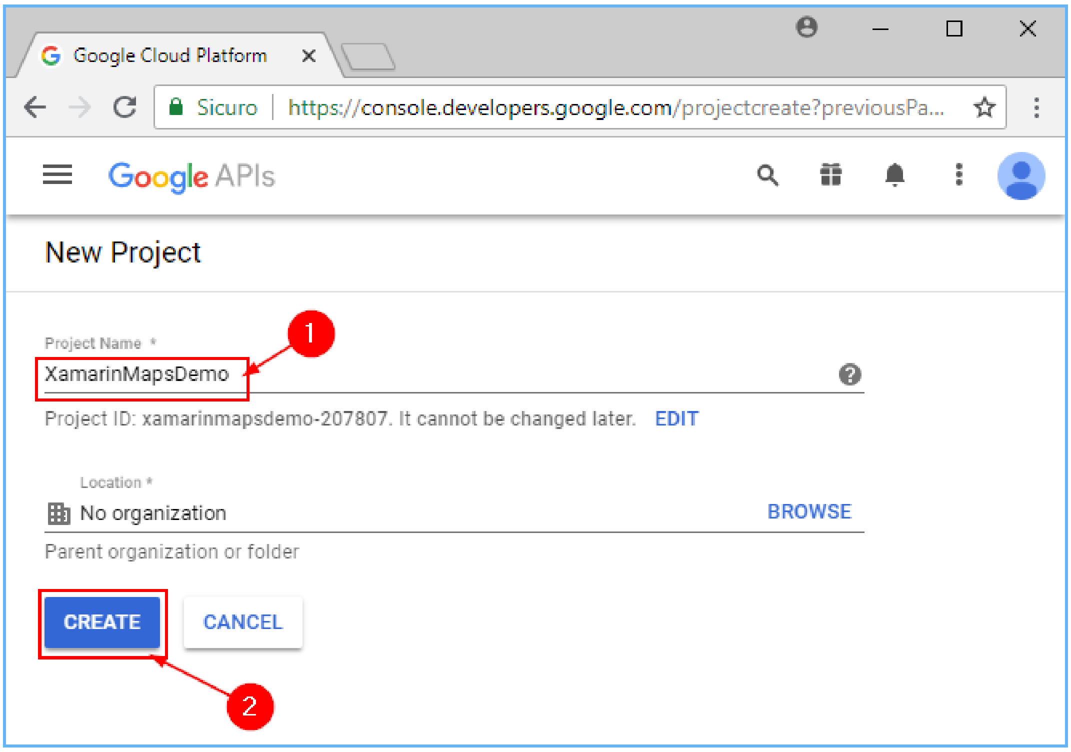
Creating an API project

After you have retrieved the SHA-1 fingerprint of the signing keystore, it is necessary to create a new project in the Google APIs console (or add the Google Maps Android API v2 service to an existing project).

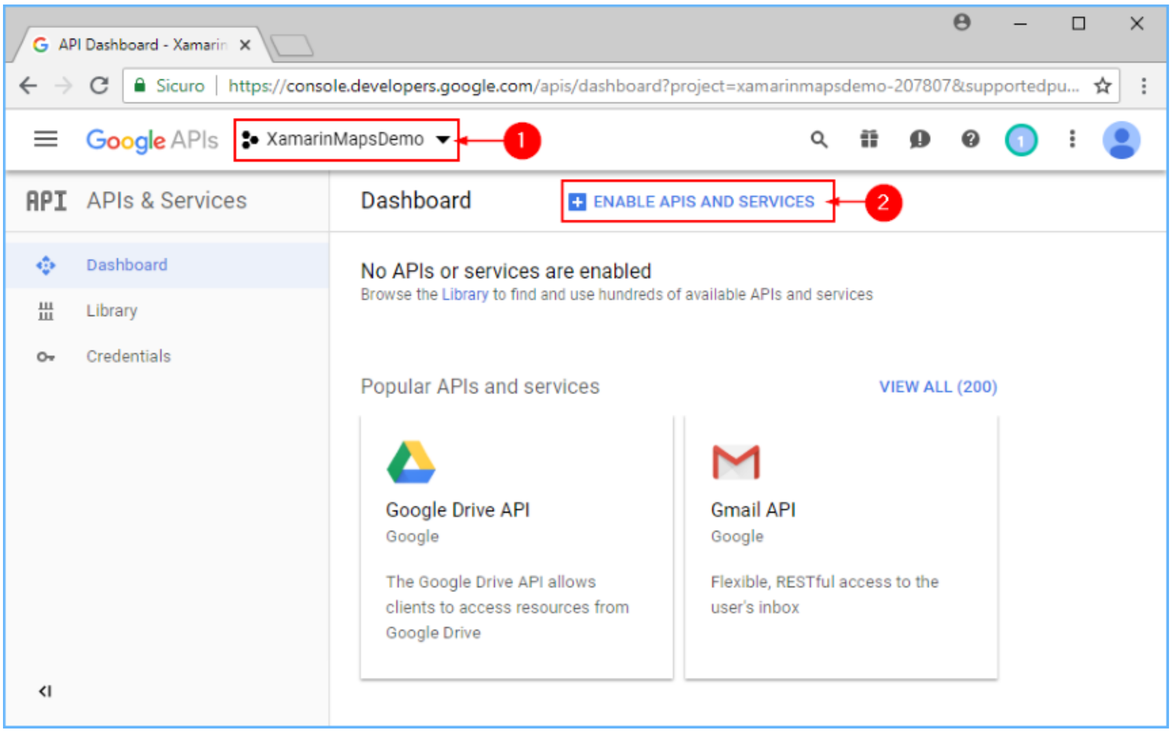
1. In a browser, navigate to the Google Developers Console API & Services Dashboard and click **Select a project**. Click on a project name or create a new one by clicking **NEW PROJECT**:



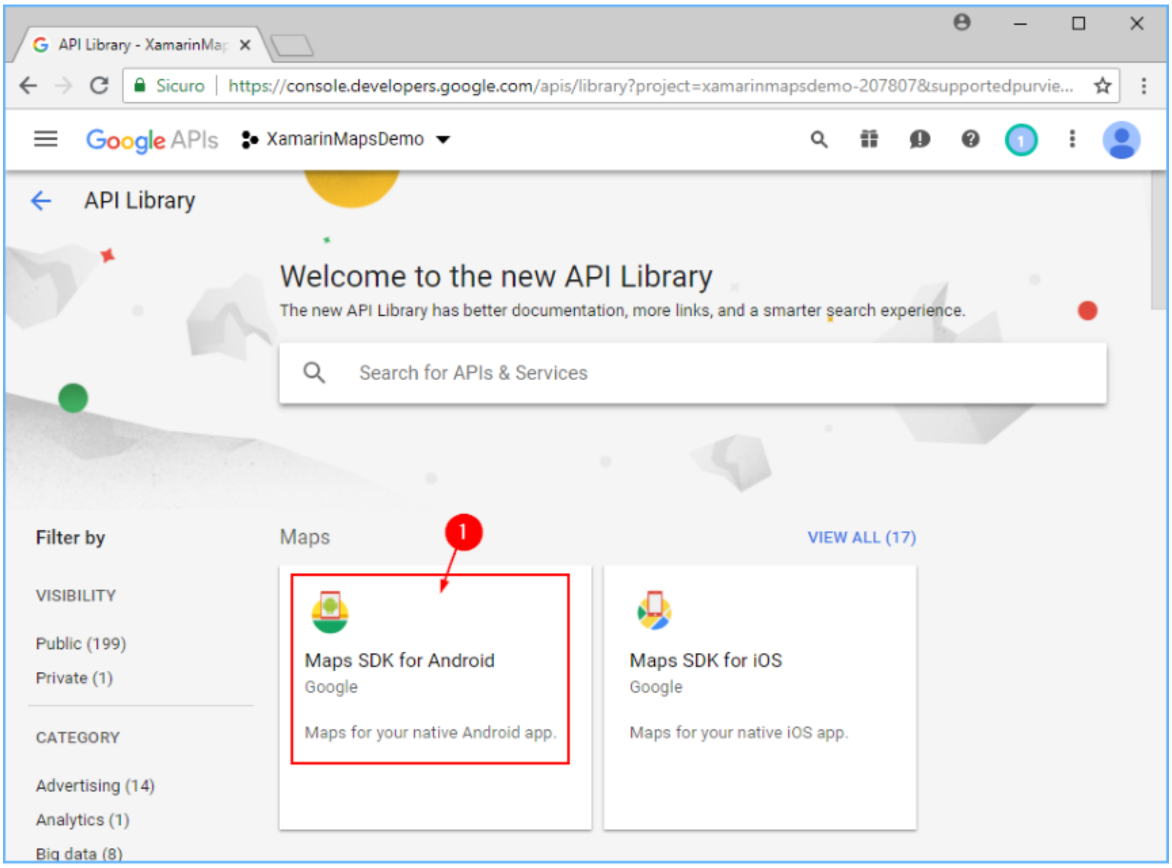
2. If you created a new project, enter the project name in the **New Project** dialog that is displayed. This dialog will manufacture a unique project ID that is based on your project name. Next, click the **Create** button as shown in this example:

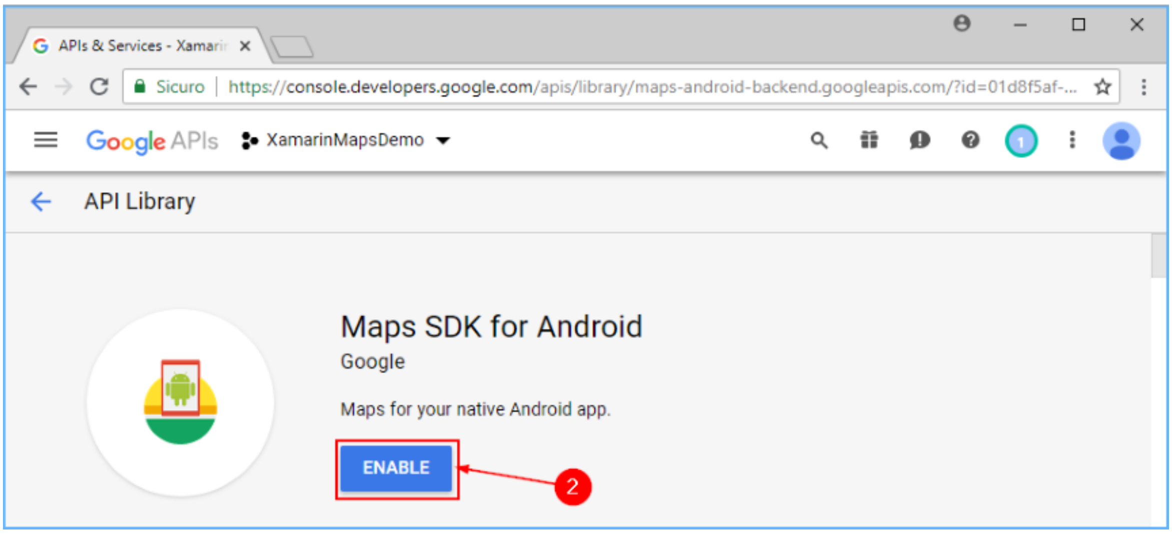


3. After a minute or so, the project is created and you are taken to the **Dashboard** page of the project. From there, click **ENABLE APIS AND SERVICES**:



4. From the **API Library** page, click **Maps SDK for Android**. On the next page, click **ENABLE** to turn on the service for this project:



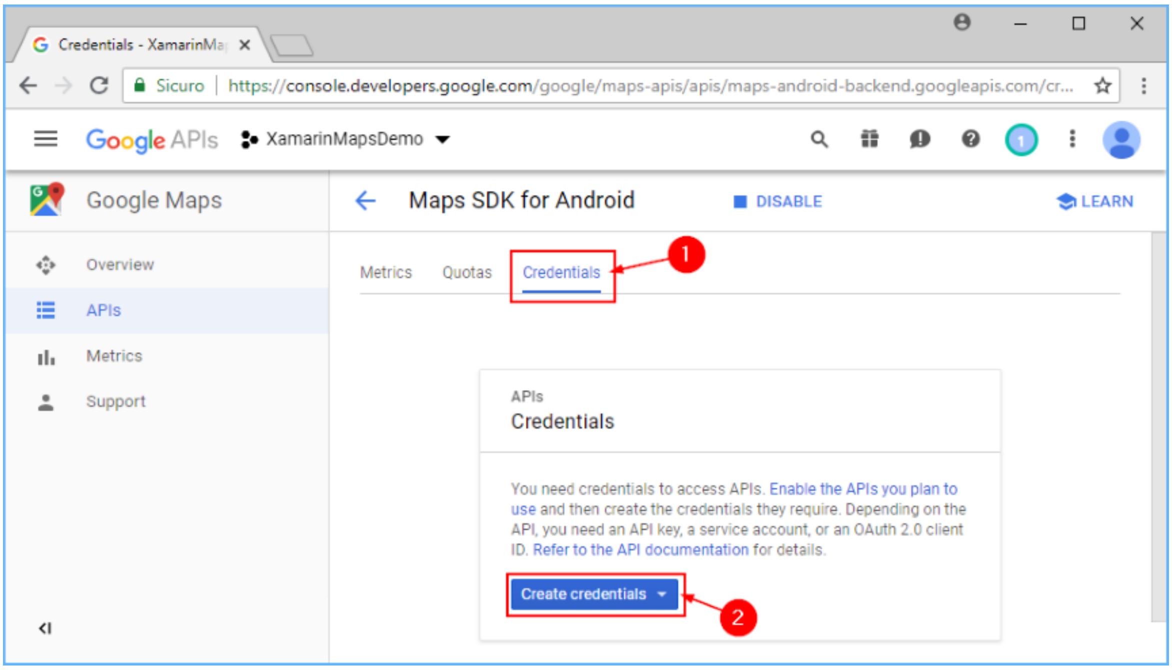


At this point the API project has been created and Google Maps Android API v2 has been added to it. However, you cannot use this API in your project until you create credentials for it. The next section explains how to create an API key and authorize a Xamarin.Android application to use this key.

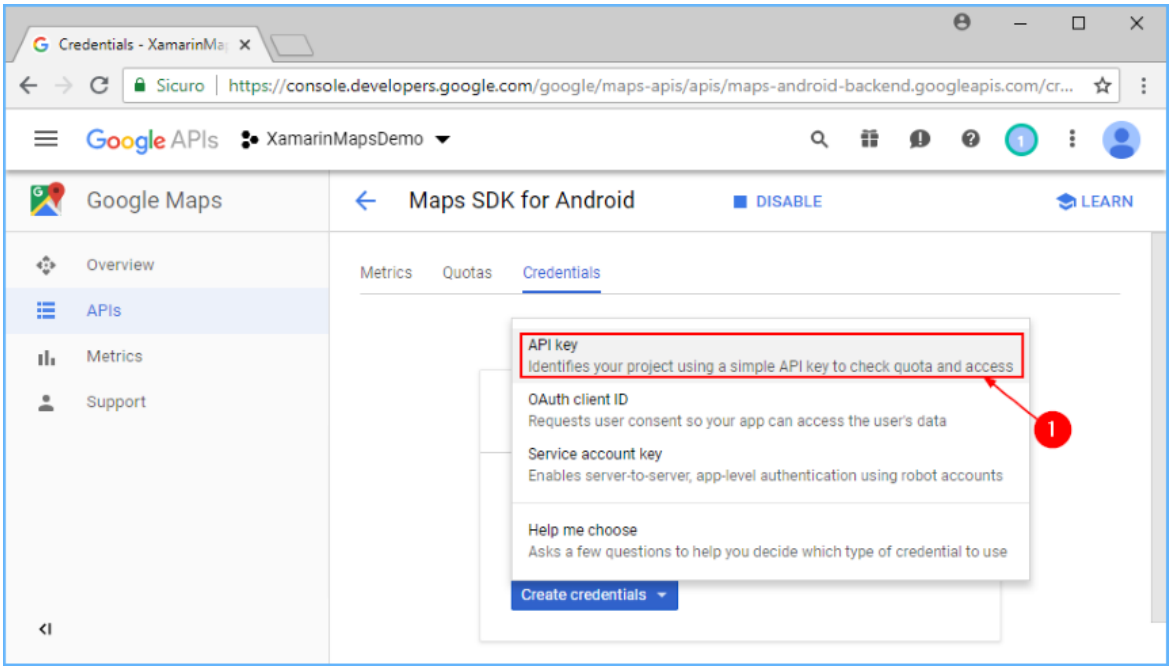
Obtaining the API Key

After the **Google Developer Console** API project has been created, it is necessary to create an Android API key. Xamarin.Android applications must have an API key before they are granted access to Android Map API v2.

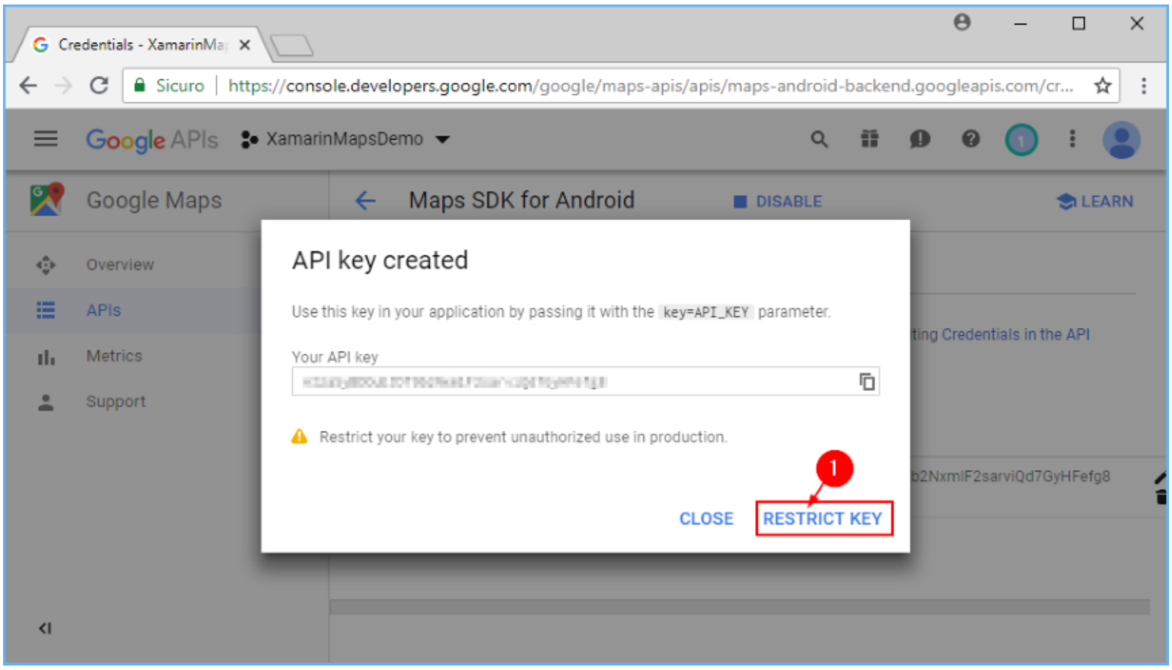
1. In the **Maps SDK for Android** page that is displayed (after clicking **ENABLE** in the previous step), go to the **Credentials** tab and click the **Create credentials** button:



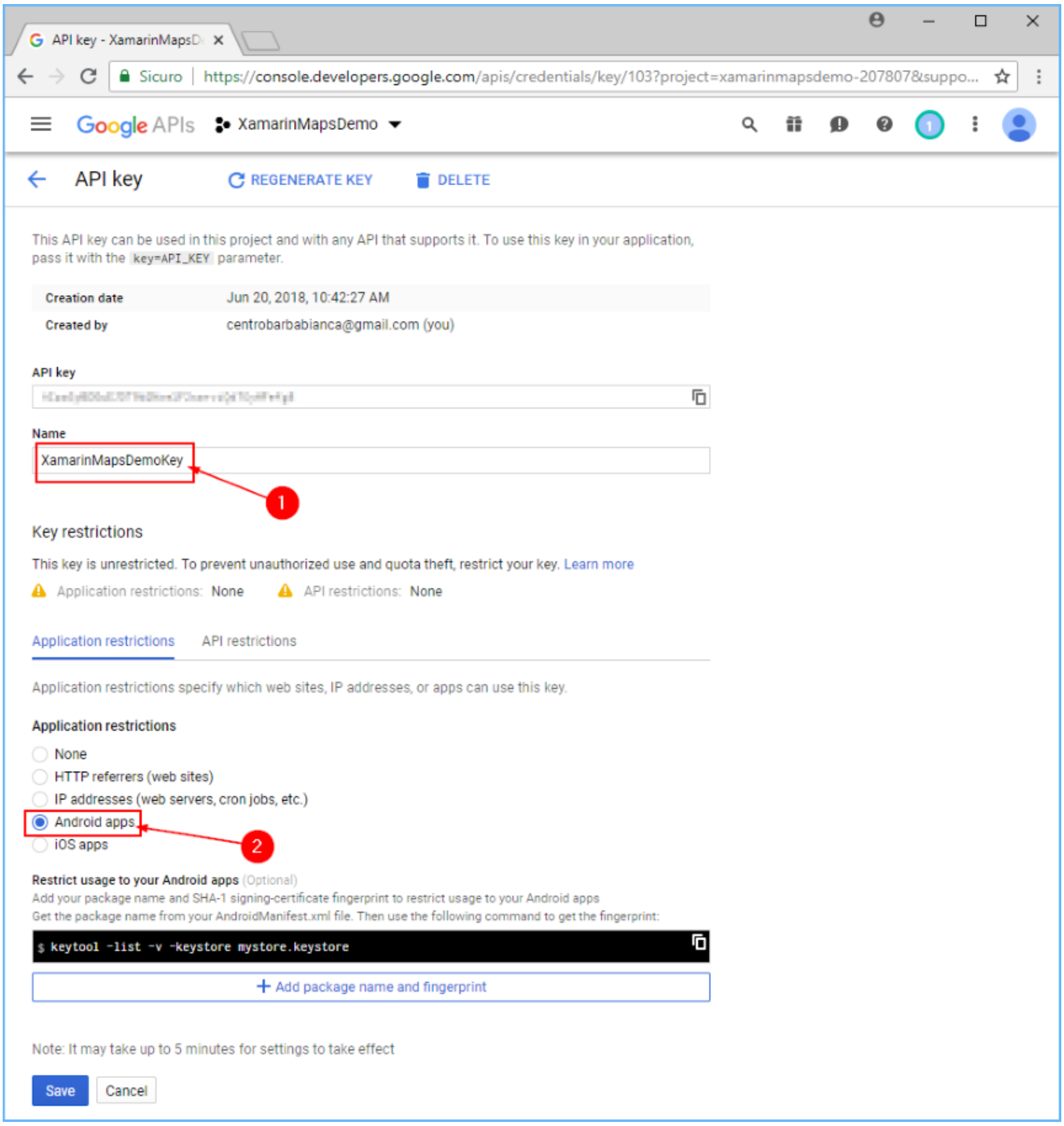
2. Click **API key**:



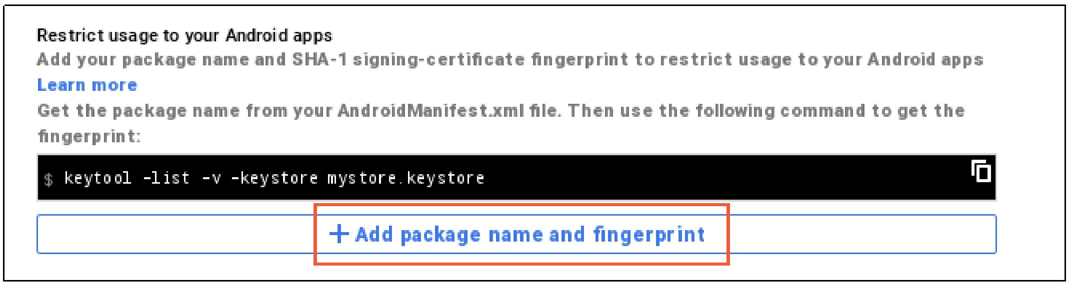
3. After this button is clicked, the API key is generated. Next it is necessary to restrict this key so that only your app can call APIs with this key. Click **RESTRICT KEY**:



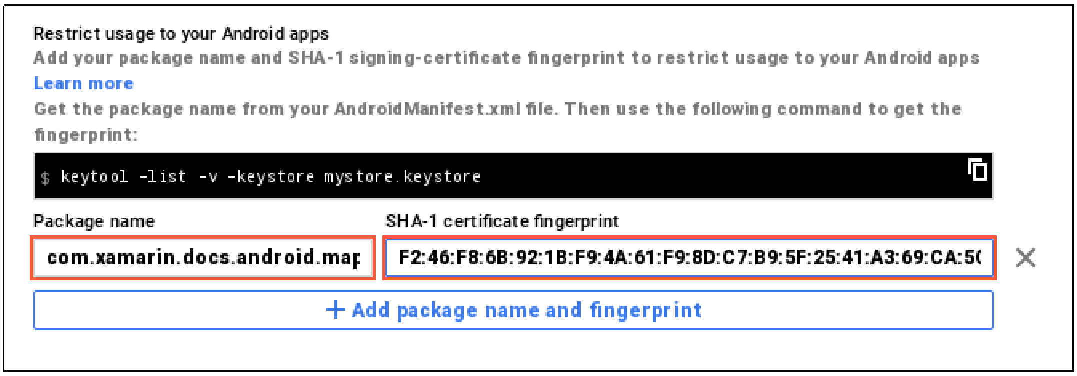
4. Change the **Name** field from **API Key 1** to a name that will help you remember what the key is used for (**XamarinMapsDemoKey** is used in this example). Next, click the **Android apps** radio button:



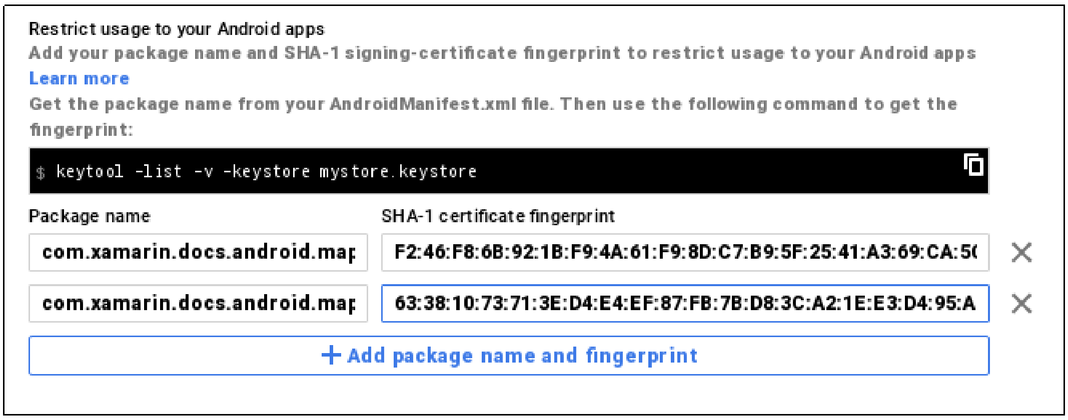
5. To add the SHA-1 fingerprint, click **+ Add package name and fingerprint**:



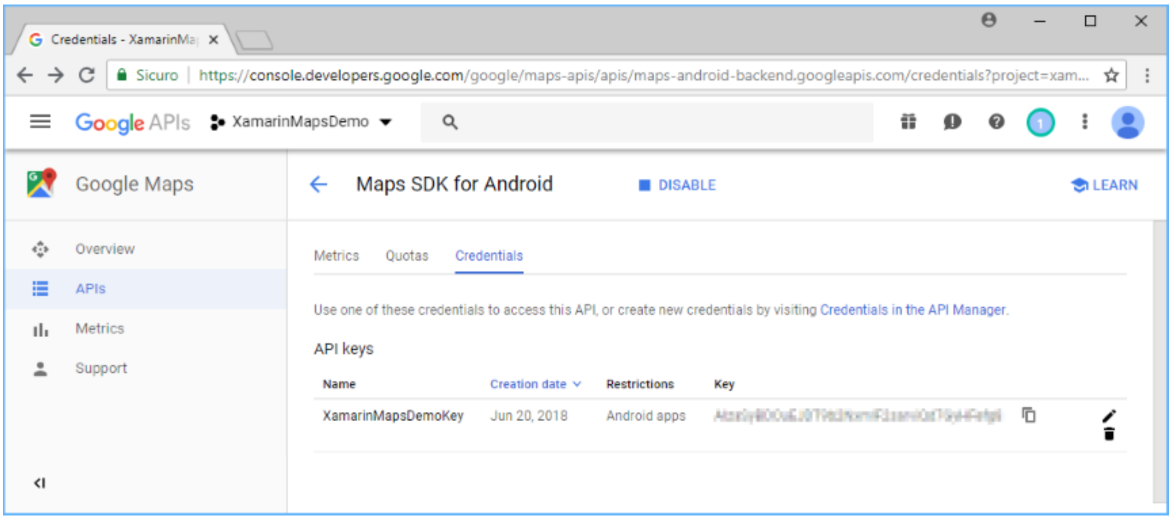
6. Enter your app's package name and enter the SHA-1 certificate fingerprint (obtained via `keytool` as explained earlier in this guide). In the following example, the package name for `XamarinMapsDemo` is entered, followed by the SHA-1 certificate fingerprint obtained from **debug.keystore**:



7. Note that, in order for your APK to access Google Maps, you must include SHA-1 fingerprints and package names for every keystore (debug and release) that you use to sign your APK. For example, if you use one computer for debug and another computer for generating the release APK, you should include the SHA-1 certificate fingerprint from the debug keystore of the first computer and the SHA-1 certificate fingerprint from the release keystore of the second computer. Click **+ Add package name and fingerprint** to add another fingerprint and package name as shown in this example:



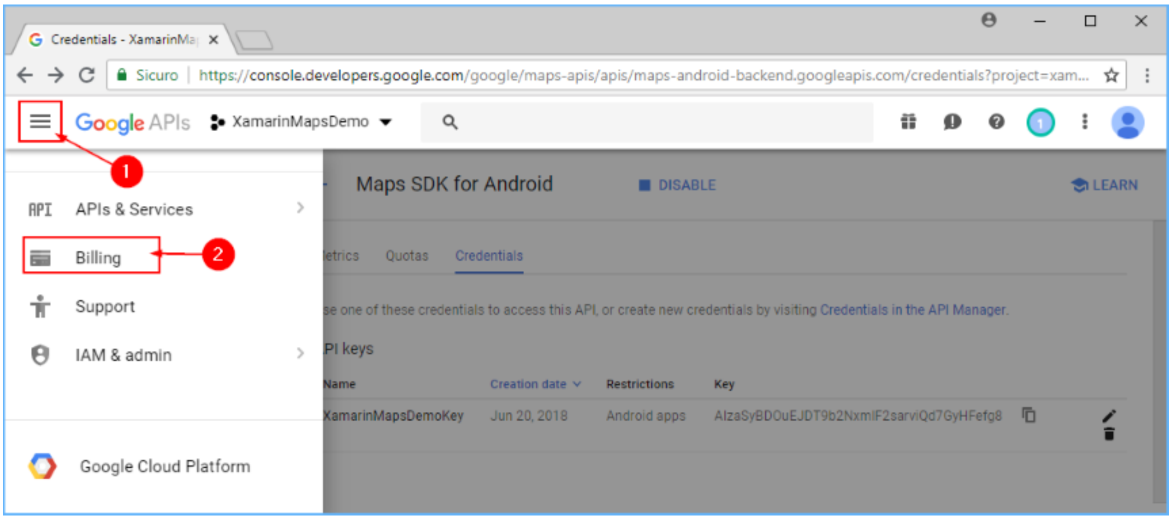
8. Click the **Save** button to save your changes. Next, you are returned to the list of your API keys. If you have other API keys that you have created earlier, they will also be listed here. In this example, only one API key (created in the previous steps) is listed:



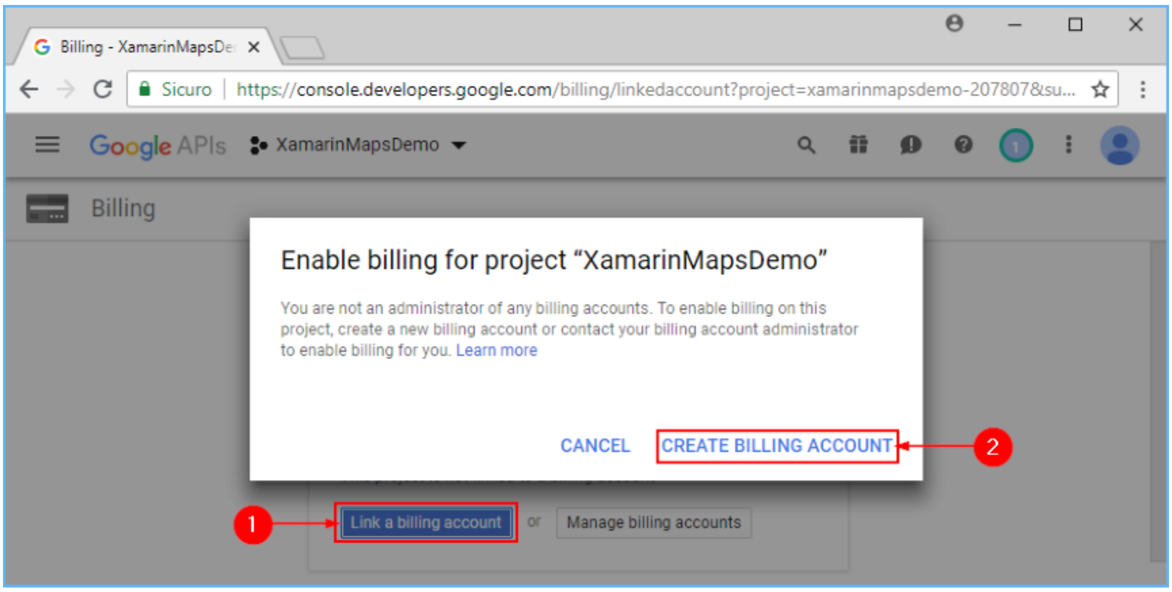
Connect the project to a billable account

Beginning June,11 2018, the API key will not work if the project is not connected to a billable account (even if the service is still free for mobile apps).

1. Click the hamburger menu button and select the **Billing** page:



2. Link the project to a billing account by clicking **Link a billing account** followed by **CREATE BILLING ACCOUNT** on the displayed popup (if you don't have an account, you will be guided to create a new one):



Adding the Key to Your Project

Finally, add this API key to the **AndroidManifest.XML** file of your Xamarin.Android app. In the following example, `YOUR_API_KEY` is to be replaced with the API key generated in the previous steps:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionName="4.10" package="com.xamarin.docs.android.mapsandlocationdemo"
    android:versionCode="10">
...
  <application android:label="@string/app_name">
    <!-- Put your Google Maps V2 API Key here. -->
    <meta-data android:name="com.google.android.maps.v2.API_KEY" android:value="YOUR_API_KEY" />
    <meta-data android:name="com.google.android.gms.version"
android:value="@integer/google_play_services_version" />
  </application>
</manifest>
```

Related Links

- Google APIs Console
- The Google Maps API Key
- keytool

Android Speech

7/8/2021 • 8 minutes to read • Edit Online

This article covers the basics of using the very powerful Android.Speech namespace. Since its inception, Android has been able to recognize speech and output it as text. It is a relatively simple process. For text to speech, however, the process is more involved, as not only does the speech engine have to be taken into account, but also the languages available and installed from the Text To Speech (TTS) system.

Speech Overview

Having a system, which "understands" human speech and enunciates what is being typed—Speech to Text, and Text to Speech—is an ever growing area within mobile development as the demand for natural communication with our devices rises. There are many instances where having a feature that converts text into speech, or vice versa, is a very useful tool to incorporate into your android application.

For example, with the clamp down on mobile phone use while driving, users want a hands free way of operating their devices. The plethora of different Android form factors—such as Android Wear—and the ever-widening inclusion of those able to use Android devices (such as tablets and note pads), has created a larger focus on great TTS applications.

Google supplies the developer with a rich set of APIs in the Android.Speech namespace to cover most instances of making a device "speech aware" (such as software designed for the blind). The namespace includes the facility to allow text to be translated into speech through `Android.Speech.Tts`, control over the engine used to perform the translation, as well as a number of `RecognizerIntent`s which allow speech to be converted to text.

While the facilities are there for speech to be understood, there are limitations based on the hardware used. It is unlikely that the device will successfully interpret everything spoken to it in every language available.

Requirements

There are no special requirements for this guide, other than your device having a microphone and speaker.

The core of an Android device interpreting speech is the use of an `Intent` with a corresponding `OnActivityResult`. It is important, though, to recognize that the speech is not understood—but interpreted to text. The difference is important.

The difference between understanding and interpreting

A simple definition of understanding is that you are able to determine by tone and context the real meaning of what is being said. To interpret just means to take the words and output them in another form.

Consider the following simple example that is used in everyday conversation:

```
Hello, how are you?
```

Without inflection (emphasis placed on specific words or parts of words), it is a simple question. However, if a slow pace is applied to the line, the person listening will detect that the asker is not too happy and perhaps needs cheering up or that the asker is unwell. If the emphasis is placed on "are", the person asking is usually more interested in the response.

Without fairly powerful audio processing to make use of the inflection, and a degree of artificial intelligence (AI) to understand the context, the software cannot even begin to understand what was said—the best a simple phone can do is convert the speech to text.

Setting up

Before using the speech system, it is always wise to check to ensure the device has a microphone. There would be little point trying to run your app on a Kindle or Google note pad without a microphone installed.

The code sample below demonstrates querying if a microphone is available and if not, to create an alert. If no microphone is available at this point you would either quit the activity or disable the ability to record the speech.

```
string rec = Android.Content.PM.PackageManager.FeatureMicrophone;
if (rec != "android.hardware.microphone")
{
    var alert = new AlertDialog.Builder(recButton.Context);
    alert.SetTitle("You don't seem to have a microphone to record with");
    alert.SetPositiveButton("OK", (sender, e) =>
    {
        return;
    });
    alert.Show();
}
```

Creating the intent

The intent for the speech system uses a particular type of intent called the `RecognizerIntent`. This intent controls a large number of parameters, including how long to wait with silence until the recording is considered over, any additional languages to recognize and output, and any text to include on the `Intent`'s modal dialog as means of instruction. In this snippet, `VOICE` is a `readonly int` used for recognition in `OnActivityResult`.

```
var voiceIntent = new Intent(RecognizerIntent.ActionRecognizeSpeech);
voiceIntent.PutExtra(RecognizerIntent.ExtraLanguageModel, RecognizerIntent.LanguageModelFreeForm);
voiceIntent.PutExtra(RecognizerIntent.ExtraPrompt,
Application.Context.GetString(Resource.String.messageSpeakNow));
voiceIntent.PutExtra(RecognizerIntent.ExtraSpeechInputCompleteSilenceLengthMillis, 1500);
voiceIntent.PutExtra(RecognizerIntent.ExtraSpeechInputPossiblyCompleteSilenceLengthMillis, 1500);
voiceIntent.PutExtra(RecognizerIntent.ExtraSpeechInputMinimumLengthMillis, 15000);
voiceIntent.PutExtra(RecognizerIntent.ExtraMaxResults, 1);
voiceIntent.PutExtra(RecognizerIntent.ExtraLanguage, Java.Util.Locale.Default);
StartActivityForResult(voiceIntent, VOICE);
```

Conversion of the speech

The text interpreted from the speech will be delivered within the `Intent`, which is returned when the activity has been completed and is accessed via `GetStringArrayListExtra(RecognizerIntent.ExtraResults)`. This will return an `IList<string>`, of which the index can be used and displayed, depending on the number of languages requested in the caller intent (and specified in the `RecognizerIntent.ExtraMaxResults`). As with any list though, it is worth checking to ensure that there is data to be displayed.

When listening for the return value of a `StartActivityForResult`, the `OnActivityResult` method has to be supplied.

In the example below, `textBox` is a `TextBox` used for outputting what has been dictated. It could equally be used to pass the text to some form of interpreter and from there, the application can compare the text and branch to another part of the application.

```
protected override void OnActivityResult(int requestCode, Result resultVal, Intent data)
{
    if (requestCode == VOICE)
    {
        if (resultVal == Result.Ok)
        {
            var matches = data.GetStringArrayListExtra(RecognizerIntent.ExtraResults);
            if (matches.Count != 0)
            {
                string textInput = textBox.Text + matches[0];
                textBox.Text = textInput;
                switch (matches[0].Substring(0, 5).ToLower())
                {
                    case "north":
                        MovePlayer(0);
                        break;
                    case "south":
                        MovePlayer(1);
                        break;
                }
            }
            else
            {
                textBox.Text = "No speech was recognised";
            }
        }
        base.OnActivityResult(requestCode, resultVal, data);
    }
}
```

Text to Speech

Text to speech is not quite the reverse of speech to text and relies on two key components; a text-to-speech engine being installed on the device and a language being installed.

Largely, Android devices come with the default Google TTS service installed and at least one language. This is established when the device is first set up and will be based on where the device is at the time (for example, a phone set up in Germany will install the German language, whereas one in America will have American English).

Step 1 - Instantiating TextToSpeech

`TextToSpeech` can take up to 3 parameters, the first two are required with the third being optional (`AppContext`, `IOnInitListener`, `engine`). The listener is used to bind to the service and test for failure with the engine being any number of available Android text to speech engines. At a minimum, the device will have Google's own engine.

Step 2 - Finding the languages available

The `Java.Util.Locale` class contains a helpful method called `GetAvailableLocales()`. This list of languages supported by the speech engine can then be tested against the installed languages.

It is a trivial matter to generate the list of "understood" languages. There will always be a default language (the language the user set when they first set their device up), so in this example the `List<string>` has "Default" as the first parameter, the remainder of the list will be filled depending on the result of the `textToSpeech.IsLanguageAvailable(locale)`.

```
var langAvailable = new List<string>{ "Default" };
var localesAvailable = Java.Util.Locale.GetAvailableLocales().ToList();
foreach (var locale in localesAvailable)
{
    var res = textToSpeech.IsLanguageAvailable(locale);
    switch (res)
    {
        case LanguageAvailableResult.Available:
          langAvailable.Add(locale.DisplayLanguage);
          break;
        case LanguageAvailableResult.CountryAvailable:
          langAvailable.Add(locale.DisplayLanguage);
          break;
        case LanguageAvailableResult.CountryVarAvailable:
          langAvailable.Add(locale.DisplayLanguage);
          break;
    }
}
langAvailable = langAvailable.OrderBy(t => t).Distinct().ToList();
```

This code calls TextToSpeech.IsLanguageAvailable to test if the language package for a given locale is already present on the device. This method returns a LanguageAvailableResult, which indicates whether the language for the passed locale is available. If `LanguageAvailableResult` indicates that the language is `NotSupported`, then there is no voice package available (even for download) for that language. If `LanguageAvailableResult` is set to `MissingData`, then it is possible to download a new language package as explained below in Step 4.

Step 3 - Setting the speed and pitch

Android allows the user to alter the sound of the speech by altering the `SpeechRate` and `Pitch` (the rate of speed and the tone of the speech). This goes from 0 to 1, with "normal" speech being 1 for both.

Step 4 - Testing and loading new languages

Downloading a new language is performed by using an `Intent`. The result of this intent causes the OnActivityResult method to be invoked. Unlike the speech-to-text example (which used the RecognizerIntent as a `PutExtra` parameter to the `Intent`), the testing and loading `Intent`s are `Action`-based:

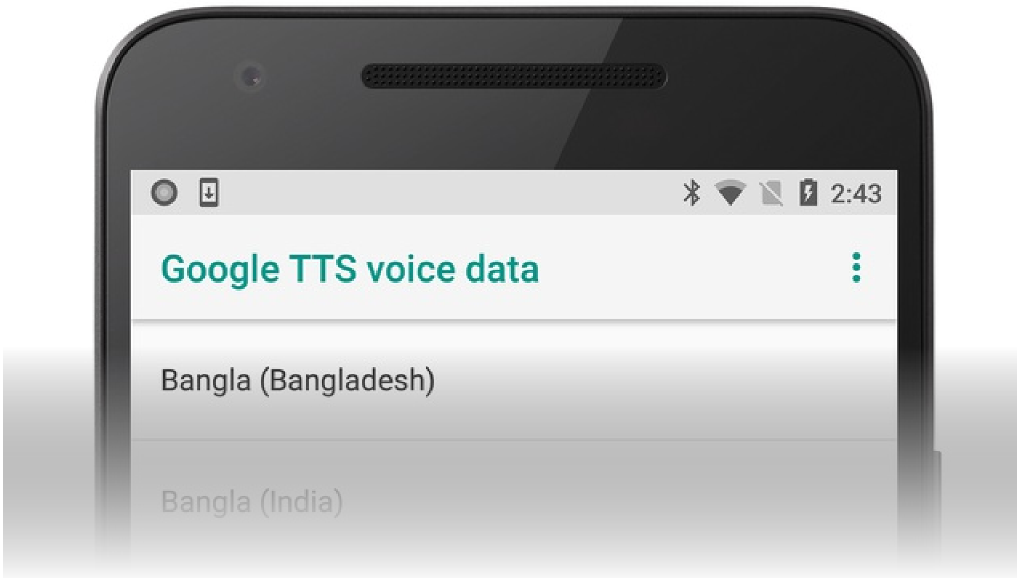
- TextToSpeech.Engine.ActionCheckTtsData – Starts an activity from the platform `TextToSpeech` engine to verify proper installation and availability of language resources on the device.
- TextToSpeech.Engine.ActionInstallTtsData – Starts an activity that prompts the user to download the necessary languages.

The following code example illustrates how to use these actions to test for language resources and download a new language:

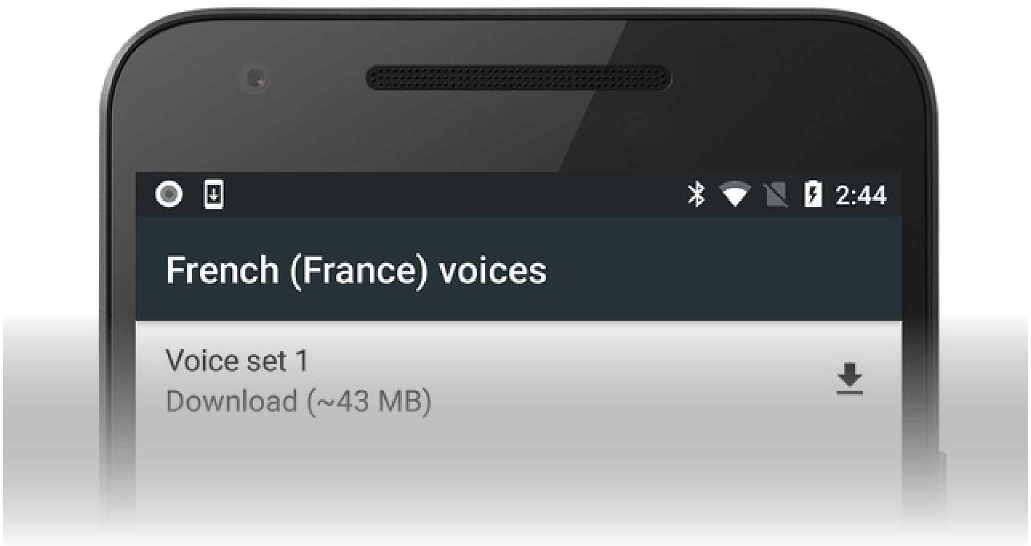
```
var checkTTSIntent = new Intent();
checkTTSIntent.SetAction(TextToSpeech.Engine.ActionCheckTtsData);
StartActivityForResult(checkTTSIntent, NeedLang);
//
protected override void OnActivityResult(int req, Result res, Intent data)
{
    if (req == NeedLang)
    {
        var installTTS = new Intent();
        installTTS.SetAction(TextToSpeech.Engine.ActionInstallTtsData);
        StartActivity(installTTS);
    }
}
```

`TextToSpeech.Engine.ActionCheckTtsData` tests for the availability of language resources. `OnActivityResult` is invoked when this test completes. If language resources need to be downloaded, `OnActivityResult` fires off the `TextToSpeech.Engine.ActionInstallTtsData` action to start an activity that allows the user to download the necessary languages. Note that this `OnActivityResult` implementation does not check the `Result` code because, in this simplified example, the determination has already been made that the language package needs to be downloaded.

The `TextToSpeech.Engine.ActionInstallTtsData` action causes the **Google TTS voice data** activity to be presented to the user for choosing languages to download:



As an example, the user might pick French and click the download icon to download French voice data:



Installation of this data happens automatically after the download completes.

Step 5 - The IOnInitListener

For an activity to be able to convert the text to speech, the interface method `OnInit` has to be implemented (this is the second parameter specified for the instantiation of the `TextToSpeech` class). This initializes the listener and tests the result.

The listener should test for both `OperationResult.Success` and `OperationResult.Failure` at a minimum. The following example shows just that:

```
void TextToSpeech.IOnInitListener.OnInit(OperationResult status)
{
    // if we get an error, default to the default language
    if (status == OperationResult.Error)
        textToSpeech.SetLanguage(Java.Util.Locale.Default);
    // if the listener is ok, set the lang
    if (status == OperationResult.Success)
        textToSpeech.SetLanguage(lang);
}
```

Summary

In this guide we have looked at the basics of converting text to speech and speech to text and possible methods of how to include them within your own apps. While they do not cover every particular case, you should now have a basic understanding of how speech is interpreted, how to install new languages, and how to increase the inclusivity of your apps.

Related Links

- Xamarin.Forms DependencyService
- Text to Speech (sample)
- Speech to Text (sample)
- Android.Speech namespace
- Android.Speech.Tts namespace

Java integration with Xamarin.Android

10/29/2019 • 2 minutes to read • Edit Online

The Java ecosystem includes a diverse and immense collection of components. Many of these components can be used to reduce the time it takes to develop an Android application. This document will introduce and provide a high-level overview of some of the ways that developers can use these existing Java components to improve their Xamarin.Android application development experience.

Overview

Given the extent of the Java ecosystem, it is very likely that any given functionality required for a Xamarin.Android application has already been coded in Java. Because of this, it is appealing to try and reuse these existing libraries when creating a Xamarin.Android application.

There are three possible ways to reuse Java libraries in a Xamarin.Android application:

- **Create a Java Bindings Library** – With this technique, a Xamarin.Android project is used to create C# wrappers around the Java types. A Xamarin.Android application can then reference the C# wrappers created by this project, and then use the `.jar` file.
- **Java Native Interface** – The *Java Native Interface* (JNI) is a framework that allows non-Java code (such as C++ or C#) to call or be called by Java code running inside a JVM.
- **Port the Code** – This method involves taking the Java source code, and then converting it to C#. This can be done manually, or by using an automated tool such as Sharpen.

At the core of the first two techniques is the *Java Native Interface* (JNI). JNI is a framework that allows applications not written in Java to interact with Java code running in a Java Virtual Machine. Xamarin.Android uses JNI to create *bindings* for C# code.

The first technique is a more automated, declarative approach to binding Java libraries. It involves using either Visual Studio for Mac or a Visual Studio project type that is provided by Xamarin.Android – the Java Bindings Library. To successfully create these bindings, a Java Bindings Library may still require some manual modifications, but not as many as would a pure JNI approach. See Binding a Java Library for more information about Java Binding libraries.

The second technique, using JNI, works at a much lower level, but can provide for finer control and access to Java methods that would not normally be accessible through a Java Binding Library.

The third technique is radically different from the previous two: porting the code from Java to C#. Porting code from one language to another can be a very laborious process, but it is possible to reduce that effort with the help of a tool called *Sharpen*. Sharpen is an open source tool that is a Java-to-C# converter.

Summary

This document provided a high-level overview of some of the different ways that libraries from Java can be reused in a Xamarin.Android application. It introduced the concepts of bindings and managed callable wrappers, and discussed options for porting Java code to C#.

Related Links

- Architecture

- Binding a Java Library
- Working with JNI
- Sharpen
- Java Native Interface

Android Callable Wrappers for Xamarin.Android

4/20/2020 • 3 minutes to read • Edit Online

Android Callable Wrappers (ACWs) are required whenever the Android runtime invokes managed code. These wrappers are required because there is no way to register classes with ART (the Android runtime) at runtime. (Specifically, the JNI DefineClass() function is not supported by the Android runtime.} Android Callable Wrappers thus make up for the lack of runtime type registration support.

Every time Android code needs to execute a `virtual` or interface method that is `overridden` or implemented in managed code, Xamarin.Android must provide a Java proxy so that this method is dispatched to the appropriate managed type. These Java proxy types are Java code that has the "same" base class and Java interface list as the managed type, implementing the same constructors and declaring any overridden base class and interface methods.

Android callable wrappers are generated by the **monodroid.exe** program during the build process: they are generated for all types that (directly or indirectly) inherit Java.Lang.Object.

Android Callable Wrapper Naming

Package names for Android Callable Wrappers are based on the MD5SUM of the assembly-qualified name of the type being exported. This naming technique makes it possible for the same fully-qualified type name to be made available by different assemblies without introducing a packaging error.

Because of this MD5SUM naming scheme, you cannot directly access your types by name. For example, the following `adb` command will not work because the type name `my.ActivityType` is not generated by default:

```
adb shell am start -n My.Package.Name/my.ActivityType
```

Also, you may see errors like the following if you attempt to reference a type by name:

```
java.lang.ClassNotFoundException: Didn't find class "com.company.app.MainActivity"
on path: DexPathList[[zip file "/data/app/com.company.App-1.apk"] ...
```

If you *do* require access to types by name, you can declare a name for that type in an attribute declaration. For example, here is code that declares an activity with the fully-qualified name `My.ActivityType`:

```
namespace My {
    [Activity]
    public partial class ActivityType : Activity {
        /* ... */
    }
}
```

The `ActivityAttribute.Name` property can be set to explicitly declare the name of this activity:

```
namespace My {
    [Activity(Name="my.ActivityType")]
    public partial class ActivityType : Activity {
        /* ... */
    }
}
```

After this property setting is added, `my.ActivityType` can be accessed by name from external code and from `adb` scripts. The `Name` attribute can be set for many different types including `Activity`, `Application`, `Service`, `BroadcastReceiver`, and `ContentProvider`:

- ActivityAttribute.Name
- ApplicationAttribute.Name
- ServiceAttribute.Name
- BroadcastReceiverAttribute.Name
- ContentProviderAttribute.Name

MD5SUM-based ACW naming was introduced in Xamarin.Android 5.0. For more information about attribute naming, see RegisterAttribute.

Implementing Interfaces

There are times when you may need to implement an Android interface, such as Android.Content.IComponentCallbacks. Since all Android classes and interface extend the Android.Runtime.IJavaObject interface, the question arises: how do we implement `IJavaObject`?

The question was answered above: the reason all Android types need to implement `IJavaObject` is so that Xamarin.Android has an Android callable wrapper to provide to Android, i.e. a Java proxy for the given type. Since **monodroid.exe** only looks for `Java.Lang.Object` subclasses, and `Java.Lang.Object` implements `IJavaObject`, the answer is obvious: subclass `Java.Lang.Object`:

```
class MyComponentCallbacks : Java.Lang.Object, Android.Content.IComponentCallbacks {

    public void OnConfigurationChanged (Android.Content.Res.Configuration newConfig)
    {
        // implementation goes here...
    }

    public void OnLowMemory ()
    {
        // implementation goes here...
    }
}
```

Implementation Details

The remainder of this page provides implementation details subject to change without notice (and is presented here only because developers will be curious about what's going on).

For example, given the following C# source:

```
using System;
using Android.App;
using Android.OS;

namespace Mono.Samples.HelloWorld
{
    public class HelloAndroid : Activity
    {
        protected override void OnCreate (Bundle savedInstanceState)
        {
            base.OnCreate (savedInstanceState);
            SetContentView (R.layout.main);
        }
    }
}
```

The **mandroid.exe** program will generate the following Android Callable Wrapper:

```
package mono.samples.helloWorld;

public class HelloAndroid
    extends android.app.Activity
{
    static final String __md_methods;
    static {
        __md_methods = "n_onCreate:(Landroid/os/Bundle;)V:GetOnCreate_Landroid_os_Bundle_Handler\n" + "";
        mono.android.Runtime.register (
            "Mono.Samples.HelloWorld.HelloAndroid, HelloWorld, Version=1.0.0.0,
            Culture=neutral, PublicKeyToken=null", HelloAndroid.class, __md_methods);
    }

    public HelloAndroid ()
    {
        super ();
        if (getClass () == HelloAndroid.class)
            mono.android.TypeManager.Activate (
                "Mono.Samples.HelloWorld.HelloAndroid, HelloWorld, Version=1.0.0.0,
                Culture=neutral, PublicKeyToken=null", "", this, new java.lang.Object[] {  });
    }

    @Override
    public void onCreate (android.os.Bundle p0)
    {
        n_onCreate (p0);
    }

    private native void n_onCreate (android.os.Bundle p0);
}
```

Notice that the base class is preserved, and `native` method declarations are provided for each method that is overridden within managed code.

Working with JNI and Xamarin.Android

11/2/2020 • 46 minutes to read • Edit Online

Xamarin.Android permits writing Android apps with C# instead of Java. Several assemblies are provided with Xamarin.Android which provide bindings for Java libraries, including Mono.Android.dll and Mono.Android.GoogleMaps.dll. However, bindings are not provided for every possible Java library, and the bindings that are provided may not bind every Java type and member. To use unbound Java types and members, the Java Native Interface (JNI) may be used. This article illustrates how to use JNI to interact with Java types and members from Xamarin.Android applications.

Overview

It is not always necessary or possible to create a Managed Callable Wrapper (MCW) to invoke Java code. In many cases, "inline" JNI is perfectly acceptable and useful for one-off use of unbound Java members. It is often simpler to use JNI to invoke a single method on a Java class than to generate an entire .jar binding.

Xamarin.Android provides the `Mono.Android.dll` assembly, which provides a binding for Android's `android.jar` library. Types and members not present within `Mono.Android.dll` and types not present within `android.jar` may be used by manually binding them. To bind Java types and members, you use the **Java Native Interface** (**JNI**) to lookup types, read and write fields, and invoke methods.

The JNI API in Xamarin.Android is conceptually very similar to the `System.Reflection` API in .NET: it makes it possible for you to look up types and members by name, read and write field values, invoke methods, and more. You can use JNI and the `Android.Runtime.RegisterAttribute` custom attribute to declare virtual methods that can be bound to support overriding. You can bind interfaces so that they can be implemented in C#.

This document explains:

- How JNI refers to types.
- How to lookup, read, and write fields.
- How to lookup and invoke methods.
- How to expose virtual methods to allow overriding from managed code.
- How to expose interfaces.

Requirements

JNI, as exposed through the Android.Runtime.JNIEnv namespace, is available in every version of Xamarin.Android. To bind Java types and interfaces, you must use Xamarin.Android 4.0 or later.

Managed Callable Wrappers

A **Managed Callable Wrapper** (**MCW**) is a *binding* for a Java class or interface which wraps up the all the JNI machinery so that client C# code doesn't need to worry about the underlying complexity of JNI. Most of `Mono.Android.dll` consists of managed callable wrappers.

Managed callable wrappers serve two purposes:

1. Encapsulate JNI use so that client code doesn't need to know about the underlying complexity.
2. Make it possible to sub-class Java types and implement Java interfaces.

The first purpose is purely for convenience and encapsulation of complexity so that consumers have a simple, managed set of classes to use. This requires use of the various JNIEnv members as described later in this article. Keep in mind that managed callable wrappers aren't strictly necessary – "inline" JNI use is perfectly acceptable and is useful for one-off use of unbound Java members. Sub-classing and interface implementation requires the use of managed callable wrappers.

Android Callable Wrappers

Android callable wrappers (ACW) are required whenever the Android runtime (ART) needs to invoke managed code; these wrappers are required because there is no way to register classes with ART at runtime. (Specifically, the DefineClass JNI function is not supported by the Android runtime. Android callable wrappers thus make up for the lack of runtime type registration support.)

Whenever Android code needs to execute a virtual or interface method that is overridden or implemented in managed code, Xamarin.Android must provide a Java proxy so that this method gets dispatched to the appropriate managed type. These Java proxy types are Java code that have the "same" base class and Java interface list as the managed type, implementing the same constructors and declaring any overridden base class and interface methods.

Android callable wrappers are generated by the **monodroid.exe** program during the build process, and are generated for all types that (directly or indirectly) inherit Java.Lang.Object.

Implementing Interfaces

There are times when you may need to implement an Android interface, (such as Android.Content.IComponentCallbacks).

All Android classes and interfaces extend the Android.Runtime.IJavaObject interface; therefore, all Android types must implement `IJavaObject`. Xamarin.Android takes advantage of this fact – it uses `IJavaObject` to provide Android with a Java proxy (an Android callable wrapper) for the given managed type. Because **monodroid.exe** only looks for `Java.Lang.Object` subclasses (which must implement `IJavaObject`), subclassing `Java.Lang.Object` provides us with a way to implement interfaces in managed code. For example:

```
class MyComponentCallbacks : Java.Lang.Object, Android.Content.IComponentCallbacks {
    public void OnConfigurationChanged (Android.Content.Res.Configuration newConfig) {
        // implementation goes here...
    }
    public void OnLowMemory () {
        // implementation goes here...
    }
}
```

Implementation Details

The remainder of this article provides implementation details subject to change without notice (and is presented here only because developers may be curious about what's going on under the hood).

For example, given the following C# source:

```
using System;
using Android.App;
using Android.OS;

namespace Mono.Samples.HelloWorld
{
    public class HelloAndroid : Activity
    {
        protected override void OnCreate (Bundle savedInstanceState)
        {
            base.OnCreate (savedInstanceState);
            SetContentView (R.layout.main);
        }
    }
}
```

The **mandroid.exe** program will generate the following Android Callable Wrapper:

```
package mono.samples.helloWorld;

public class HelloAndroid extends android.app.Activity {
    static final String __md_methods;
    static {
        __md_methods =
            "n_onCreate:(Landroid/os/Bundle;)V:GetOnCreate_Landroid_os_Bundle_Handler\n" +
            "";
        mono.android.Runtime.register (
                "Mono.Samples.HelloWorld.HelloAndroid, HelloWorld, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null",
                HelloAndroid.class,
                __md_methods);
    }

    public HelloAndroid ()
    {
        super ();
        if (getClass () == HelloAndroid.class)
            mono.android.TypeManager.Activate (
                "Mono.Samples.HelloWorld.HelloAndroid, HelloWorld, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null",
                "", this, new java.lang.Object[] { });
    }

    @Override
    public void onCreate (android.os.Bundle p0)
    {
        n_onCreate (p0);
    }

    private native void n_onCreate (android.os.Bundle p0);
}
```

Notice that the base class is preserved, and native method declarations are provided for each method that is overridden within managed code.

ExportAttribute and ExportFieldAttribute

Typically, Xamarin.Android automatically generates the Java code that comprises the ACW; this generation is based on the class and method names when a class derives from a Java class and overrides existing Java methods. However, in some scenarios, the code generation is not adequate, as outlined below:

- Android supports action names in layout XML attributes, for example the android:onClick XML attribute. When it is specified, the inflated View instance tries to look up the Java method.

- The java.io.Serializable interface requires `readObject` and `writeObject` methods. Since they are not members of this interface, our corresponding managed implementation does not expose these methods to Java code.

- The android.os.Parcelable interface expects that an implementation class must have a static field `CREATOR` of type `Parcelable.Creator`. The generated Java code requires some explicit field. With our standard scenario, there is no way to output field in Java code from managed code.

Because code generation does not provide a solution to generate arbitrary Java methods with arbitrary names, starting with Xamarin.Android 4.2, the ExportAttribute and ExportFieldAttribute were introduced to offer a solution to the above scenarios. Both attributes reside in the `Java.Interop` namespace:

- `ExportAttribute` – specifies a method name and its expected exception types (to give explicit "throws" in Java). When it is used on a method, the method will "export" a Java method that generates a dispatch code to the corresponding JNI invocation to the managed method. This can be used with `android:onClick` and `java.io.Serializable`.

- `ExportFieldAttribute` – specifies a field name. It resides on a method that works as a field initializer. This can be used with `android.os.Parcelable`.

Troubleshooting ExportAttribute and ExportFieldAttribute

- Packaging fails due to missing **Mono.Android.Export.dll** – if you used `ExportAttribute` or `ExportFieldAttribute` on some methods in your code or dependent libraries, you have to add **Mono.Android.Export.dll**. This assembly is isolated to support callback code from Java. It is separate from **Mono.Android.dll** as it adds additional size to the application.

- In Release build, `MissingMethodException` occurs for Export methods – In Release build, `MissingMethodException` occurs for Export methods. (This issue is fixed in the latest version of Xamarin.Android.)

ExportParameterAttribute

`ExportAttribute` and `ExportFieldAttribute` provide functionality that Java run-time code can use. This run-time code accesses managed code through the generated JNI methods driven by those attributes. As a result, there is no existing Java method that the managed method binds; hence, the Java method is generated from a managed method signature.

However, this case is not fully determinant. Most notably, this is true in some advanced mappings between managed types and Java types such as:

- InputStream
- OutputStream
- XmlPullParser
- XmlResourceParser

When types such as these are needed for exported methods, the `ExportParameterAttribute` must be used to explicitly give the corresponding parameter or return value a type.

Annotation Attribute

In Xamarin.Android 4.2, we converted `IAnnotation` implementation types into attributes (System.Attribute), and added support for annotation generation in Java wrappers.

This means the following directional changes:

- The binding generator generates `Java.Lang.DeprecatedAttribute` from `java.Lang.Deprecated` (while it should be `[Obsolete]` in managed code).

- This does not mean that existing `Java.Lang.Deprecated` class will vanish. These Java-based objects could be still used as usual Java objects (if such usage exists). There will be `Deprecated` and `DeprecatedAttribute` classes.
- The `Java.Lang.DeprecatedAttribute` class is marked as `[Annotation]` . When there is a custom attribute that is inherited from this `[Annotation]` attribute, msbuild task will generate a Java annotation for that custom attribute (@Deprecated) in the Android Callable Wrapper (ACW).
- Annotations could be generated onto classes, methods and exported fields (which is a method in managed code).

If the containing class (the annotated class itself, or the class that contains the annotated members) is not registered, the entire Java class source is not generated at all, including annotations. For methods, you can specify the `ExportAttribute` to get the method explicitly generated and annotated. Also, it is not a feature to "generate" a Java annotation class definition. In other words, if you define a custom managed attribute for a certain annotation, you'll have to add another .jar library that contains the corresponding Java annotation class. Adding a Java source file that defines the annotation type is not sufficient. The Java compiler does not work in the same way as **apt**.

Additionally the following limitations apply:

- This conversion process does not consider `@Target` annotation on the annotation type so far.
- Attributes onto a property does not work. Use attributes for property getter or setter instead.

Class Binding

Binding a class means writing a managed callable wrapper to simplify invocation of the underlying Java type.

Binding virtual and abstract methods to permit overriding from C# requires Xamarin.Android 4.0. However, any version of Xamarin.Android can bind non-virtual methods, static methods, or virtual methods without supporting overrides.

A binding typically contains the following items:

- A JNI handle to the Java type being bound.
- JNI field IDs and properties for each bound field.
- JNI method IDs and methods for each bound method.
- If sub-classing is required, the type needs to have a RegisterAttribute custom attribute on the type declaration with RegisterAttribute.DoNotGenerateAcw set to `true` .

Declaring Type Handle

The field and method lookup methods require an object reference referring to their declaring type. By convention, this is held in a `class_ref` field:

```
static IntPtr class_ref = JNIEnv.FindClass(CLASS);
```

See the JNI Type References section for details about the `CLASS` token.

Binding Fields

Java fields are exposed as C# properties, for example the Java field java.lang.System.in is bound as the C# property Java.Lang.JavaSystem.In. Furthermore, since JNI distinguishes between static fields and instance fields, different methods be used when implementing the properties.

Field binding involves three sets of methods:

1. The *get field id* method. The *get field id* method is responsible for returning a field handle that the *get field value* and *set field value* methods will use. Obtaining the field id requires knowing the declaring type, the name of the field, and the JNI type signature of the field.

2. The *get field value* methods. These methods require the field handle and are responsible for reading the field's value from Java. The method to use depends upon the field's type.

3. The *set field value* methods. These methods require the field handle and are responsible for writing the field's value within Java. The method to use depends upon the field's type.

Static fields use the JNIEnv.GetStaticFieldID, `JNIEnv.GetStatic*Field`, and JNIEnv.SetStaticField methods.

Instance fields use the JNIEnv.GetFieldID, `JNIEnv.Get*Field`, and JNIEnv.SetField methods.

For example, the static property `JavaSystem.In` can be implemented as:

```
static IntPtr in_jfieldID;
public static System.IO.Stream In
{
    get {
        if (in_jfieldId == IntPtr.Zero)
            in_jfieldId = JNIEnv.GetStaticFieldID (class_ref, "in", "Ljava/io/InputStream;");
        IntPtr __ret = JNIEnv.GetStaticObjectField (class_ref, in_jfieldId);
        return InputStreamInvoker.FromJniHandle (__ret, JniHandleOwnership.TransferLocalRef);
    }
}
```

Note: We're using InputStreamInvoker.FromJniHandle to convert the JNI reference into a `System.IO.Stream` instance, and we're using `JniHandleOwnership.TransferLocalRef` because JNIEnv.GetStaticObjectField returns a local reference.

Many of the Android.Runtime types have `FromJniHandle` methods which will convert a JNI reference into the desired type.

Method Binding

Java methods are exposed as C# methods and as C# properties. For example, the Java method java.lang.Runtime.runFinalizersOnExit method is bound as the Java.Lang.Runtime.RunFinalizersOnExit method, and the java.lang.Object.getClass method is bound as the Java.Lang.Object.Class property.

Method invocation is a two-step process:

1. The *get method id* for the method to invoke. The *get method id* method is responsible for returning a method handle that the method invocation methods will use. Obtaining the method id requires knowing the declaring type, the name of the method, and the JNI type signature of the method.

2. Invoke the method.

Just as with fields, the methods to use to get the method id and invoke the method differ between static methods and instance methods.

Static methods use JNIEnv.GetStaticMethodID() to lookup the method id, and use the `JNIEnv.CallStatic*Method` family of methods for invocation.

Instance methods use JNIEnv.GetMethodID to lookup the method id, and use the `JNIEnv.Call*Method` and `JNIEnv.CallNonvirtual*Method` families of methods for invocation.

Method binding is potentially more than just method invocation. Method binding also includes allowing a method to be overridden (for abstract and non-final methods) or implemented (for interface methods). The

Supporting Inheritance, Interfaces section covers the complexities of supporting virtual methods and interface methods.

Static Methods

Binding a static method involves using `JNIEnv.GetStaticMethodID` to obtain a method handle, then using the appropriate `JNIEnv.CallStatic*Method` method, depending on the method's return type. The following is an example of a binding for the Runtime.getRuntime method:

```
static IntPtr id_getRuntime;

[Register ("getRuntime", "()Ljava/lang/Runtime;", "")]
public static Java.Lang.Runtime GetRuntime ()
{
    if (id_getRuntime == IntPtr.Zero)
        id_getRuntime = JNIEnv.GetStaticMethodID (class_ref,
                "getRuntime", "()Ljava/lang/Runtime;");

    return Java.Lang.Object.GetObject<Java.Lang.Runtime> (
            JNIEnv.CallStaticObjectMethod  (class_ref, id_getRuntime),
            JniHandleOwnership.TransferLocalRef);
}
```

Note that we store the method handle in a static field, `id_getRuntime`. This is a performance optimization, so that the method handle doesn't need to be looked up on every invocation. It is not necessary to cache the method handle in this way. Once the method handle is obtained, JNIEnv.CallStaticObjectMethod is used to invoke the method. `JNIEnv.CallStaticObjectMethod` returns an `IntPtr` which contains the handle of the returned Java instance. Java.Lang.Object.GetObject<T>(IntPtr, JniHandleOwnership) is used to convert the Java handle into a strongly typed object instance.

Non-virtual Instance Method Binding

Binding a `final` instance method, or an instance method which doesn't require overriding, involves using `JNIEnv.GetMethodID` to obtain a method handle, then using the appropriate `JNIEnv.Call*Method` method, depending on the method's return type. The following is an example of a binding for the `Object.Class` property:

```
static IntPtr id_getClass;
public Java.Lang.Class Class {
    get {
        if (id_getClass == IntPtr.Zero)
            id_getClass = JNIEnv.GetMethodID (class_ref, "getClass", "()Ljava/lang/Class;");
        return Java.Lang.Object.GetObject<Java.Lang.Class> (
                JNIEnv.CallObjectMethod (Handle, id_getClass),
                JniHandleOwnership.TransferLocalRef);
    }
}
```

Note that we store the method handle in a static field, `id_getClass`. This is a performance optimization, so that the method handle doesn't need to be looked up on every invocation. It is not necessary to cache the method handle in this way. Once the method handle is obtained, JNIEnv.CallStaticObjectMethod is used to invoke the method. `JNIEnv.CallStaticObjectMethod` returns an `IntPtr` which contains the handle of the returned Java instance. Java.Lang.Object.GetObject<T>(IntPtr, JniHandleOwnership) is used to convert the Java handle into a strongly typed object instance.

Binding Constructors

Constructors are Java methods with the name `"<init>"`. Just as with Java instance methods, `JNIEnv.GetMethodID` is used to lookup the constructor handle. Unlike Java methods, the JNIEnv.NewObject methods are used to invoke the constructor method handle. The return value of `JNIEnv.NewObject` is a JNI local reference:

```
int value = 42;
IntPtr class_ref    = JNIEnv.FindClass ("java/lang/Integer");
IntPtr id_ctor_I    = JNIEnv.GetMethodID (class_ref, "<init>", "(I)V");
IntPtr lrefInstance = JNIEnv.NewObject (class_ref, id_ctor_I, new JValue (value));
// Dispose of lrefInstance, class_ref…
```

Normally a class binding will subclass Java.Lang.Object. When subclassing `Java.Lang.Object`, an additional semantic comes into play: a `Java.Lang.Object` instance maintains a global reference to a Java instance through the `Java.Lang.Object.Handle` property.

1. The `Java.Lang.Object` default constructor will allocate a Java instance.
2. If the type has a `RegisterAttribute`, and `RegisterAttribute.DoNotGenerateAcw` is `true`, then an instance of the `RegisterAttribute.Name` type is created through its default constructor.
3. Otherwise, the Android Callable Wrapper (ACW) corresponding to `this.GetType` is instantiated through its default constructor. Android Callable Wrappers are generated during package creation for every `Java.Lang.Object` subclass for which `RegisterAttribute.DoNotGenerateAcw` is not set to `true`.

For types which are not class bindings, this is the expected semantic: instantiating a `Mono.Samples.HelloWorld.HelloAndroid` C# instance should construct a Java `mono.samples.helloworld.HelloAndroid` instance which is a generated Android Callable Wrapper.

For class bindings, this may be the correct behavior if the Java type contains a default constructor and/or no other constructor needs to be invoked. Otherwise, a constructor must be provided which performs the following actions:

1. Invoking the Java.Lang.Object(IntPtr, JniHandleOwnership) instead of the default `Java.Lang.Object` constructor. This is needed to avoid creating a new Java instance.
2. Check the value of Java.Lang.Object.Handle before creating any Java instances. The `Object.Handle` property will have a value other than `IntPtr.Zero` if an Android Callable Wrapper was constructed in Java code, and the class binding is being constructed to contain the created Android Callable Wrapper instance. For example, when Android creates a `mono.samples.helloworld.HelloAndroid` instance, the Android Callable Wrapper will be created first , and the Java `HelloAndroid` constructor will create an instance of the corresponding `Mono.Samples.HelloWorld.HelloAndroid` type, with the `Object.Handle` property being set to the Java instance prior to constructor execution.
3. If the current runtime type is not the same as the declaring type, then an instance of the corresponding Android Callable Wrapper must be created, and use Object.SetHandle to store the handle returned by JNIEnv.CreateInstance.
4. If the current runtime type is the same as the declaring type, then invoke the Java constructor and use Object.SetHandle to store the handle returned by `JNIEnv.NewInstance` .

For example, consider the java.lang.Integer(int) constructor. This is bound as:

```
// Cache the constructor's method handle for later use
static IntPtr id_ctor_I;

// Need [Register] for subclassing
// RegisterAttribute.Name is always ".ctor"
// RegisterAttribute.Signature is tye JNI type signature of constructor
// RegisterAttribute.Connector is ignored; use ""
[Register (".ctor", "(I)V", "")]
public Integer (int value)
    // 1. Prevent Object default constructor execution
    : base (IntPtr.Zero, JniHandleOwnership.DoNotTransfer)
{
    // 2. Don't allocate Java instance if already allocated
    if (Handle != IntPtr.Zero)
        return;

    // 3. Derived type? Create Android Callable Wrapper
    if (GetType () != typeof (Integer)) {
        SetHandle (
                Android.Runtime.JNIEnv.CreateInstance (GetType (), "(I)V", new JValue (value)),
                JniHandleOwnership.TransferLocalRef);
        return;
    }

    // 4. Declaring type: lookup &amp; cache method id...
    if (id_ctor_I == IntPtr.Zero)
        id_ctor_I = JNIEnv.GetMethodID (class_ref, "<init>", "(I)V");
    // ...then create the Java instance and store
    SetHandle (
            JNIEnv.NewObject (class_ref, id_ctor_I, new JValue (value)),
            JniHandleOwnership.TransferLocalRef);
}
```

The JNIEnv.CreateInstance methods are helpers to perform a `JNIEnv.FindClass`, `JNIEnv.GetMethodID`, `JNIEnv.NewObject`, and `JNIEnv.DeleteGlobalReference` on the value returned from `JNIEnv.FindClass`. See the next section for details.

Supporting Inheritance, Interfaces

Subclassing a Java type or implementing a Java interface requires the generation of Android Callable Wrappers (ACWs) that are generated for every `Java.Lang.Object` subclass during the packaging process. ACW generation is controlled through the Android.Runtime.RegisterAttribute custom attribute.

For C# types, the `[Register]` custom attribute constructor requires one argument: the JNI simplified type reference for the corresponding Java type. This allows providing different names between Java and C#.

Prior to Xamarin.Android 4.0, the `[Register]` custom attribute was unavailable to "alias" existing Java types. This is because the ACW generation process would generate ACWs for every `Java.Lang.Object` subclass encountered.

Xamarin.Android 4.0 introduced the RegisterAttribute.DoNotGenerateAcw property. This property instructs the ACW generation process to *skip* the annotated type, allowing the declaration of new Managed Callable Wrappers that will not result in ACWs being generated at package creation time. This allows binding existing Java types. For instance, consider the following simple Java class, `Adder`, which contains one method, `add`, that adds to integers and returns the result:

```
package mono.android.test;
public class Adder {
    public int add (int a, int b) {
        return a + b;
    }
}
```

The `Adder` type could be bound as:

```
[Register ("mono/android/test/Adder", DoNotGenerateAcw=true)]
public partial class Adder : Java.Lang.Object {
    static IntPtr class_ref = JNIEnv.FindClass ( "mono/android/test/Adder");

    public Adder ()
    {
    }

    public Adder (IntPtr handle, JniHandleOwnership transfer)
        : base (handle, transfer)
    {
    }
}
partial class ManagedAdder : Adder {
}
```

Here, the `Adder` C# type *aliases* the `Adder` Java type. The `[Register]` attribute is used to specify the JNI name of the `mono.android.test.Adder` Java type, and the `DoNotGenerateAcw` property is used to inhibit ACW generation. This will result in the generation of an ACW for the `ManagedAdder` type, which properly subclasses the `mono.android.test.Adder` type. If the `RegisterAttribute.DoNotGenerateAcw` property hadn't been used, then the Xamarin.Android build process would have generated a new `mono.android.test.Adder` Java type. This would result in compilation errors, as the `mono.android.test.Adder` type would be present twice, in two separate files.

Binding Virtual Methods

`ManagedAdder` subclasses the Java `Adder` type, but it isn't particularly interesting: the C# `Adder` type doesn't define any virtual methods, so `ManagedAdder` can't override anything.

Binding `virtual` methods to permit overriding by subclasses requires several things that need to be done which fall into the following two categories:

1. Method Binding
2. Method Registration

Method Binding

A method binding requires the addition of two support members to the C# `Adder` definition: `ThresholdType`, and `ThresholdClass`.

ThresholdType

The `ThresholdType` property returns the current type of the binding:

```
partial class Adder {
    protected override System.Type ThresholdType {
        get {
            return typeof (Adder);
        }
    }
}
```

`ThresholdType` is used in the Method Binding to determine when it should perform virtual vs. non-virtual method dispatch. It should always return a `System.Type` instance which corresponds to the declaring C# type.

ThresholdClass

The `ThresholdClass` property returns the JNI class reference for the bound type:

```
partial class Adder {
    protected override IntPtr ThresholdClass {
        get {
            return class_ref;
        }
    }
}
```

`ThresholdClass` is used in the Method Binding when invoking non-virtual methods.

Binding Implementation

The method binding implementation is responsible for runtime invocation of the Java method. It also contains a `[Register]` custom attribute declaration that is part of the method registration, and will be discussed in the Method Registration section:

```
[Register ("add", "(II)I", "GetAddHandler")]
    public virtual int Add (int a, int b)
    {
        if (id_add == IntPtr.Zero)
            id_add = JNIEnv.GetMethodID (class_ref, "add", "(II)I");
        if (GetType () == ThresholdType)
            return JNIEnv.CallIntMethod (Handle, id_add, new JValue (a), new JValue (b));
        return JNIEnv.CallNonvirtualIntMethod (Handle, ThresholdClass, id_add, new JValue (a), new JValue
(b));
    }
}
```

The `id_add` field contains the method ID for the Java method to invoke. The `id_add` value is obtained from `JNIEnv.GetMethodID`, which requires the declaring class (`class_ref`), the Java method name (`"add"`), and the JNI signature of the method (`"(II)I"`).

Once the method ID is obtained, `GetType` is compared to `ThresholdType` to determine if virtual or non-virtual dispatch is required. Virtual dispatch is required when `GetType` matches `ThresholdType`, as `Handle` may refer to a Java-allocated subclass which overrides the method.

When `GetType` doesn't match `ThresholdType`, `Adder` has been subclassed (e.g. by `ManagedAdder`), and the `Adder.Add` implementation will only be invoked if the subclass invoked `base.Add`. This is the non-virtual dispatch case, which is where `ThresholdClass` comes in. `ThresholdClass` specifies which Java class will provide the implementation of the method to invoke.

Method Registration

Assume we have an updated `ManagedAdder` definition which overrides the `Adder.Add` method:

```
partial class ManagedAdder : Adder {
    public override int Add (int a, int b) {
        return (a*2) + (b*2);
    }
}
```

Recall that `Adder.Add` had a `[Register]` custom attribute:

```
[Register ("add", "(II)I", "GetAddHandler")]
```

The `[Register]` custom attribute constructor accepts three values:

1. The name of the Java method, `"add"` in this case.
2. The JNI Type Signature of the method, `"(II)I"` in this case.
3. The *connector method*, `GetAddHandler` in this case. Connector methods will be discussed later.

The first two parameters allow the ACW generation process to generate a method declaration to override the method. The resulting ACW would contain some of the following code:

```
public class ManagedAdder extends mono.android.test.Adder {
    static final String __md_methods;
    static {
        __md_methods = "n_add:(II)I:GetAddHandler\n" +
            "";
        mono.android.Runtime.register (...);
    }
    @Override
    public int add (int p0, int p1) {
        return n_add (p0, p1);
    }
    private native int n_add (int p0, int p1);
    // ...
}
```

Note that an `@Override` method is declared, which delegates to an `n_`-prefixed method of the same name. This ensure that when Java code invokes `ManagedAdder.add`, `ManagedAdder.n_add` will be invoked, which will allow the overriding C# `ManagedAdder.Add` method to be executed.

Thus, the most important question: how is `ManagedAdder.n_add` hooked up to `ManagedAdder.Add`?

Java `native` methods are registered with the Java (the Android runtime) runtime through the JNI RegisterNatives function. `RegisterNatives` takes an array of structures containing the Java method name, the JNI Type Signature, and a function pointer to invoke that follows JNI calling convention. The function pointer must be a function that takes two pointer arguments followed by the method parameters. The Java `ManagedAdder.n_add` method must be implemented through a function that has the following C prototype:

```
int FunctionName(JNIEnv *env, jobject this, int a, int b)
```

Xamarin.Android does not expose a `RegisterNatives` method. Instead, the ACW and the MCW together provide the information necessary to invoke `RegisterNatives`: the ACW contains the method name and the JNI type signature, the only thing missing is a function pointer to hook up.

This is where the *connector method* comes in. The third `[Register]` custom attribute parameter is the name of a method defined in the registered type or a base class of the registered type that accepts no parameters and returns a `System.Delegate`. The returned `System.Delegate` in turn refers to a method that has the correct JNI function signature. Finally, the delegate that the connector method returns *must* be rooted so that the GC doesn't collect it, as the delegate is being provided to Java.

```
#pragma warning disable 0169
static Delegate cb_add;
// This method must match the third parameter of the [Register]
// custom attribute, must be static, must return System.Delegate,
// and must accept no parameters.
static Delegate GetAddHandler ()
{
    if (cb_add == null)
        cb_add = JNINativeWrapper.CreateDelegate ((Func<IntPtr, IntPtr, int, int, int>) n_Add);
    return cb_add;
}
// This method is registered with JNI.
static int n_Add (IntPtr jnienv, IntPtr lrefThis, int a, int b)
{
    Adder __this = Java.Lang.Object.GetObject<Adder>(lrefThis, JniHandleOwnership.DoNotTransfer);
    return __this.Add (a, b);
}
#pragma warning restore 0169
```

The `GetAddHandler` method creates a `Func<IntPtr, IntPtr, int, int, int>` delegate which refers to the `n_Add` method, then invokes JNINativeWrapper.CreateDelegate. `JNINativeWrapper.CreateDelegate` wraps the provided method in a try/catch block, so that any unhandled exceptions are handled and will result in raising the AndroidEvent.UnhandledExceptionRaiser event. The resulting delegate is stored in the static `cb_add` variable so that the GC will not free the delegate.

Finally, the `n_Add` method is responsible for marshaling the JNI parameters to the corresponding managed types, then delegating the method call.

Note: Always use `JniHandleOwnership.DoNotTransfer` when obtaining an MCW over a Java instance. Treating them as a local reference (and thus calling `JNIEnv.DeleteLocalRef`) will break managed -> Java -> managed stack transitions.

Complete Adder Binding

The complete managed binding for the `mono.android.tests.Adder` type is:

```
[Register ("mono/android/test/Adder", DoNotGenerateAcw=true)]
public class Adder : Java.Lang.Object {

    static IntPtr class_ref = JNIEnv.FindClass ("mono/android/test/Adder");

    public Adder ()
    {
    }

    public Adder (IntPtr handle, JniHandleOwnership transfer)
        : base (handle, transfer)
    {
    }

    protected override Type ThresholdType {
        get {return typeof (Adder);}
    }

    protected override IntPtr ThresholdClass {
        get {return class_ref;}
    }

#region Add
    static IntPtr id_add;

    [Register ("add", "(II)I", "GetAddHandler")]
    public virtual int Add (int a, int b)
    {
        if (id_add == IntPtr.Zero)
            id_add = JNIEnv.GetMethodID (class_ref, "add", "(II)I");
        if (GetType () == ThresholdType)
            return JNIEnv.CallIntMethod (Handle, id_add, new JValue (a), new JValue (b));
        return JNIEnv.CallNonvirtualIntMethod (Handle, ThresholdClass, id_add, new JValue (a), new JValue
(b));
    }

#pragma warning disable 0169
    static Delegate cb_add;
    static Delegate GetAddHandler ()
    {
        if (cb_add == null)
            cb_add = JNINativeWrapper.CreateDelegate ((Func<IntPtr, IntPtr, int, int, int>) n_Add);
        return cb_add;
    }

    static int n_Add (IntPtr jnienv, IntPtr lrefThis, int a, int b)
    {
        Adder __this = Java.Lang.Object.GetObject<Adder>(lrefThis, JniHandleOwnership.DoNotTransfer);
        return __this.Add (a, b);
    }
#pragma warning restore 0169
#endregion
}
```

Restrictions

When writing a type that matches the following criteria:

1. Subclasses `Java.Lang.Object`
2. Has a `[Register]` custom attribute
3. `RegisterAttribute.DoNotGenerateAcw` is `true`

Then for GC interaction the type *must not* have any fields which may refer to a `Java.Lang.Object` or `Java.Lang.Object` subclass at runtime. For example, fields of type `System.Object` and any interface type are not permitted. Types which cannot refer to `Java.Lang.Object` instances are permitted, such as `System.String` and `List<int>`. This restriction is to prevent premature object collection by the GC.

If the type must contain an instance field that can refer to a `Java.Lang.Object` instance, then the field type must be `System.WeakReference` or `GCHandle`.

Binding Abstract Methods

Binding `abstract` methods is largely identical to binding virtual methods. There are only two differences:

1. The abstract method is abstract. It still retains the `[Register]` attribute and the associated Method Registration, the Method Binding is just moved to the `Invoker` type.
2. A non- `abstract` `Invoker` type is created which subclasses the abstract type. The `Invoker` type must override all abstract methods declared in the base class, and the overridden implementation is the Method Binding implementation, though the non-virtual dispatch case can be ignored.

For example, assume that the above `mono.android.test.Adder.add` method were `abstract`. The C# binding would change so that `Adder.Add` were abstract, and a new `AdderInvoker` type would be defined which implemented `Adder.Add`:

```
partial class Adder {
    [Register ("add", "(II)I", "GetAddHandler")]
    public abstract int Add (int a, int b);

    // The Method Registration machinery is identical to the
    // virtual method case...
}

partial class AdderInvoker : Adder {
    public AdderInvoker (IntPtr handle, JniHandleOwnership transfer)
        : base (handle, transfer)
    {
    }

    static IntPtr id_add;
    public override int Add (int a, int b)
    {
        if (id_add == IntPtr.Zero)
            id_add = JNIEnv.GetMethodID (class_ref, "add", "(II)I");
        return JNIEnv.CallIntMethod (Handle, id_add, new JValue (a), new JValue (b));
    }
}
```

The `Invoker` type is only necessary when obtaining JNI references to Java-created instances.

Binding Interfaces

Binding interfaces is conceptually similar to binding classes containing virtual methods, but many of the specifics differ in subtle (and not so subtle) ways. Consider the following Java interface declaration:

```
public interface Progress {
    void onAdd(int[] values, int currentIndex, int currentSum);
}
```

Interface bindings have two parts: the C# interface definition, and an Invoker definition for the interface.

Interface Definition

The C# interface definition must fulfill the following requirements:

- The interface definition must have a `[Register]` custom attribute.
- The interface definition must extend the `IJavaObject interface`. Failure to do so will prevent ACWs from inheriting from the Java interface.
- Each interface method must contain a `[Register]` attribute specifying the corresponding Java method name, the JNI signature, and the connector method.
- The connector method must also specify the type that the connector method can be located on.

When binding `abstract` and `virtual` methods, the connector method would be searched within the inheritance hierarchy of the type being registered. Interfaces can have no methods containing bodies, so this doesn't work, thus the requirement that a type be specified indicating where the connector method is located. The type is specified within the connector method string, after a colon `':'`, and must be the assembly qualified type name of the type containing the invoker.

Interface method declarations are a translation of the corresponding Java method using *compatible* types. For Java builtin types, the compatible types are the corresponding C# types, e.g. Java `int` is C# `int`. For reference types, the compatible type is a type that can provide a JNI handle of the appropriate Java type.

The interface members will not be directly invoked by Java – invocation will be mediated through the Invoker type – so some amount of flexibility is permitted.

The Java Progress interface can be declared in C# as:

```
[Register ("mono/android/test/Adder$Progress", DoNotGenerateAcw=true)]
public interface IAdderProgress : IJavaObject {
    [Register ("onAdd", "([III)V",
            "GetOnAddHandler:Mono.Samples.SanityTests.IAdderProgressInvoker, SanityTests, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null")]
    void OnAdd (JavaArray<int> values, int currentIndex, int currentSum);
}
```

Notice in the above that we map the Java `int[]` parameter to a JavaArray<int>. This isn't necessary: we could have bound it to a C# `int[]`, or an `IList<int>`, or something else entirely. Whatever type is chosen, the `Invoker` needs to be able to translate it into a Java `int[]` type for invocation.

Invoker Definition

The `Invoker` type definition must inherit `Java.Lang.Object`, implement the appropriate interface, and provide all connection methods referenced in the interface definition. There is one more suggestion that differs from a class binding: the `class_ref` field and method IDs should be instance members, not static members.

The reason for preferring instance members has to do with `JNIEnv.GetMethodID` behavior in the Android runtime. (This may be Java behavior as well; it hasn't been tested.) `JNIEnv.GetMethodID` returns null when looking up a method that comes from an implemented interface and not the declared interface. Consider the java.util.SortedMap<K, V> Java interface, which implements the java.util.Map<K, V> interface. Map provides a clear method, thus a seemingly reasonable `Invoker` definition for SortedMap would be:

```
// Fails at runtime. DO NOT FOLLOW
partial class ISortedMapInvoker : Java.Lang.Object, ISortedMap {
    static IntPtr class_ref = JNIEnv.FindClass ("java/util/SortedMap");
    static IntPtr id_clear;
    public void Clear()
    {
        if (id_clear == IntPtr.Zero)
            id_clear = JNIEnv.GetMethodID(class_ref, "clear", "()V");
        JNIEnv.CallVoidMethod(Handle, id_clear);
    }
     // ...
}
```

The above will fail because `JNIEnv.GetMethodID` will return `null` when looking up the `Map.clear` method through the `SortedMap` class instance.

There are two solutions to this: track which interface every method comes from, and have a `class_ref` for each interface, or keep everything as instance members and perform the method lookup on the most-derived class type, not the interface type. The latter is done in **Mono.Android.dll**.

The Invoker definition has six sections: the constructor, the `Dispose` method, the `ThresholdType` and `ThresholdClass` members, the `GetObject` method, interface method implementation, and the connector method implementation.

Constructor

The constructor needs to lookup the runtime class of the instance being invoked and store the runtime class in the instance `class_ref` field:

```
partial class IAdderProgressInvoker {
    IntPtr class_ref;
    public IAdderProgressInvoker (IntPtr handle, JniHandleOwnership transfer)
        : base (handle, transfer)
    {
        IntPtr lref = JNIEnv.GetObjectClass (Handle);
        class_ref   = JNIEnv.NewGlobalRef (lref);
        JNIEnv.DeleteLocalRef (lref);
    }
}
```

Note: The `Handle` property must be used within the constructor body, and not the `handle` parameter, as on Android v4.0 the `handle` parameter may be invalid after the base constructor finishes executing.

Dispose Method

The `Dispose` method needs to free the global reference allocated in the constructor:

```
partial class IAdderProgressInvoker {
    protected override void Dispose (bool disposing)
    {
        if (this.class_ref != IntPtr.Zero)
            JNIEnv.DeleteGlobalRef (this.class_ref);
        this.class_ref = IntPtr.Zero;
        base.Dispose (disposing);
    }
}
```

ThresholdType and ThresholdClass

The `ThresholdType` and `ThresholdClass` members are identical to what is found in a class binding:

```
partial class IAdderProgressInvoker {
    protected override Type ThresholdType {
        get {
            return typeof (IAdderProgressInvoker);
        }
    }
    protected override IntPtr ThresholdClass {
        get {
            return class_ref;
        }
    }
}
```

GetObject Method

A static `GetObject` method is required to support Extensions.JavaCast<T>():

```
partial class IAdderProgressInvoker {
    public static IAdderProgress GetObject (IntPtr handle, JniHandleOwnership transfer)
    {
        return new IAdderProgressInvoker (handle, transfer);
    }
}
```

Interface Methods

Every method of the interface needs to have an implementation, which invokes the corresponding Java method through JNI:

```
partial class IAdderProgressInvoker {
    IntPtr id_onAdd;
    public void OnAdd (JavaArray<int> values, int currentIndex, int currentSum)
    {
        if (id_onAdd == IntPtr.Zero)
            id_onAdd = JNIEnv.GetMethodID (class_ref, "onAdd", "([III)V");
        JNIEnv.CallVoidMethod (Handle, id_onAdd, new JValue (JNIEnv.ToJniHandle (values)), new JValue
(currentIndex), new JValue (currentSum));
    }
}
```

Connector Methods

The connector methods and supporting infrastructure are responsible for marshaling the JNI parameters to appropriate C# types. The Java `int[]` parameter will be passed as a JNI `jintArray`, which is an `IntPtr` within C#. The `IntPtr` must be marshaled to a `JavaArray<int>` in order to support invoking the C# interface:

```
partial class IAdderProgressInvoker {
    static Delegate cb_onAdd;
    static Delegate GetOnAddHandler ()
    {
        if (cb_onAdd == null)
            cb_onAdd = JNINativeWrapper.CreateDelegate ((Action<IntPtr, IntPtr, IntPtr, int, int>) n_OnAdd);
        return cb_onAdd;
    }

    static void n_OnAdd (IntPtr jnienv, IntPtr lrefThis, IntPtr values, int currentIndex, int currentSum)
    {
        IAdderProgress __this = Java.Lang.Object.GetObject<IAdderProgress>(lrefThis,
JniHandleOwnership.DoNotTransfer);
        using (var _values = new JavaArray<int>(values, JniHandleOwnership.DoNotTransfer)) {
            __this.OnAdd (_values, currentIndex, currentSum);
        }
    }
}
```

If `int[]` would be preferred over `JavaList<int>`, then JNIEnv.GetArray() could be used instead:

```
int[] _values = (int[]) JNIEnv.GetArray(values, JniHandleOwnership.DoNotTransfer, typeof (int));
```

Note, however, that `JNIEnv.GetArray` copies the entire array between VMs, so for large arrays this could result in lots of added GC pressure.

Complete Invoker Definition

The complete IAdderProgressInvoker definition:

```
class IAdderProgressInvoker : Java.Lang.Object, IAdderProgress {

    IntPtr class_ref;

    public IAdderProgressInvoker (IntPtr handle, JniHandleOwnership transfer)
        : base (handle, transfer)
    {
        IntPtr lref = JNIEnv.GetObjectClass (Handle);
        class_ref = JNIEnv.NewGlobalRef (lref);
        JNIEnv.DeleteLocalRef (lref);
    }

    protected override void Dispose (bool disposing)
    {
        if (this.class_ref != IntPtr.Zero)
            JNIEnv.DeleteGlobalRef (this.class_ref);
        this.class_ref = IntPtr.Zero;
        base.Dispose (disposing);
    }

    protected override Type ThresholdType {
        get {return typeof (IAdderProgressInvoker);}
    }

    protected override IntPtr ThresholdClass {
        get {return class_ref;}
    }

    public static IAdderProgress GetObject (IntPtr handle, JniHandleOwnership transfer)
    {
        return new IAdderProgressInvoker (handle, transfer);
    }

#region OnAdd
    IntPtr id_onAdd;
    public void OnAdd (JavaArray<int> values, int currentIndex, int currentSum)
    {
        if (id_onAdd == IntPtr.Zero)
            id_onAdd = JNIEnv.GetMethodID (class_ref, "onAdd",
                    "([III)V");
        JNIEnv.CallVoidMethod (Handle, id_onAdd,
                new JValue (JNIEnv.ToJniHandle (values)),
                new JValue (currentIndex),
new JValue (currentSum));
    }

#pragma warning disable 0169
    static Delegate cb_onAdd;
    static Delegate GetOnAddHandler ()
    {
        if (cb_onAdd == null)
            cb_onAdd = JNINativeWrapper.CreateDelegate ((Action<IntPtr, IntPtr, IntPtr, int, int>) n_OnAdd);
        return cb_onAdd;
    }

    static void n_OnAdd (IntPtr jnienv, IntPtr lrefThis, IntPtr values, int currentIndex, int currentSum)
    {
        IAdderProgress __this = Java.Lang.Object.GetObject<IAdderProgress>(lrefThis,
JniHandleOwnership.DoNotTransfer);
        using (var _values = new JavaArray<int>(values, JniHandleOwnership.DoNotTransfer)) {
            __this.OnAdd (_values, currentIndex, currentSum);
        }
    }
#pragma warning restore 0169
#endregion
}
```

JNI Object References

Many JNIEnv methods return *JNI object references*, which are similar to `GCHandle` s. JNI provides three different types of object references: local references, global references, and weak global references. All three are represented as `System.IntPtr` , *but* (as per the JNI Function Types section) not all `IntPtr` s returned from `JNIEnv` methods are references. For example, JNIEnv.GetMethodID returns an `IntPtr` , but it doesn't return an object reference, it returns a `jmethodID` . Consult the JNI function documentation for details.

Local references are created by *most* reference-creating methods. Android only allows a limited number of local references to exist at any given time, usually 512. Local references can be deleted via JNIEnv.DeleteLocalRef. Unlike JNI, not all reference JNIEnv methods which return object references return local references; JNIEnv.FindClass returns a *global* reference. It is strongly recommended that you delete local references as quickly as you can, possibly by constructing a Java.Lang.Object around the object and specifying `JniHandleOwnership.TransferLocalRef` to the Java.Lang.Object(IntPtr handle, JniHandleOwnership transfer) constructor.

Global references are created by JNIEnv.NewGlobalRef and JNIEnv.FindClass. They can be destroyed with JNIEnv.DeleteGlobalRef. Emulators have a limit of 2,000 outstanding global references, while hardware devices have a limit of around 52,000 global references.

Weak global references are only available on Android v2.2 (Froyo) and later. Weak global references can be deleted with JNIEnv.DeleteWeakGlobalRef.

Dealing With JNI Local References

The JNIEnv.GetObjectField, JNIEnv.GetStaticObjectField, JNIEnv.CallObjectMethod, JNIEnv.CallNonvirtualObjectMethod and JNIEnv.CallStaticObjectMethod methods return an `IntPtr` which contains a JNI local reference to a Java object, or `IntPtr.Zero` if Java returned `null` . Due to the limited number of local references that can be outstanding at once (512 entries), it is desirable to ensure that the references are deleted in a timely fashion. There are three ways that local references can be dealt with: explicitly deleting them, creating a `Java.Lang.Object` instance to hold them, and using `Java.Lang.Object.GetObject<T>()` to create a managed callable wrapper around them.

Explicitly Deleting Local References

JNIEnv.DeleteLocalRef is used to delete local references. Once the local reference has been deleted, it cannot be used anymore, so care must be taken to ensure that `JNIEnv.DeleteLocalRef` is the last thing done with the local reference.

```
IntPtr lref = JNIEnv.CallObjectMethod(instance, methodID);
try {
    // Do something with `lref`
}
finally {
    JNIEnv.DeleteLocalRef (lref);
}
```

Wrapping with Java.Lang.Object

`Java.Lang.Object` provides a Java.Lang.Object(IntPtr handle, JniHandleOwnership transfer) constructor which can be used to wrap an exiting JNI reference. The JniHandleOwnership parameter determines how the `IntPtr` parameter should be treated:

- JniHandleOwnership.DoNotTransfer – The created `Java.Lang.Object` instance will create a new global reference from the `handle` parameter, and `handle` is unchanged. The caller is responsible to freeing `handle` , if necessary.
- JniHandleOwnership.TransferLocalRef – The created `Java.Lang.Object` instance will create a new global

reference from the `handle` parameter, and `handle` is deleted with JNIEnv.DeleteLocalRef . The caller must not free `handle` , and must not use `handle` after the constructor finishes executing.

- JniHandleOwnership.TransferGlobalRef – The created `Java.Lang.Object` instance will take over ownership of the `handle` parameter. The caller must not free `handle` .

Since the JNI method invocation methods return local refs, `JniHandleOwnership.TransferLocalRef` would normally be used:

```
IntPtr lref = JNIEnv.CallObjectMethod(instance, methodID);
var value = new Java.Lang.Object (lref, JniHandleOwnership.TransferLocalRef);
```

The created global reference will not be freed until the `Java.Lang.Object` instance is garbage collected. If you are able to, disposing of the instance will free up the global reference, speeding up garbage collections:

```
IntPtr lref = JNIEnv.CallObjectMethod(instance, methodID);
using (var value = new Java.Lang.Object (lref, JniHandleOwnership.TransferLocalRef)) {
    // use value ...
}
```

Using Java.Lang.Object.GetObject<T>()

`Java.Lang.Object` provides a Java.Lang.Object.GetObject<T>(IntPtr handle, JniHandleOwnership transfer) method that can be used to create a managed callable wrapper of the specified type.

The type `T` must fulfill the following requirements:

1. `T` must be a reference type.

2. `T` must implement the `IJavaObject` interface.

3. If `T` is not an abstract class or interface, then `T` must provide a constructor with the parameter types `(IntPtr, JniHandleOwnership)` .

4. If `T` is an abstract class or an interface, there *must* be an *invoker* available for `T` . An invoker is a non-abstract type that inherits `T` or implements `T` , and has the same name as `T` with an Invoker suffix. For example, if T is the interface `Java.Lang.IRunnable` , then the type `Java.Lang.IRunnableInvoker` must exist and must contain the required `(IntPtr, JniHandleOwnership)` constructor.

Since the JNI method invocation methods return local refs, `JniHandleOwnership.TransferLocalRef` would normally be used:

```
IntPtr lrefString = JNIEnv.CallObjectMethod(instance, methodID);
Java.Lang.String value = Java.Lang.Object.GetObject<Java.Lang.String>( lrefString,
JniHandleOwnership.TransferLocalRef);
```

Looking up Java Types

To lookup a field or method in JNI, the declaring type for the field or method must be looked up first. The Android.Runtime.JNIEnv.FindClass(string)) method is used to lookup Java types. The string parameter is the *simplified type reference* or the *full type reference* for the Java type. See the JNI Type References section for details about simplified and full type references.

Note: Unlike every other `JNIEnv` method which returns object instances, `FindClass` returns a global reference, not a local reference.

Instance Fields

Fields are manipulated through *field IDs*. Field IDs are obtained via JNIEnv.GetFieldID, which requires the class that the field is defined in, the name of the field, and the JNI Type Signature of the field.

Field IDs do not need to be freed, and are valid as long as the corresponding Java type is loaded. (Android does not currently support class unloading.)

There are two sets of methods for manipulating instance fields: one for reading instance fields and one for writing instance fields. All sets of methods require a field ID to read or write the field value.

Reading Instance Field Values

The set of methods for reading instance field values follows the naming pattern:

```
* JNIEnv.Get*Field(IntPtr instance, IntPtr fieldID);
```

where `*` is the type of the field:

- JNIEnv.GetObjectField – Read the value of any instance field that isn't a builtin type, such as `java.lang.Object` , arrays, and interface types. The value returned is a JNI local reference.
- JNIEnv.GetBooleanField – Read the value of `bool` instance fields.
- JNIEnv.GetByteField – Read the value of `sbyte` instance fields.
- JNIEnv.GetCharField – Read the value of `char` instance fields.
- JNIEnv.GetShortField – Read the value of `short` instance fields.
- JNIEnv.GetIntField – Read the value of `int` instance fields.
- JNIEnv.GetLongField – Read the value of `long` instance fields.
- JNIEnv.GetFloatField – Read the value of `float` instance fields.
- JNIEnv.GetDoubleField – Read the value of `double` instance fields.

Writing Instance Field Values

The set of methods for writing instance field values follows the naming pattern:

```
JNIEnv.SetField(IntPtr instance, IntPtr fieldID, Type value);
```

where *Type* is the type of the field:

- JNIEnv.SetField) – Write the value of any field that isn't a builtin type, such as `java.lang.Object` , arrays, and interface types. The `IntPtr` value may be a JNI local reference, JNI global reference, JNI weak global reference, or `IntPtr.Zero` (for `null`).
- JNIEnv.SetField) – Write the value of `bool` instance fields.
- JNIEnv.SetField) – Write the value of `sbyte` instance fields.
- JNIEnv.SetField) – Write the value of `char` instance fields.
- JNIEnv.SetField) – Write the value of `short` instance fields.
- JNIEnv.SetField) – Write the value of `int` instance fields.
- JNIEnv.SetField) – Write the value of `long` instance fields.

- JNIEnv.SetField) – Write the value of `float` instance fields.
- JNIEnv.SetField) – Write the value of `double` instance fields.

Static Fields

Static Fields are manipulated through *field IDs*. Field IDs are obtained via JNIEnv.GetStaticFieldID, which requires the class that the field is defined in, the name of the field, and the JNI Type Signature of the field.

Field IDs do not need to be freed, and are valid as long as the corresponding Java type is loaded. (Android does not currently support class unloading.)

There are two sets of methods for manipulating static fields: one for reading instance fields and one for writing instance fields. All sets of methods require a field ID to read or write the field value.

Reading Static Field Values

The set of methods for reading static field values follows the naming pattern:

```
* JNIEnv.GetStatic*Field(IntPtr class, IntPtr fieldID);
```

where `*` is the type of the field:

- JNIEnv.GetStaticObjectField – Read the value of any static field that isn't a builtin type, such as `java.lang.Object` , arrays, and interface types. The value returned is a JNI local reference.
- JNIEnv.GetStaticBooleanField – Read the value of `bool` static fields.
- JNIEnv.GetStaticByteField – Read the value of `sbyte` static fields.
- JNIEnv.GetStaticCharField – Read the value of `char` static fields.
- JNIEnv.GetStaticShortField – Read the value of `short` static fields.
- JNIEnv.GetStaticLongField – Read the value of `long` static fields.
- JNIEnv.GetStaticFloatField – Read the value of `float` static fields.
- JNIEnv.GetStaticDoubleField – Read the value of `double` static fields.

Writing Static Field Values

The set of methods for writing static field values follows the naming pattern:

```
JNIEnv.SetStaticField(IntPtr class, IntPtr fieldID, Type value);
```

where *Type* is the type of the field:

- JNIEnv.SetStaticField) – Write the value of any static field that isn't a builtin type, such as `java.lang.Object` , arrays, and interface types. The `IntPtr` value may be a JNI local reference, JNI global reference, JNI weak global reference, or `IntPtr.Zero` (for `null`).
- JNIEnv.SetStaticField) – Write the value of `bool` static fields.
- JNIEnv.SetStaticField) – Write the value of `sbyte` static fields.
- JNIEnv.SetStaticField) – Write the value of `char` static fields.
- JNIEnv.SetStaticField) – Write the value of `short` static fields.

- JNIEnv.SetStaticField) – Write the value of `int` static fields.
- JNIEnv.SetStaticField) – Write the value of `long` static fields.
- JNIEnv.SetStaticField) – Write the value of `float` static fields.
- JNIEnv.SetStaticField) – Write the value of `double` static fields.

Instance Methods

Instance Methods are invoked through *method IDs*. Method IDs are obtained via JNIEnv.GetMethodID, which requires the type that the method is defined in, the name of the method, and the JNI Type Signature of the method.

Method IDs do not need to be freed, and are valid as long as the corresponding Java type is loaded. (Android does not currently support class unloading.)

There are two sets of methods for invoking methods: one for invoking methods virtually, and one for invoking methods non-virtually. Both sets of methods require a method ID to invoke the method, and non-virtual invocation also requires that you specify which class implementation should be invoked.

Interface methods can only be looked up within the declaring type; methods that come from extended/inherited interfaces cannot be looked up. See the later Binding Interfaces / Invoker Implementation section for more details.

Any method declared in the class or any base class or implemented interface can be looked up.

Virtual Method Invocation

The set of methods for invoking methods virtually follows the naming pattern:

```
* JNIEnv.Call*Method( IntPtr instance, IntPtr methodID, params JValue[] args );
```

where `*` is the return type of the method.

- JNIEnv.CallObjectMethod – Invoke a method which returns a non-builtin type, such as `java.lang.Object` , arrays, and interfaces. The value returned is a JNI local reference.
- JNIEnv.CallBooleanMethod – Invoke a method which returns a `bool` value.
- JNIEnv.CallByteMethod – Invoke a method which returns a `sbyte` value.
- JNIEnv.CallCharMethod – Invoke a method which returns a `char` value.
- JNIEnv.CallShortMethod – Invoke a method which returns a `short` value.
- JNIEnv.CallLongMethod – Invoke a method which returns a `long` value.
- JNIEnv.CallFloatMethod – Invoke a method which returns a `float` value.
- JNIEnv.CallDoubleMethod – Invoke a method which returns a `double` value.

Non-virtual Method Invocation

The set of methods for invoking methods non-virtually follows the naming pattern:

```
* JNIEnv.CallNonvirtual*Method( IntPtr instance, IntPtr class, IntPtr methodID, params JValue[] args );
```

where `*` is the return type of the method. Non-virtual method invocation is usually used to invoke the base method of a virtual method.

- JNIEnv.CallNonvirtualObjectMethod – Non-virtually invoke a method which returns a non-builtin type, such as `java.lang.Object` , arrays, and interfaces. The value returned is a JNI local reference.
- JNIEnv.CallNonvirtualBooleanMethod – Non-virtually invoke a method which returns a `bool` value.
- JNIEnv.CallNonvirtualByteMethod – Non-virtually invoke a method which returns a `sbyte` value.
- JNIEnv.CallNonvirtualCharMethod – Non-virtually invoke a method which returns a `char` value.
- JNIEnv.CallNonvirtualShortMethod – Non-virtually invoke a method which returns a `short` value.
- JNIEnv.CallNonvirtualLongMethod – Non-virtually invoke a method which returns a `long` value.
- JNIEnv.CallNonvirtualFloatMethod – Non-virtually invoke a method which returns a `float` value.
- JNIEnv.CallNonvirtualDoubleMethod – Non-virtually invoke a method which returns a `double` value.

Static Methods

Static Methods are invoked through *method IDs*. Method IDs are obtained via JNIEnv.GetStaticMethodID, which requires the type that the method is defined in, the name of the method, and the JNI Type Signature of the method.

Method IDs do not need to be freed, and are valid as long as the corresponding Java type is loaded. (Android does not currently support class unloading.)

Static Method Invocation

The set of methods for invoking methods virtually follows the naming pattern:

```
* JNIEnv.CallStatic*Method( IntPtr class, IntPtr methodID, params JValue[] args );
```

where `*` is the return type of the method.

- JNIEnv.CallStaticObjectMethod – Invoke a static method which returns a non-builtin type, such as `java.lang.Object` , arrays, and interfaces. The value returned is a JNI local reference.
- JNIEnv.CallStaticBooleanMethod – Invoke a static method which returns a `bool` value.
- JNIEnv.CallStaticByteMethod – Invoke a static method which returns a `sbyte` value.
- JNIEnv.CallStaticCharMethod – Invoke a static method which returns a `char` value.
- JNIEnv.CallStaticShortMethod – Invoke a static method which returns a `short` value.
- JNIEnv.CallStaticLongMethod – Invoke a static method which returns a `long` value.
- JNIEnv.CallStaticFloatMethod – Invoke a static method which returns a `float` value.
- JNIEnv.CallStaticDoubleMethod – Invoke a static method which returns a `double` value.

JNI Type Signatures

JNI Type Signatures are JNI Type References (though not simplified type references), except for methods. With methods, the JNI Type Signature is an open parenthesis `'('` , followed by the type references for all of the parameter types concatenated together (with no separating commas or anything else), followed by a closing parenthesis `')'` , followed by the JNI type reference of the method return type.

For example, given the Java method:

```
long f(int n, String s, int[] array);
```

The JNI type signature would be:

```
(ILjava/lang/String;[I)J
```

In general, it is *strongly* recommended to use the `javap` command to determine JNI signatures. For example, the JNI Type Signature of the java.lang.Thread.State.valueOf(String) method is "(Ljava/lang/String;)Ljava/lang/Thread$State;", while the JNI Type Signature of the java.lang.Thread.State.values method is "()[Ljava/lang/Thread$State;". Watch out for the trailing semicolons; those *are* part of the JNI type signature.

JNI Type References

JNI type references are different from Java type references. You cannot use fully qualified Java type names such as `java.lang.String` with JNI, you must instead use the JNI variations `"java/lang/String"` or `"Ljava/lang/String;"`, depending on context; see below for details. There are four types of JNI type references:

- built-in
- simplified
- type
- array

Built-in Type References

Built-in type references are a single character, used to reference built-in value types. The mapping is as follows:

- `"B"` for `sbyte` .
- `"S"` for `short` .
- `"I"` for `int` .
- `"J"` for `long` .
- `"F"` for `float` .
- `"D"` for `double` .
- `"C"` for `char` .
- `"Z"` for `bool` .
- `"V"` for `void` method return types.

Simplified Type References

Simplified type references can only be used in JNIEnv.FindClass(string)). There are two ways to derive a simplified type reference:

1. From a fully-qualified Java name, replace every `'.'` within the package name and before the type name with `'/'` , and every `'.'` within a type name with `'$'` .
2. Read the output of `'unzip -l android.jar | grep JavaName'` .

Either of the two will result in the Java type java.lang.Thread.State being mapped to the simplified type reference `java/lang/Thread$State`.

Type References

A type reference is a built-in type reference or a simplified type reference with an `'L'` prefix and a `';'` suffix. For the Java type java.lang.String, the simplified type reference is `"java/lang/String"`, while the type reference is `"Ljava/lang/String;"`.

Type references are used with Array type references and with JNI Signatures.

An additional way to obtain a type reference is by reading the output of `'javap -s -classpath android.jar fully.qualified.Java.Name'`. Depending on the type involved, you can use a constructor declaration or method return type to determine the JNI name. For example:

```
$ javap -classpath android.jar -s java.lang.Thread.State
Compiled from "Thread.java"
```

```
public final class java.lang.Thread$State extends java.lang.Enum{
public static final java.lang.Thread$State NEW;
  Signature: Ljava/lang/Thread$State;
public static final java.lang.Thread$State RUNNABLE;
  Signature: Ljava/lang/Thread$State;
public static final java.lang.Thread$State BLOCKED;
  Signature: Ljava/lang/Thread$State;
public static final java.lang.Thread$State WAITING;
  Signature: Ljava/lang/Thread$State;
public static final java.lang.Thread$State TIMED_WAITING;
  Signature: Ljava/lang/Thread$State;
public static final java.lang.Thread$State TERMINATED;
  Signature: Ljava/lang/Thread$State;
public static java.lang.Thread$State[] values();
  Signature: ()[Ljava/lang/Thread$State;
public static java.lang.Thread$State valueOf(java.lang.String);
  Signature: (Ljava/lang/String;)Ljava/lang/Thread$State;
static {};
  Signature: ()V
}
```

`Thread.State` is a Java enum type, so we can use the Signature of the `valueOf` method to determine that the type reference is Ljava/lang/Thread$State;.

Array Type References

Array type references are `'['` prefixed to a JNI type reference. Simplified type references cannot be used when specifying arrays.

For example, `int[]` is `"[I"`, `int[][]` is `"[[I"`, and `java.lang.Object[]` is `"[Ljava/lang/Object;"`.

Java Generics and Type Erasure

Most of the time, as seen through JNI, Java generics *do not exist*. There are some "wrinkles," but those wrinkles are in how Java interacts with generics, not with how JNI looks up and invokes generic members.

There is no difference between a generic type or member and a non-generic type or member when interacting through JNI. For example, the generic type java.lang.Class<T> is also the "raw" generic type `java.lang.Class`, both of which have the same simplified type reference, `"java/lang/Class"`.

Java Native Interface Support

Android.Runtime.JNIEnv is managed wrapper for the Jave Native Interface (JNI). JNI Functions are declared within the Java Native Interface Specification, though the methods have been changed to remove the explicit `JNIEnv*` parameter and `IntPtr` is used instead of `jobject`, `jclass`, `jmethodID`, etc. For example, consider the

JNI NewObject function:

```
jobject NewObjectA(JNIEnv *env, jclass clazz, jmethodID methodID, jvalue *args);
```

This is exposed as the JNIEnv.NewObject method:

```
public static IntPtr NewObject(IntPtr clazz, IntPtr jmethod, params JValue[] parms);
```

Translating between the two calls is reasonably straightforward. In C you would have:

```
jobject CreateMapActivity(JNIEnv *env)
{
    jclass    Map_Class   = (*env)->FindClass(env, "mono/samples/googlemaps/MyMapActivity");
    jmethodID Map_defCtor = (*env)->GetMethodID (env, Map_Class, "<init>", "()V");
    jobject   instance    = (*env)->NewObject (env, Map_Class, Map_defCtor);

    return instance;
}
```

The C# equivalent would be:

```
IntPtr CreateMapActivity()
{
    IntPtr Map_Class   = JNIEnv.FindClass ("mono/samples/googlemaps/MyMapActivity");
    IntPtr Map_defCtor = JNIEnv.GetMethodID (Map_Class, "<init>", "()V");
    IntPtr instance    = JNIEnv.NewObject (Map_Class, Map_defCtor);

    return instance;
}
```

Once you have a Java Object instance held in an IntPtr, you'll probably want to do something with it. You can use JNIEnv methods such as JNIEnv.CallVoidMethod() to do so, but if there is already an analogue C# wrapper then you'll want to construct a wrapper over the JNI reference. You can do so through the Extensions.JavaCast<T> extension method:

```
IntPtr lrefActivity = CreateMapActivity();

// imagine that Activity were instead an interface or abstract type...
Activity mapActivity = new Java.Lang.Object(lrefActivity, JniHandleOwnership.TransferLocalRef)
    .JavaCast<Activity>();
```

You can also use the Java.Lang.Object.GetObject<T> method:

```
IntPtr lrefActivity = CreateMapActivity();

// imagine that Activity were instead an interface or abstract type...
Activity mapActivity = Java.Lang.Object.GetObject<Activity>(lrefActivity,
JniHandleOwnership.TransferLocalRef);
```

Furthermore, all of the JNI functions have been modified by removing the `JNIEnv*` parameter present in every JNI function.

Summary

Dealing directly with JNI is a terrible experience that should be avoided at all costs. Unfortunately, it's not always avoidable; hopefully this guide will provide some assistance when you hit the unbound Java cases with Mono for Android.

Related links

- Java Native Interface Specification
- Java Native Interface Functions

Porting Java to C# for Xamarin.Android

10/29/2019 • 2 minutes to read • Edit Online

This approach may be of interest to organizations that:

- **Are switching technology stacks from Java to C#.**
- **Must maintain a C# and a Java version of the same product.**
- **Wish to have a .NET version of a popular Java library.**

There are two ways to port Java code to C#. The first way is to port the code manually. This involves skilled developers who understand both .NET and Java and are familiar with the proper idioms for each language. This approach makes the most sense for small amounts of code, or for organizations that wish to completely move away from Java to C#.

The second porting methodology is to try and automate the process by using a code converter, such as Sharpen. Sharpen is an open source converter from Versant that was originally used to port the code for *db4o* from Java to C#. db4o is an object-oriented database that Versant developed in Java, and then ported to .NET. Using a code converter may make sense for projects that must exist in both languages and that require some parity between the two.

An example of when an automated code conversion tool makes sense can be seen in the ngit project. Ngit is a port of the Java project jgit. Jgit itself is a Java implementation of the Git source code management system. To generate C# code from Java, the ngit programmers use a custom automated system to extract the Java code from jgit, apply some patches to accommodate the conversion process, and then run Sharpen, which generates the C# code. This allows the ngit project to benefit from the continuous, ongoing work that is done on jgit.

There is often a non-trivial amount of work involved with bootstrapping an automated code conversion tool, and this may prove to be a barrier to use. In many cases, it may be simpler and easier to port Java to C# by hand.

Related links

- Sharpen Conversion Tool

Binding a Java Library

7/8/2021 • 7 minutes to read • Edit Online

The Android community has many Java libraries that you may want to use in your app; this guide explains how to incorporate Java libraries into your Xamarin.Android application by creating a Bindings Library.

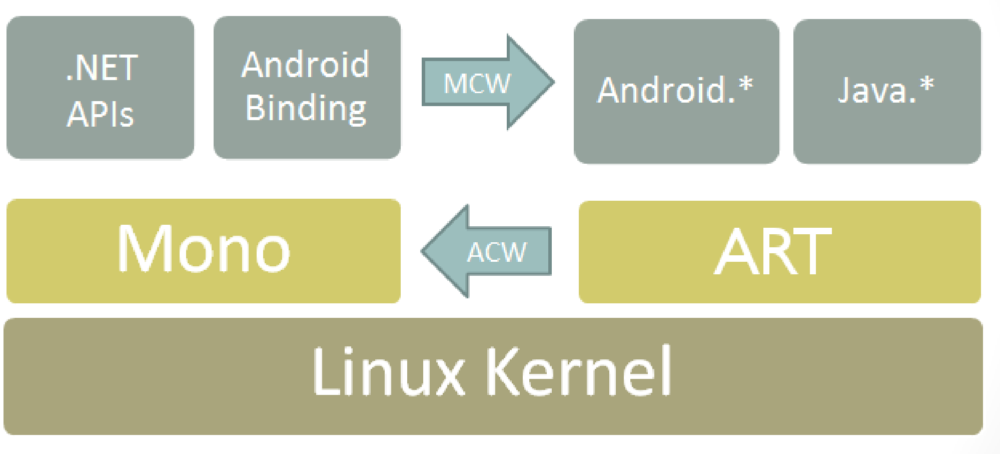
Overview

The third-party library ecosystem for Android is massive. Because of this, it frequently makes sense to use an existing Android library than to create a new one. Xamarin.Android offers two ways to use these libraries:

- Create a *Bindings Library* that automatically wraps the library with C# wrappers so you can invoke Java code via C# calls.
- Use the *Java Native Interface* (*JNI*) to invoke calls in Java library code directly. JNI is a programming framework that enables Java code to call and be called by native applications or libraries.

This guide explains the first option: how to create a *Bindings Library* that wraps one or more existing Java libraries into an assembly that you can link to in your application. For more information about using JNI, see Working with JNI.

Xamarin.Android implements bindings by using *Managed Callable Wrappers* (*MCW*). MCW is a JNI bridge that is used when managed code needs to invoke Java code. Managed callable wrappers also provide support for subclassing Java types and for overriding virtual methods on Java types. Likewise, whenever Android runtime (ART) code wishes to invoke managed code, it does so via another JNI bridge known as Android Callable Wrappers (ACW). This architecture is illustrated in the following diagram:



A Bindings Library is an assembly containing Managed Callable Wrappers for Java types. For example, here is a Java type, `MyClass`, that we want to wrap in a Bindings Library:

```
package com.xamarin.mycode;

public class MyClass
{
    public String myMethod (int i) { ... }
}
```

After we generate a Bindings Library for the **.jar** that contains `MyClass`, we can instantiate it and call methods on it from C#:

```
var instance = new MyClass ();

string result = instance.MyMethod (42);
```

To create this Bindings Library, you use the Xamarin.Android *Java Bindings Library* template. The resulting binding project creates a .NET assembly with the MCW classes, **.jar** file(s), and resources for Android Library projects embedded in it. You can also create Bindings Libraries for Android Archive (.AAR) files and Eclipse Android Library projects. By referencing the resulting Bindings Library DLL assembly, you can reuse an existing Java library in your Xamarin.Android project.

When you reference types in your Binding Library, you must use the namespace of your binding library. Typically, you add a `using` directive at the top of your C# source files that is the .NET namespace version of the Java package name. For example, if the Java package name for your bound **.jar** is the following:

```
com.company.package
```

Then you would put the following `using` statement at the top of your C# source files to access types in the bound **.jar** file:

```
using Com.Company.Package;
```

When binding an existing Android library, it is necessary to keep the following points in mind:

- **Are there any external dependencies for the library?** – Any Java dependencies required by the Android library must be included in the Xamarin.Android project as a **ReferenceJar** or as an **EmbeddedReferenceJar**. Any native assemblies must be added to the binding project as an **EmbeddedNativeLibrary**.
- **What version of the Android API does the Android library target?** – It is not possible to "downgrade" the Android API level; ensure that the Xamarin.Android binding project is targeting the same API level (or higher) as the Android library.
- **What version of the JDK was used to compile the library?** – Binding errors may occur if the Android library was built with a different version of JDK than in use by Xamarin.Android. If possible, recompile the Android library using the same version of the JDK that is used by your installation of Xamarin.Android.

Build Actions

When you create a Bindings Library, you set *build actions* on the **.jar** or .AAR files that you incorporate into your Bindings Library project – each build action determines how the **.jar** or .AAR file will be embedded into (or referenced by) your Bindings Library. The following list summarizes these build actions:

- `EmbeddedJar` – Embeds the **.jar** into the resulting Bindings Library DLL as an embedded resource. This is the simplest and most commonly-used build action. Use this option when you want the **.jar** automatically compiled into byte code and packaged into the Bindings Library.
- `InputJar` – Does not embed the **.jar** into the resulting Bindings Library .DLL. Your Bindings Library .DLL will have a dependency on this **.jar** at runtime. Use this option when you do not want to include the **.jar** in your Bindings Library (for example, for licensing reasons). If you use this option, you must ensure that the input **.jar** is available on the device that runs your app.
- `LibraryProjectZip` – Embeds an .AAR file into the resulting Bindings Library .DLL. This is similar to EmbeddedJar, except that you can access resources (as well as code) in the bound .AAR file. Use this

option when you want to embed an .AAR into your Bindings Library.

- `ReferenceJar` – Specifies a reference **.jar**: a reference **.jar** is a **.jar** that one of your bound **.jar** or .AAR files depends on. This reference **.jar** is used only to satisfy compile-time dependencies. When you use this build action, C# bindings are not created for the reference **.jar** and it is not embedded in the resulting Bindings Library .DLL. Use this option when you will make a Bindings Library for the reference **.jar** but have not done so yet. This build action is useful for packaging multiple **.jar**s (and/or .AARs) into multiple interdependent Bindings Libraries.

- `EmbeddedReferenceJar` – Embeds a reference **.jar** into the resulting Bindings Library .DLL. Use this build action when you want to create C# bindings for both the input **.jar** (or .AAR) and all of its reference **.jar**(s) in your Bindings Library.

- `EmbeddedNativeLibrary` – Embeds a native **.so** into the binding. This build action is used for **.so** files that are required by the **.jar** file being bound. It may be necessary to manually load the **.so** library before executing code from the Java library. This is described below.

These build actions are explained in more detail in the following guides.

Additionally, the following build actions are used to help importing Java API documentation and convert them into C# XML documentation:

- `JavaDocJar` is used to point to Javadoc archive Jar for a Java library that conforms to a Maven package style (usually `FOOBAR-javadoc**.jar**`).
- `JavaDocIndex` is used to point to `index.html` file within the API reference documentation HTML.
- `JavaSourceJar` is used to complement `JavaDocJar`, to first generate JavaDoc from sources and then treat the results as `JavaDocIndex`, for a Java library that conforms to a Maven package style (usually `FOOBAR-sources**.jar**`).

The API documentation should be the default doclet from Java8, Java7 or Java6 SDK (they are all different format), or the DroidDoc style.

Including a Native Library in a Binding

It may be necessary to include a **.so** library in a Xamarin.Android binding project as a part of binding a Java library. When the wrapped Java code executes, Xamarin.Android will fail to make the JNI call and the error message *java.lang.UnsatisfiedLinkError: Native method not found:* will appear in the logcat out for the application.

The fix for this is to manually load the **.so** library with a call to `Java.Lang.JavaSystem.LoadLibrary`. For example assuming that a Xamarin.Android project has shared library **libpocketsphinx_jni.so** included in the binding project with a build action of **EmbeddedNativeLibrary**, the following snippet (executed before using the shared library) will load the **.so** library:

```
Java.Lang.JavaSystem.LoadLibrary("pocketsphinx_jni");
```

Adapting Java APIs to C

The Xamarin.Android Binding Generator will change some Java idioms and patterns to correspond to .NET patterns. The following list describes how Java is mapped to C#/.NET:

- *Setter/Getter methods* in Java are *Properties* in .NET.

- *Fields* in Java are *Properties* in .NET.

- *Listeners/Listener Interfaces* in Java are *Events* in .NET. The parameters of methods in the callback interfaces will be represented by an `EventArgs` subclass.

- A *Static Nested class* in Java is a *Nested class* in .NET.
- An *Inner class* in Java is a *Nested class* with an instance constructor in C#.

Binding Scenarios

The following binding scenario guides can help you bind a Java library (or libraries) for incorporation into your app:

- Binding a .JAR is a walkthrough for creating Bindings Libraries for **.jar** files.
- Binding an .AAR is a walkthrough for creating Bindings Libraries for .AAR files. Read this walkthrough to learn how to bind Android Studio libraries.
- Binding an Eclipse Library Project is a walkthrough for creating binding libraries from Android Library Projects. Read this walkthrough to learn how to bind Eclipse Android Library Projects.
- Customizing Bindings explains how to make manual modifications to the binding to resolve build errors and shape the resulting API so that it is more "C#-like".
- Troubleshooting Bindings lists common binding error scenarios, explains possible causes, and offers suggestions for resolving these errors.

Related Links

- Working with JNI
- GAPI Metadata
- Using Native Libraries

Binding a .JAR

7/8/2021 • 6 minutes to read • Edit Online

IMPORTANT

We're currently investigating custom binding usage on the Xamarin platform. Please take **this survey** to inform future development efforts.

This walkthrough provides step-by-step instructions for creating a Xamarin.Android Java Bindings Library from an Android .JAR file.

Overview

The Android community offers many Java libraries that you may want to use in your app. These Java libraries are often packaged in .JAR (Java Archive) format, but you can package a .JAR it in a *Java Bindings Library* so that its functionality is available to Xamarin.Android apps. The purpose of the Java Bindings library is to make the APIs in the .JAR file available to C# code through automatically-generated code wrappers.

Xamarin tooling can generate a Bindings Library from one or more input .JAR files. The Bindings Library (.DLL assembly) contains the following:

- The contents of the original .JAR file(s).
- Managed Callable Wrappers (MCW), which are C# types that wrap corresponding Java types within the .JAR file(s).

The generated MCW code uses JNI (Java Native Interface) to forward your API calls to the underlying .JAR file. You can create bindings libraries for any .JAR file that was originally targeted to be used with Android (note that Xamarin tooling does not currently support the binding of non-Android Java libraries). You can also elect to build the Bindings Library without including the contents of the .JAR file so that the DLL has a dependency on the .JAR at runtime.

In this guide, we'll step through the basics of creating a Bindings Library for a single .JAR file. We'll illustrate with an example where everything goes right – that is, where no customization or debugging of bindings is required. Creating Bindings Using Metadata offers an example of a more advanced scenario where the binding process is not entirely automatic and some amount of manual intervention is required. For an overview of Java library binding in general (with a basic code example), see Binding a Java Library.

Walkthrough

In the following walkthrough, we'll create a Bindings Library for Picasso, a popular Android .JAR that provides image loading and caching functionality. We will use the following steps to bind **picasso-2.x.x.jar** to create a new .NET assembly that we can use in a Xamarin.Android project:

1. Create a new Java Bindings Library project.
2. Add the .JAR file to the project.
3. Set the appropriate build action for the .JAR file.
4. Choose a target framework that the .JAR supports.
5. Build the Bindings Library.

Once we've created the Bindings Library, we'll develop a small Android app that demonstrates our ability to call APIs in the Bindings Library. In this example, we want to access methods of **picasso-2.x.x.jar**:

```
package com.squareup.picasso

public class Picasso
{
    ...
    public static Picasso with (Context context) { ... };
    ...
    public RequestCreator load (String path) { ... };
    ...
}
```

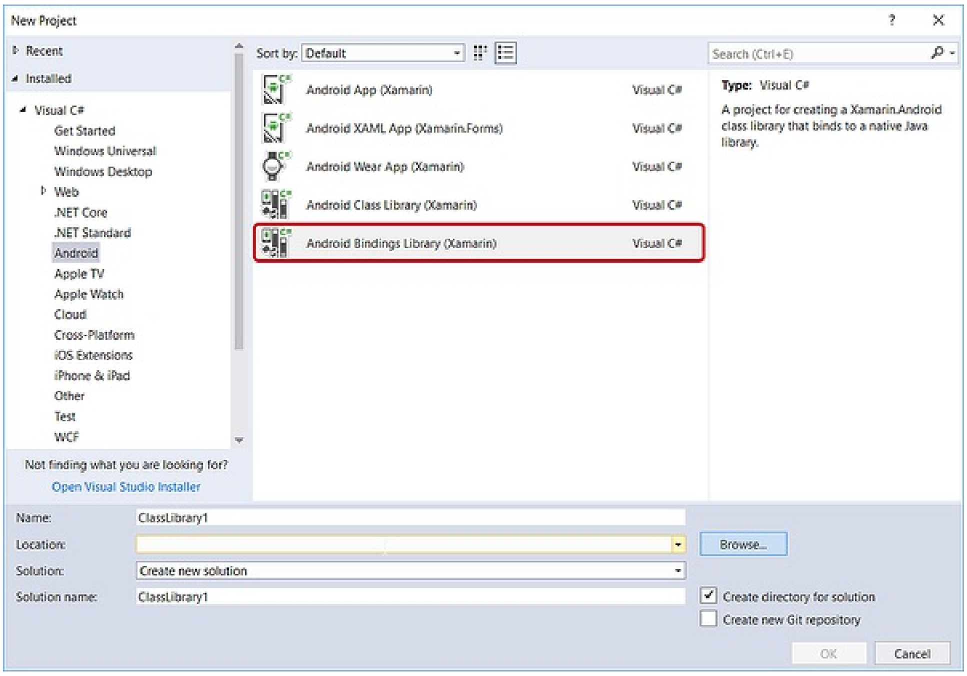
After we generate a Bindings Library for **picasso-2.x.x.jar**, we can call these methods from C#. For example:

```
using Com.Squareup.Picasso;
...
Picasso.With (this)
    .Load ("https://mydomain.myimage.jpg")
    .Into (imageView);
```

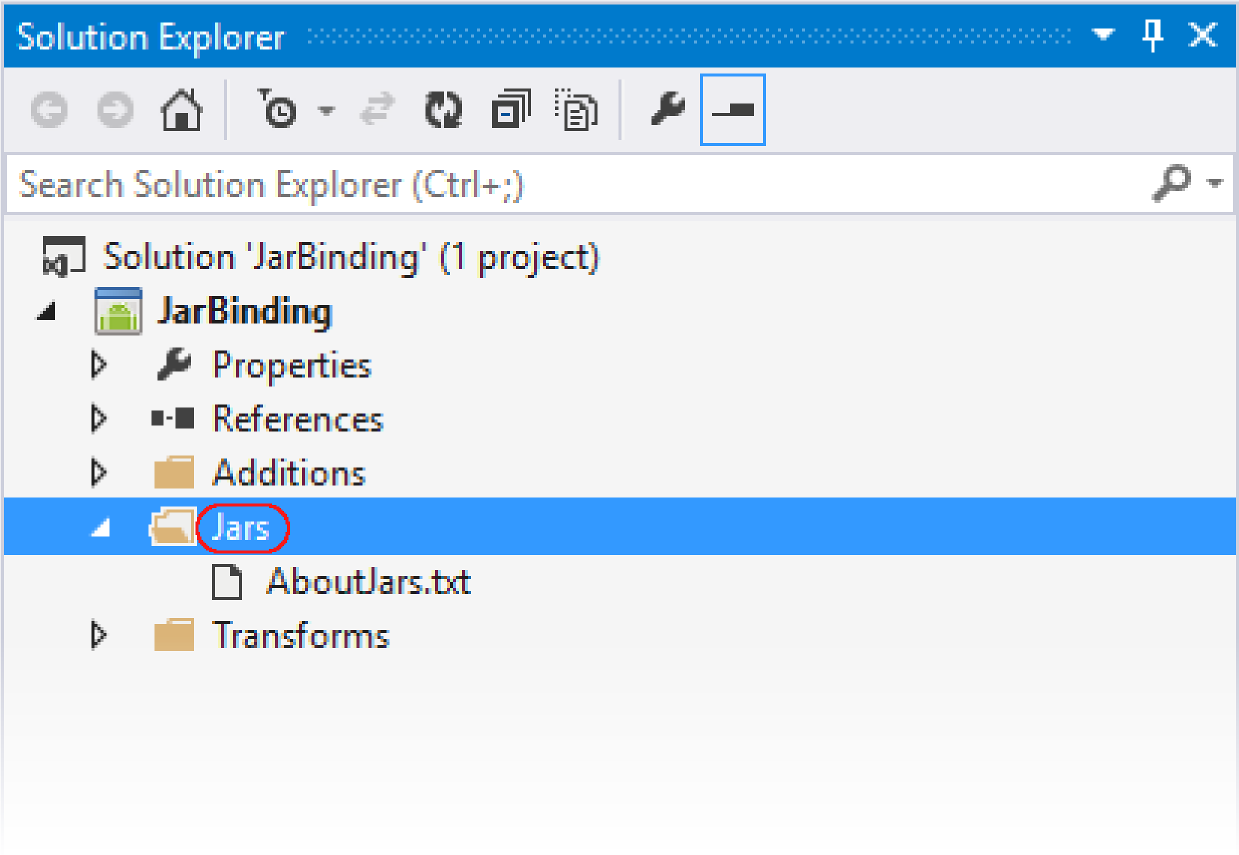
Creating the Bindings Library

Before commencing with the steps below, please download picasso-2.x.x.jar.

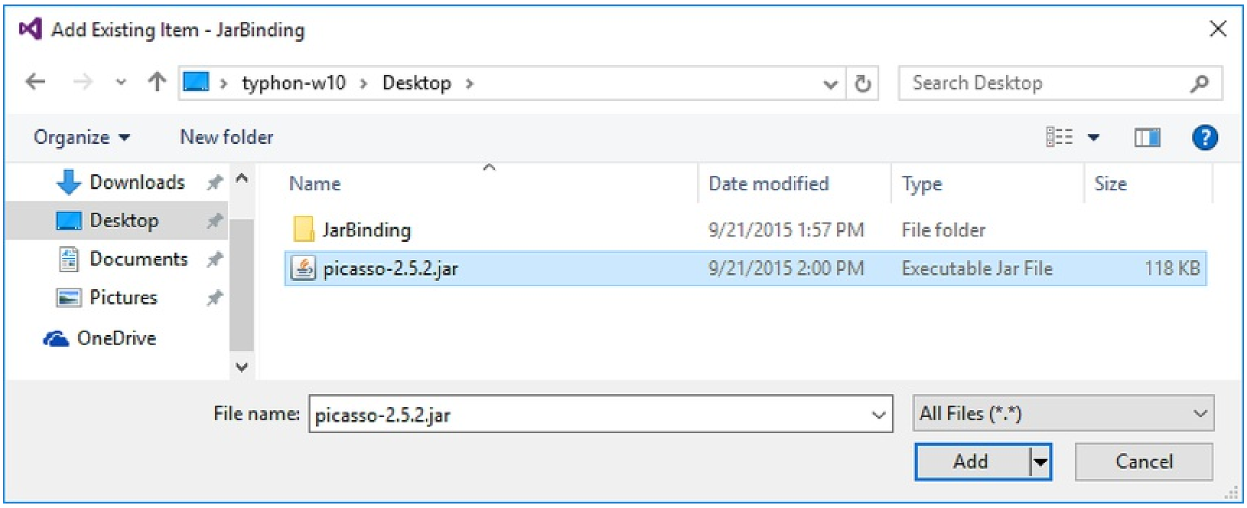
First, create a new Bindings Library project. In Visual Studio for Mac or Visual Studio, create a new Solution and select the *Android Bindings Library* template. (The screenshots in this walkthrough use Visual Studio, but Visual Studio for Mac is very similar.) Name the Solution **JarBinding**:



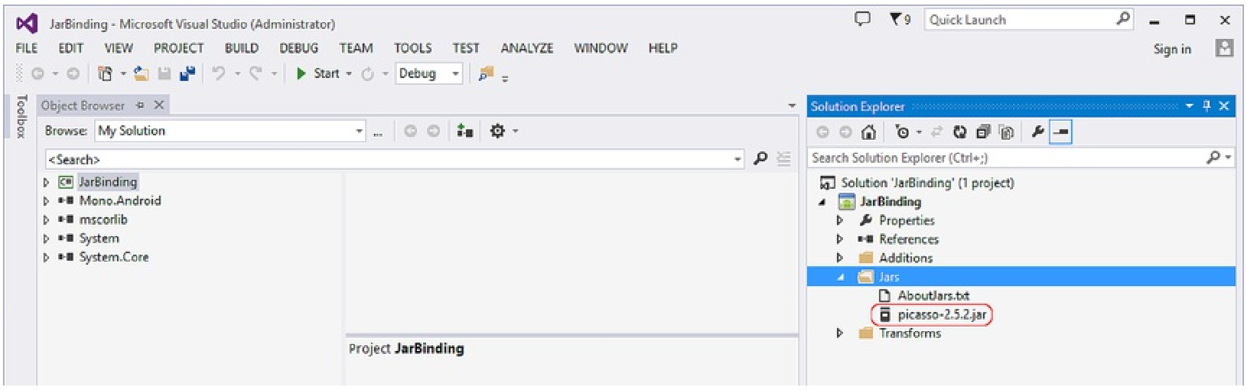
The template includes a **Jars** folder where you add your .JAR(s) to the Bindings Library project. Right-click the **Jars** folder and select **Add > Existing Item**:



Navigate to the **picasso-2.x.x.jar** file downloaded earlier, select it and click **Add**:



Verify that the **picasso-2.x.x.jar** file was successfully added to the project:

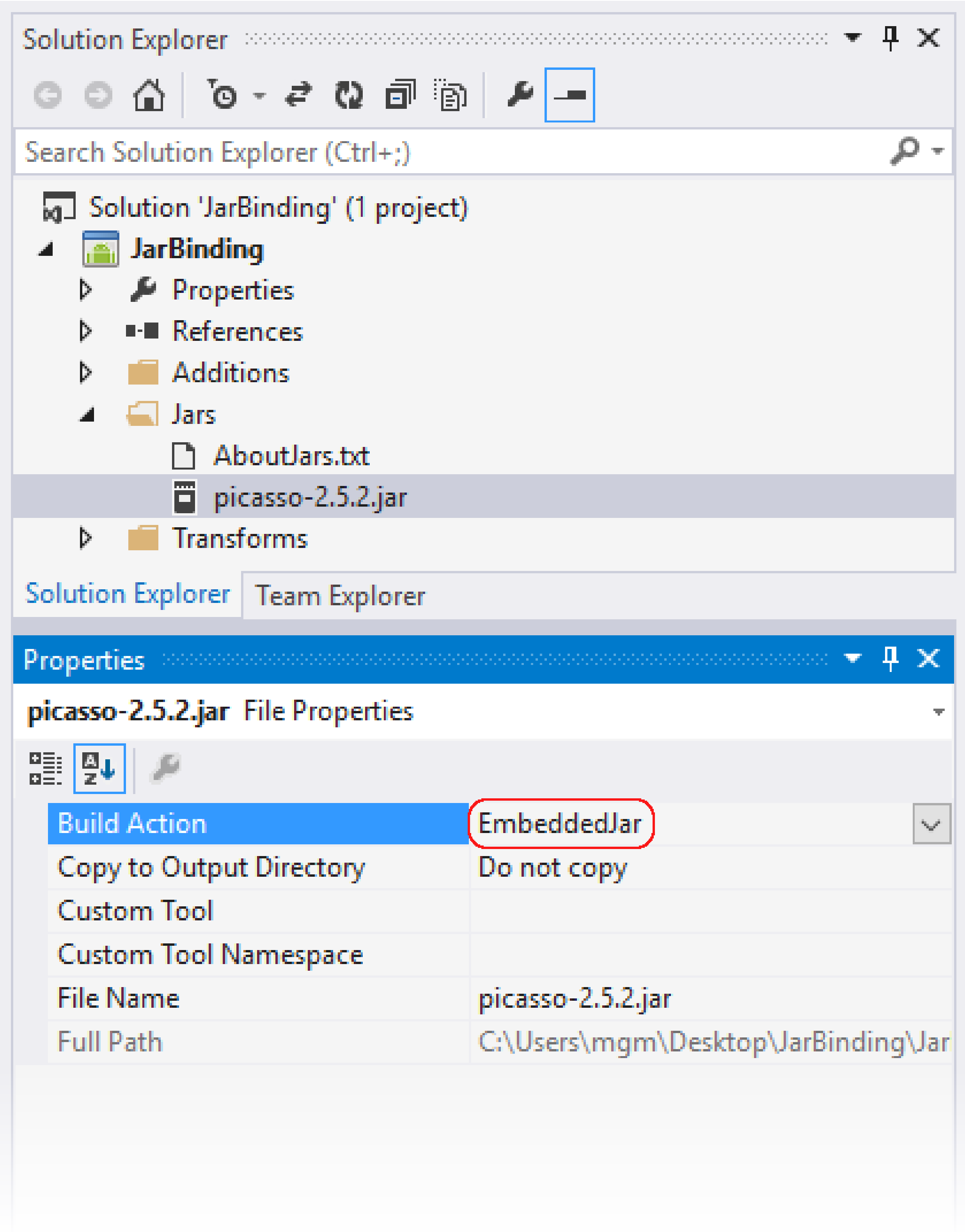


When you create a Java Bindings library project, you must specify whether the .JAR is to be embedded in the Bindings Library or packaged separately. To do that, you specify one of the following *build actions*:

- **EmbeddedJar** – the .JAR will be embedded in the Bindings Library.
- **InputJar** – the .JAR will be kept separate from the Bindings Library.

Typically, you use the **EmbeddedJar** build action so that the .JAR is automatically packaged into the bindings library. This is the simplest option – Java bytecode in the .JAR is converted into Dex bytecode and is embedded (along with the Managed Callable Wrappers) into your APK. If you want to keep the .JAR separate from the bindings library, you can use the **InputJar** option; however, you must ensure that the .JAR file is available on the device that runs your app.

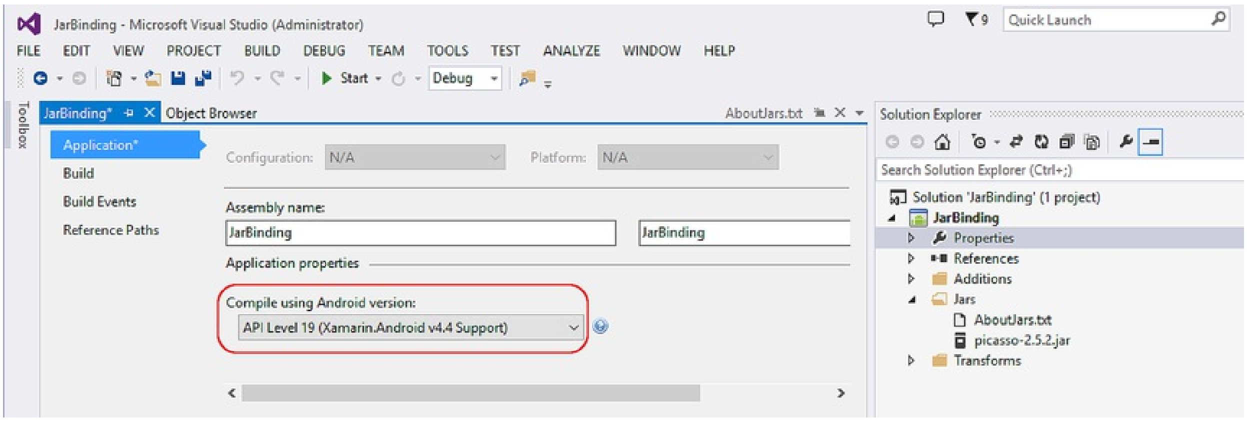
Set the build action to **EmbeddedJar**:



Next, open the project Properties to configure the *Target Framework*. If the .JAR uses any Android APIs, set the

Target Framework to the API level that the .JAR expects. Typically, the developer of the .JAR file will indicate which API level (or levels) that the .JAR is compatible with. (For more information about the Target Framework setting and Android API levels in general, see Understanding Android API Levels.)

Set the target API level for your Bindings Library (in this example, we are using API level 19):



Finally, build the Bindings Library. Although some warning messages may be displayed, the Bindings Library project should build successfully and produce an output .DLL at the following location: **JarBinding/bin/Debug/JarBinding.dll**

Using the Bindings Library

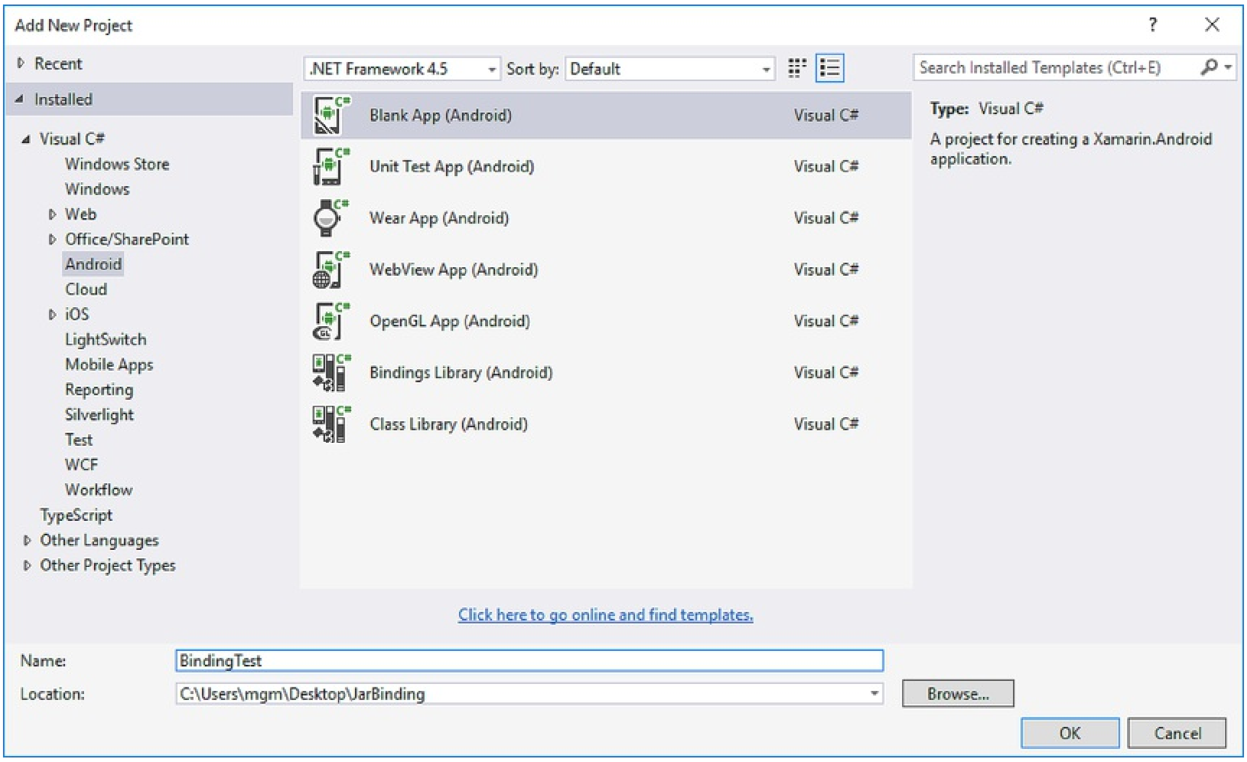
To consume this .DLL in your Xamarin.Android app, do the following:

1. Add a reference to the Bindings Library.

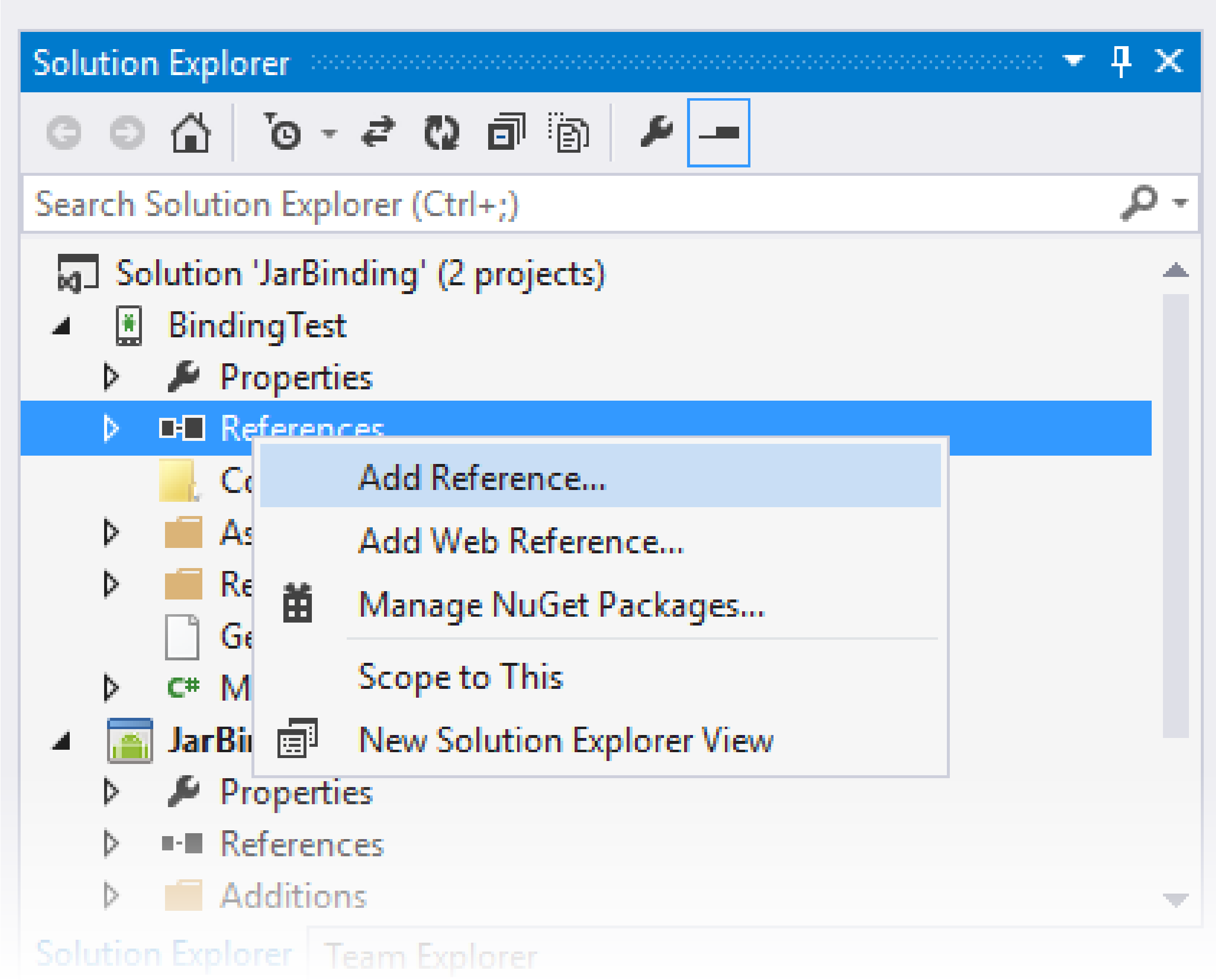
2. Make calls into the .JAR through the Managed Callable Wrappers.

In the following steps, we'll create a minimal app that uses the Bindings Library to download and display an image in an `ImageView`; the "heavy lifting" is done by the code that resides in the .JAR file.

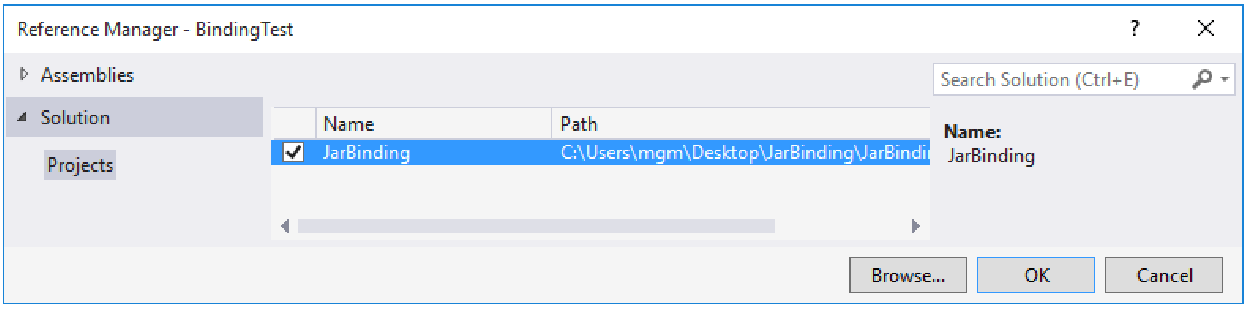
First, create a new Xamarin.Android app that consumes the Bindings Library. Right-click the Solution and select **Add New Project**; name the new project **BindingTest**. We're creating this app in the same Solution as the Bindings Library in order to simplify this walkthrough; however, the app that consumes the Bindings Library could, instead, reside in a different Solution:



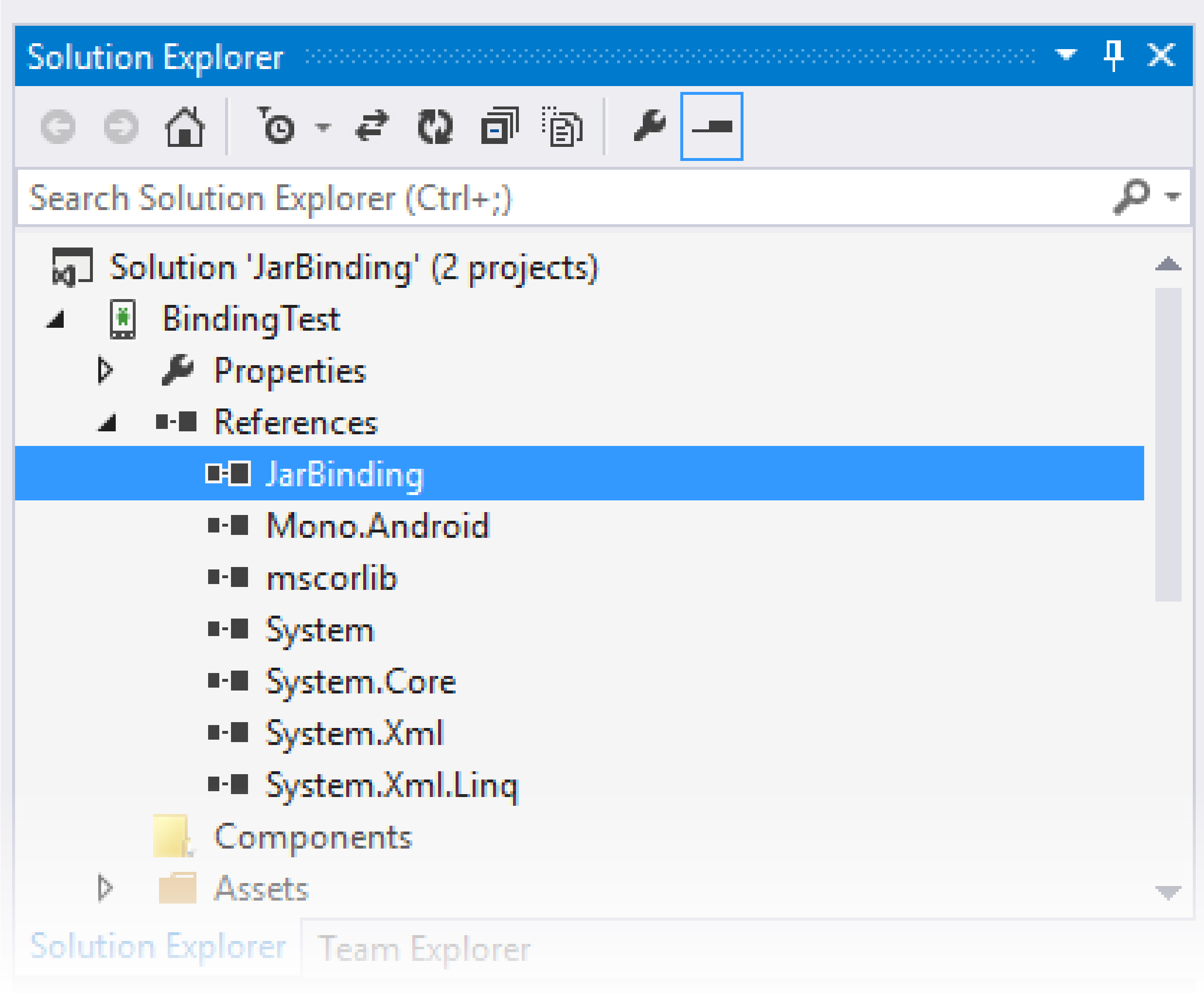
Right-click the **References** node of the **BindingTest** project and select **Add Reference...**:



Check the **JarBinding** project created earlier and click **OK**:



Open the **References** node of the **BindingTest** project and verify that the **JarBinding** reference is present:



Modify the **BindingTest** layout (**Main.axml**) so that it has a single `ImageView`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:minWidth="25px"
    android:minHeight="25px">
    <ImageView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/imageView" />
</LinearLayout>
```

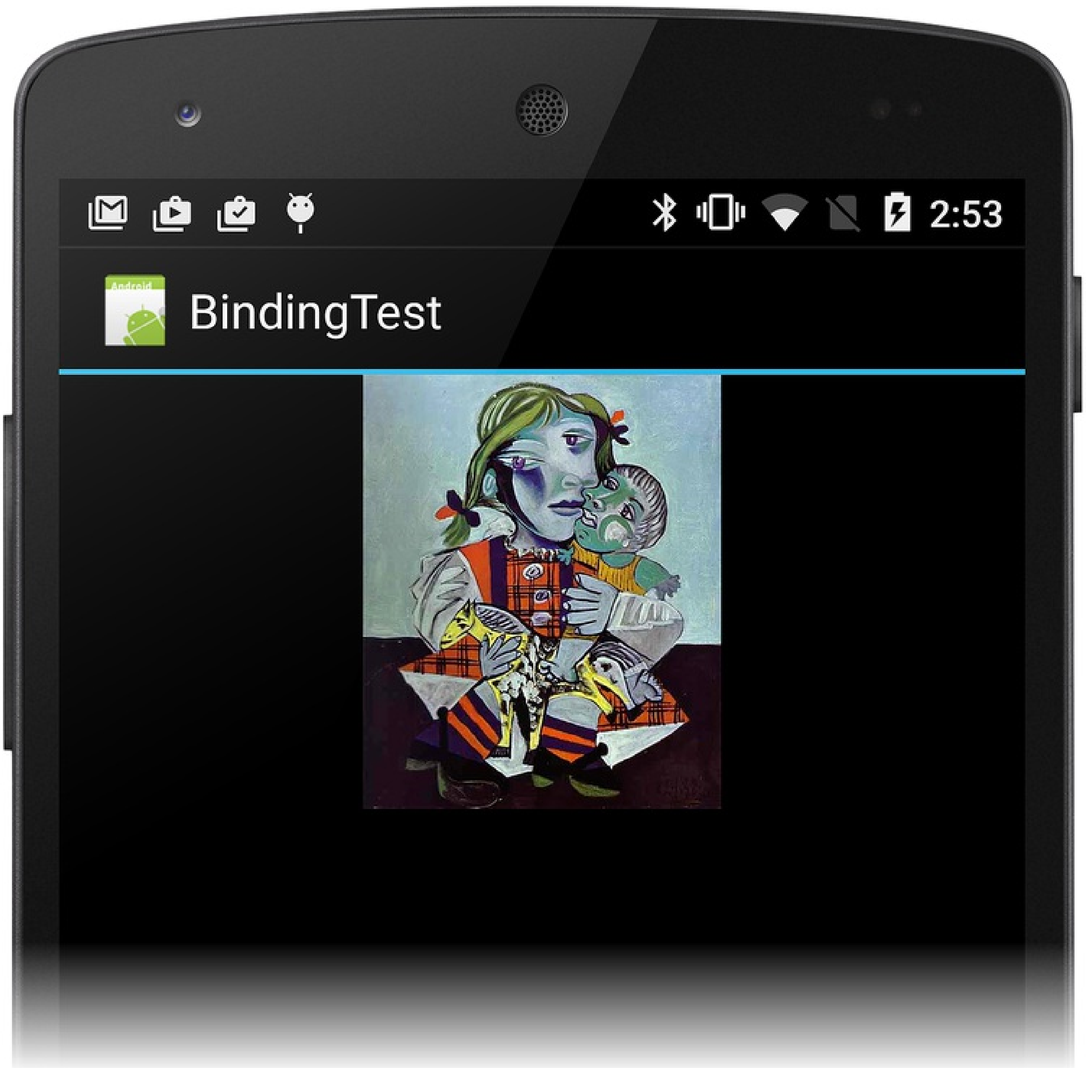
Add the following `using` statement to **MainActivity.cs** – this makes it possible to easily access the methods of the Java-based `Picasso` class that resides in the Bindings Library:

```
using Com.Squareup.Picasso;
```

Modify the `OnCreate` method so that it uses the `Picasso` class to load an image from a URL and display it in the `ImageView`:

```
public class MainActivity : Activity
{
    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);
        SetContentView(Resource.Layout.Main);
        ImageView imageView = FindViewById<ImageView>(Resource.Id.imageView);

        // Use the Picasso jar library to load and display this image:
        Picasso.With (this)
            .Load ("https://i.imgur.com/DvpvklR.jpg")
            .Into (imageView);
    }
}
```

Compile and run the **BindingTest** project. The app will startup, and after a short delay (depending on network conditions), it should download and display an image similar to the following screenshot:



Congratulations! You've successfully bound a Java library .JAR and used it in your Xamarin.Android app.

Summary

In this walkthrough, we created a Bindings Library for a third-party .JAR file, added the Bindings Library to a minimal test app, and then ran the app to verify that our C# code can call Java code residing in the .JAR file.

Related Links

- Building a Java Bindings Library (video)
- Binding a Java Library

Binding an .AAR

7/8/2021 • 8 minutes to read • Edit Online

IMPORTANT

We're currently investigating custom binding usage on the Xamarin platform. Please take **this survey** to inform future development efforts.

This walkthrough provides step-by-step instructions for creating a Xamarin.Android Java Bindings Library from an Android .AAR file.

Overview

The *Android Archive (.AAR)* file is the file format for Android libraries. An .AAR file is a .ZIP archive that contains the following:

- Compiled Java code
- Resource IDs
- Resources
- Meta-data (for example, Activity declarations, permissions)

In this guide, we'll step through the basics of creating a Bindings Library for a single .AAR file. For an overview of Java library binding in general (with a basic code example), see Binding a Java Library.

IMPORTANT

A binding project can only include one .AAR file. If the .AAR depends on other .AAR, then those dependencies should be contained in their own binding project and then referenced. See Bug 44573.

Walkthrough

We'll create a Bindings Library for an example Android archive file that was created in Android Studio, textanalyzer.aar. This .AAR contains a `TextCounter` class with static methods that count the number of vowels and consonants in a string. In addition, **textanalyzer.aar** contains an image resource to help display the counting results.

We'll use the following steps to create a Bindings Library from the .AAR file:

1. Create a new Java Bindings Library project.

2. Add a single .AAR file to the project. A binding project may only contain a single .AAR.

3. Set the appropriate build action for the .AAR file.

4. Choose a target framework that the .AAR supports.

5. Build the Bindings Library.

Once we've created the Bindings Library, we'll develop a small Android app that prompts the user for a text string, calls .AAR methods to analyze the text, retrieves the image from the .AAR, and displays the results along with the image.

The sample app will access the `TextCounter` class of **textanalyzer.aar**:

```
package com.xamarin.textcounter;

public class TextCounter
{
    ...
    public static int numVowels (String text) { ... };
    ...
    public static int numConsonants (String text) { ... };
    ...
}
```

In addition, this sample app will retrieve and display an image resource that is packaged in **textanalyzer.aar**:

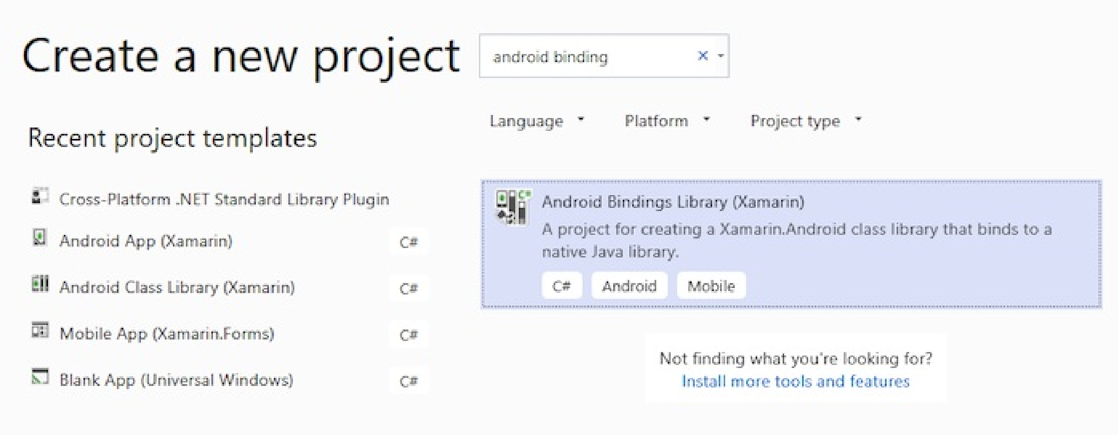
This image resource resides at **res/drawable/monkey.png** in **textanalyzer.aar**.

Creating the Bindings Library

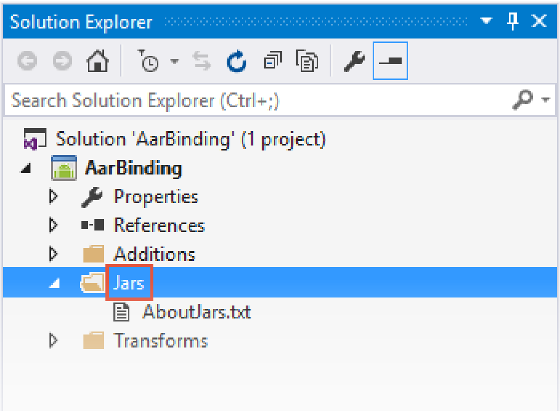
Before commencing with the steps below, please download the example textanalyzer.aar Android archive file:

1. Create a new Bindings Library project starting with the Android Bindings Library template. You can use

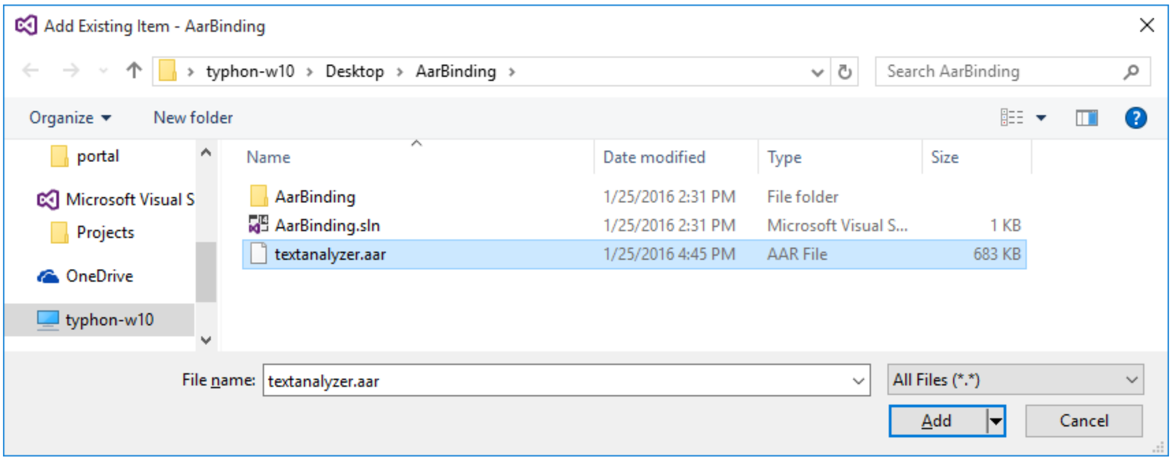
either Visual Studio for Mac or Visual Studio (the screenshots below show Visual Studio, but Visual Studio for Mac is very similar). Name the solution **AarBinding**:



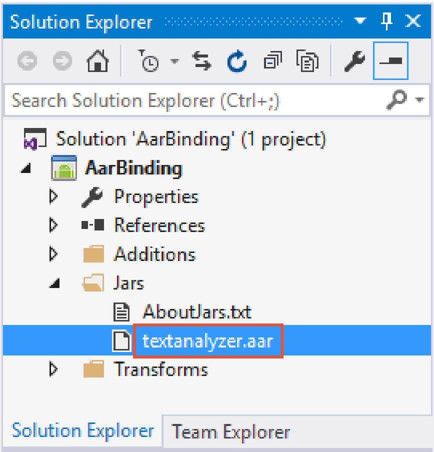
2. The template includes a **Jars** folder where you add your .AAR(s) to the Bindings Library project. Right-click the **Jars** folder and select **Add > Existing Item**:



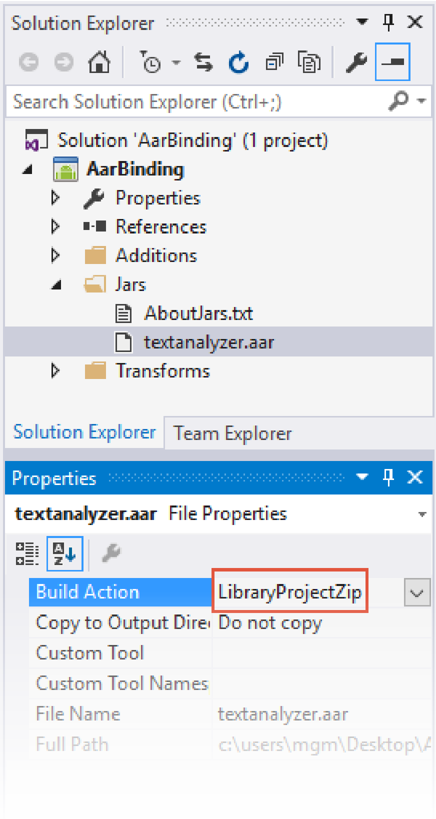
3. Navigate to the **textanalyzer.aar** file downloaded earlier, select it, and click **Add**:



4. Verify that the **textanalyzer.aar** file was successfully added to the project:

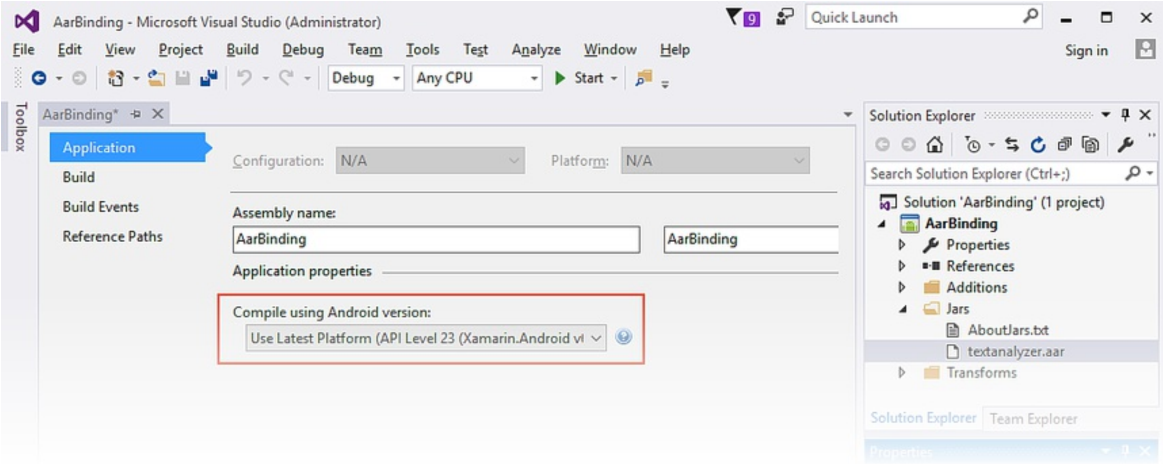


5. Set the Build Action for **textanalyzer.aar** to `LibraryProjectZip`. In Visual Studio for Mac, right-click **textanalyzer.aar** to set the Build Action. In Visual Studio, the Build Action can be set in the **Properties** pane):



6. Open the project Properties to configure the *Target Framework*. If the .AAR uses any Android APIs, set the Target Framework to the API level that the .AAR expects. (For more information about the Target Framework setting and Android API levels in general, see Understanding Android API Levels.)

   Set the target API level for your Bindings Library. In this example, we are free to use the latest platform API level (API level 23) because our **textanalyzer** does not have a dependency on Android APIs:

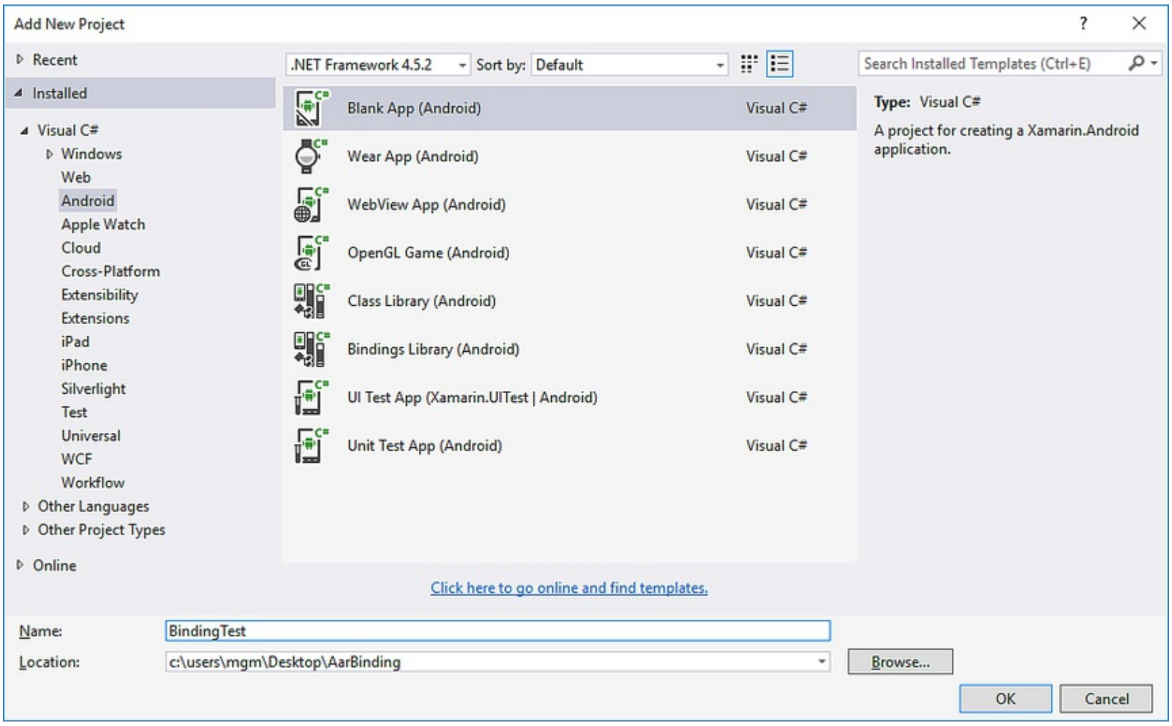


7. Build the Bindings Library. The Bindings Library project should build successfully and produce an output .DLL at the following location: **AarBinding/bin/Debug/AarBinding.dll**

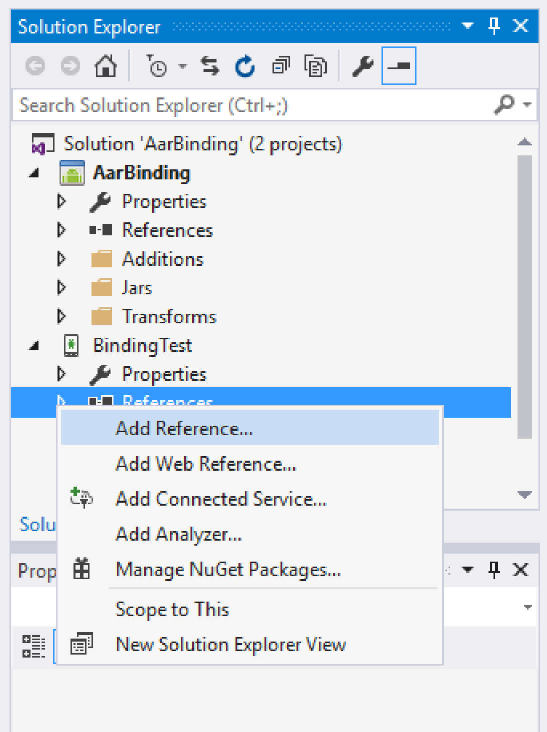
Using the Bindings Library

To consume this .DLL in your Xamarin.Android app, you must first add a reference to the Bindings Library. Use the following steps:

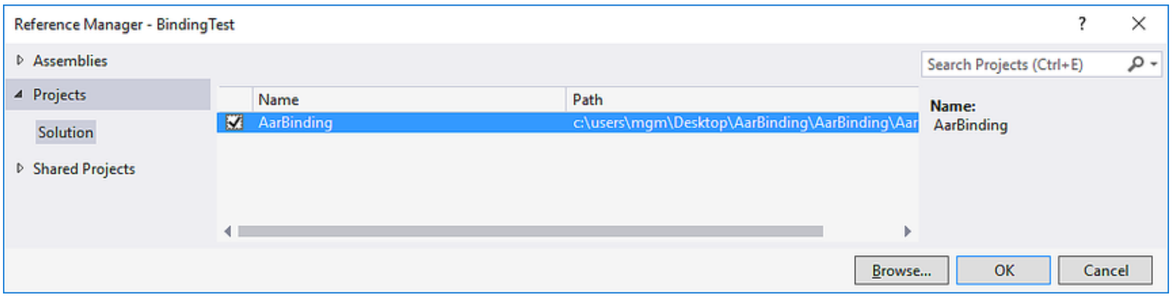
1. We're creating this app in the same Solution as the Bindings Library to simplify this walkthrough. (The app that consumes the Bindings Library could also reside in a different Solution.) Create a new Xamarin.Android app: right-click the Solution and select **Add New Project**. Name the new project **BindingTest**:



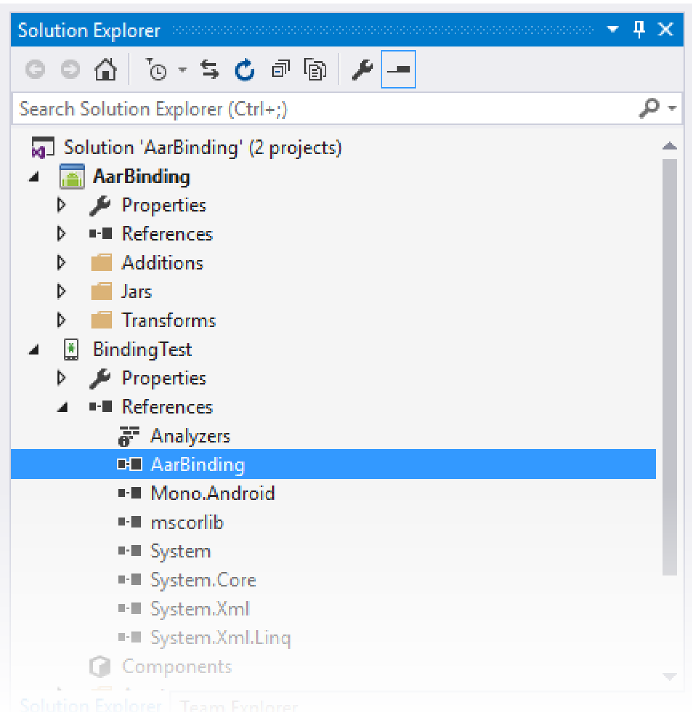
2. Right-click the **References** node of the **BindingTest** project and select **Add Reference...**:



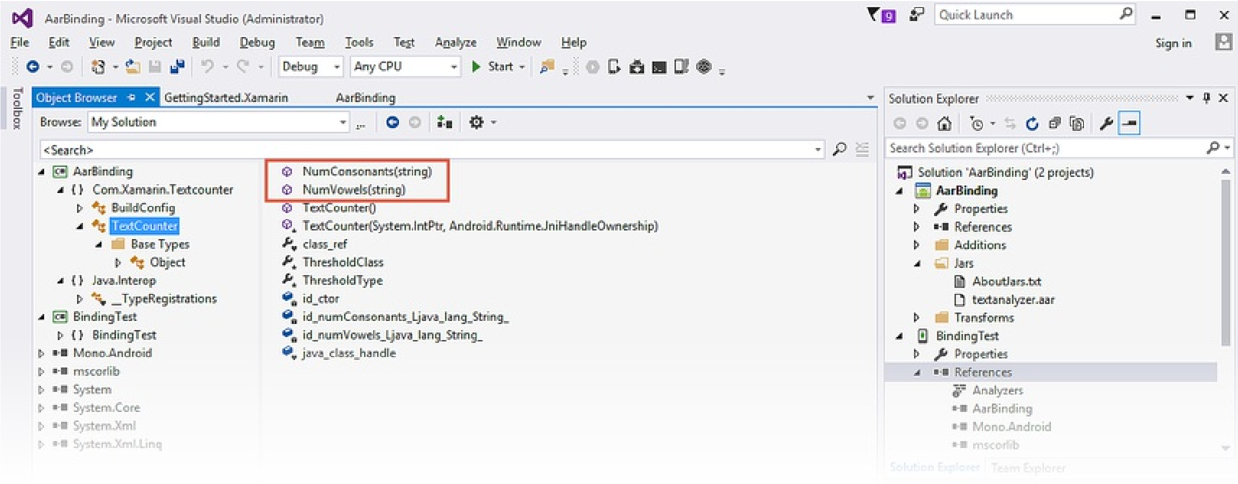
3. Select the **AarBinding** project created earlier and click **OK**:



4. Open the **References** node of the **BindingTest** project to verify that the **AarBinding** reference is present:



If you would like to view the contents of the Binding Library project, you can double-click the reference to open it in the **Object Browser**. You can see the mapped contents of the `Com.Xamarin.Textcounter` namespace (mapped from the Java `com.xamarin.textanalyzezr` package) and you can view the members of the `TextCounter` class:



The above screenshot highlights the two `TextAnalyzer` methods that the example app will call: `NumConsonants` (which wraps the underlying Java `numConsonants` method), and `NumVowels` (which wraps the underlying Java `numVowels` method).

Accessing .AAR Types

After you add a reference to your app that points to the Binding Library, you can access Java types in the .AAR as you would access C# types (thanks to the C# wrappers). C# app code can call `TextAnalyzer` methods as illustrated in this example:

```
using Com.Xamarin.Textcounter;
...
int numVowels = TextCounter.NumVowels (myText);
int numConsonants = TextCounter.NumConsonants (myText);
```

In the above example, we're calling static methods in the `TextCounter` class. However, you can also instantiate classes and call instance methods. For example, if your .AAR wraps a class called `Employee` that has the instance method `buildFullName`, you can instantiate `MyClass` and use it as seen here:

```
var employee = new Com.MyCompany.MyProject.Employee();
var name = employee.BuildFullName ();
```

The following steps add code to the app so that it prompts the user for text, uses `TextCounter` to analyze the text, and then displays the results.

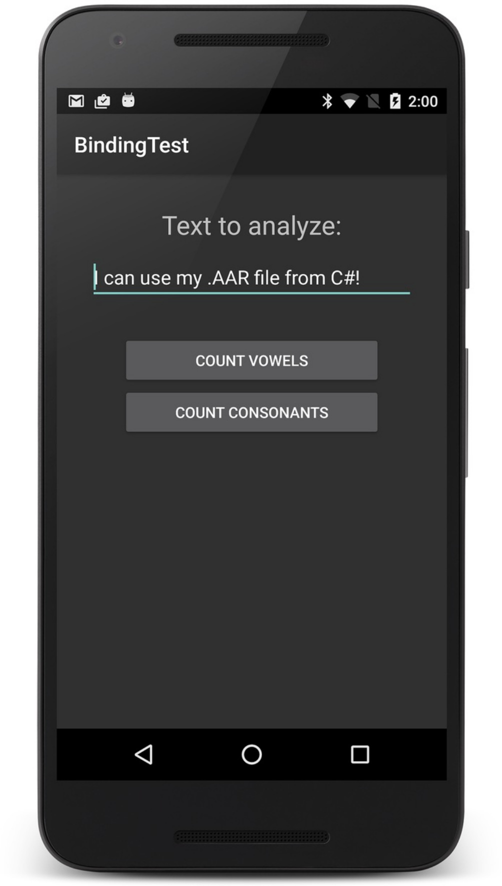
Replace the **BindingTest** layout (**Main.axml**) with the following XML. This layout has an `EditText` for text input and two buttons for initiating vowel and consonant counts:

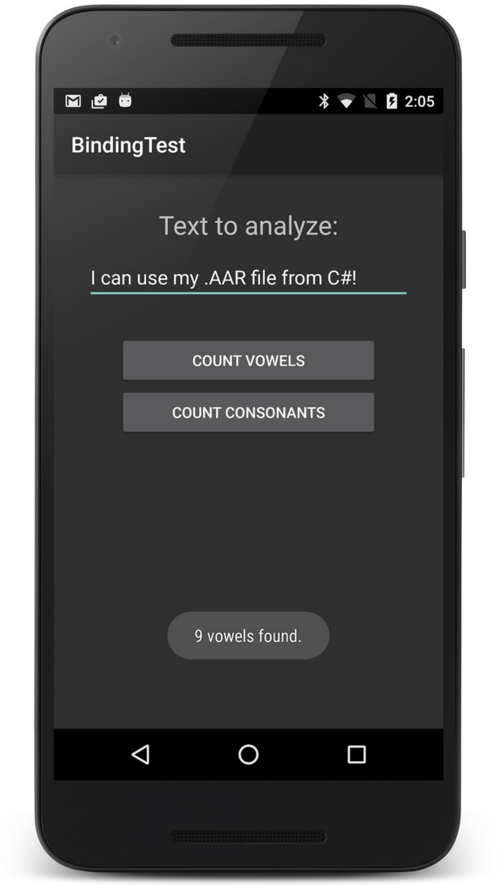
```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation             ="vertical"
    android:layout_width            ="fill_parent"
    android:layout_height           ="fill_parent" >
    <TextView
        android:text                ="Text to analyze:"
        android:textSize            ="24dp"
        android:layout_marginTop    ="30dp"
        android:layout_gravity      ="center"
        android:layout_width        ="wrap_content"
        android:layout_height       ="wrap_content" />
    <EditText
        android:id                  ="@+id/input"
        android:text                ="I can use my .AAR file from C#!"
        android:layout_marginTop    ="10dp"
        android:layout_gravity      ="center"
        android:layout_width        ="300dp"
        android:layout_height       ="wrap_content"/>
    <Button
        android:id                  ="@+id/vowels"
        android:layout_marginTop    ="30dp"
        android:layout_width        ="240dp"
        android:layout_height       ="wrap_content"
        android:layout_gravity      ="center"
        android:text                ="Count Vowels" />
    <Button
        android:id                  ="@+id/consonants"
        android:layout_width        ="240dp"
        android:layout_height       ="wrap_content"
        android:layout_gravity      ="center"
        android:text                ="Count Consonants" />
</LinearLayout>
```

Replace the contents of **MainActivity.cs** with the following code. As seen in this example, the button event handlers call wrapped `TextCounter` methods that reside in the .AAR and use toasts to display the results. Notice the `using` statement for the namespace of the bound library (in this case, `Com.Xamarin.Textcounter`):

```
using System;
using Android.App;
using Android.Content;
using Android.Runtime;
using Android.Views;
using Android.Widget;
using Android.OS;
using Android.Views.InputMethods;
using Com.Xamarin.Textcounter;

namespace BindingTest
{
    [Activity(Label = "BindingTest", MainLauncher = true, Icon = "@drawable/icon")]
    public class MainActivity : Activity
    {
        InputMethodManager imm;

        protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);

            SetContentView(Resource.Layout.Main);

            imm = (InputMethodManager)GetSystemService(Context.InputMethodService);

            var vowelsBtn = FindViewById<Button>(Resource.Id.vowels);
            var consonBtn = FindViewById<Button>(Resource.Id.consonants);
            var edittext = FindViewById<EditText>(Resource.Id.input);
            edittext.InputType = Android.Text.InputTypes.TextVariationPassword;

            edittext.KeyPress += (sender, e) =>
            {
                imm.HideSoftInputFromWindow(edittext.WindowToken, HideSoftInputFlags.NotAlways);
                e.Handled = true;
            };

            vowelsBtn.Click += (sender, e) =>
            {
                int count = TextCounter.NumVowels(edittext.Text);
                string msg = count + " vowels found.";
                Toast.MakeText (this, msg, ToastLength.Short).Show ();
            };

            consonBtn.Click += (sender, e) =>
            {
                int count = TextCounter.NumConsonants(edittext.Text);
                string msg = count + " consonants found.";
                Toast.MakeText (this, msg, ToastLength.Short).Show ();
            };

        }
    }
}
```

Compile and run the **BindingTest** project. The app will start and present the screenshot on the left (the `EditText` is initialized with some text, but you can tap it to change it). When you tap **COUNT VOWELS**, a toast displays the number of vowels as shown on the right:





Try tapping the **COUNT CONSONANTS** button. Also, you can modify the line of text and tap these buttons again to test for different vowel and consonant counts.

Accessing .AAR Resources

The Xamarin tooling merges the **R** data from the .AAR into your app's **Resource** class. As a result, you can access .AAR resources from your layout (and from code-behind) in the same way as you would access resources that are in the **Resources** path of your project.

To access an image resource, you use the **Resource.Drawable** name for the image packed inside the .AAR. For example, you can reference **image.png** in the .AAR file by using `@drawable/image`:

```
<ImageView android:src="@drawable/image" ... />
```

You can also access resource layouts that reside in the .AAR. To do this, you use the **Resource.Layout** name for the layout packaged inside the .AAR. For example:

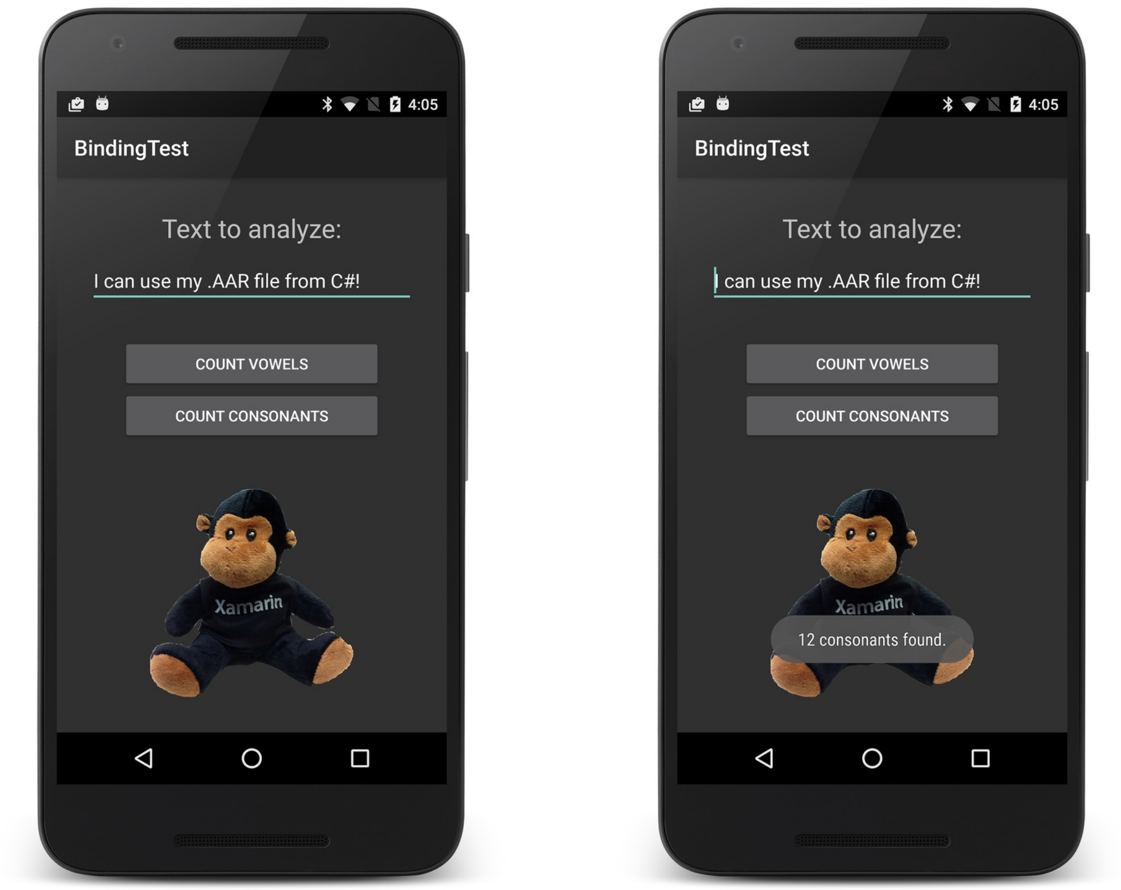
```
var a = new ArrayAdapter<string>(this, Resource.Layout.row_layout, ...);
```

The **textanalyzer.aar** example contains an image file that resides at **res/drawable/monkey.png**. Let's access this image resource and use it in our example app:

Edit the **BindingTest** layout (**Main.axml**) and add an `ImageView` to the end of the `LinearLayout` container. This `ImageView` displays the image found at **@drawable/monkey**; this image will be loaded from the resource section of **textanalyzer.aar**:

```
    ...
    <ImageView
        android:src                    ="@drawable/monkey"
        android:layout_marginTop       ="40dp"
        android:layout_width           ="200dp"
        android:layout_height          ="200dp"
        android:layout_gravity         ="center" />

</LinearLayout>
```

Compile and run the **BindingTest** project. The app will start and present the screenshot on the left – when you tap **COUNT CONSONANTS**, the results are displayed as shown on the right:



Congratulations! You've successfully bound a Java library .AAR!

Summary

In this walkthrough, we created a Bindings Library for an .AAR file, added the Bindings Library to a minimal test app, and ran the app to verify that our C# code can call Java code residing in the .AAR file. In addition, we extended the app to access and display an image resource that resides in the .AAR file.

Related Links

- Building a Java Bindings Library (video)
- Binding a .JAR
- Binding a Java Library

- AarBinding (sample)
- Bug 44573 - One project cannot bind multiple .aar files

Binding an Eclipse Library Project

7/8/2021 • 2 minutes to read • Edit Online

IMPORTANT

We're currently investigating custom binding usage on the Xamarin platform. Please take **this survey** to inform future development efforts.

This walkthrough explains how to use Xamarin.Android project templates to bind an Eclipse Android library project.

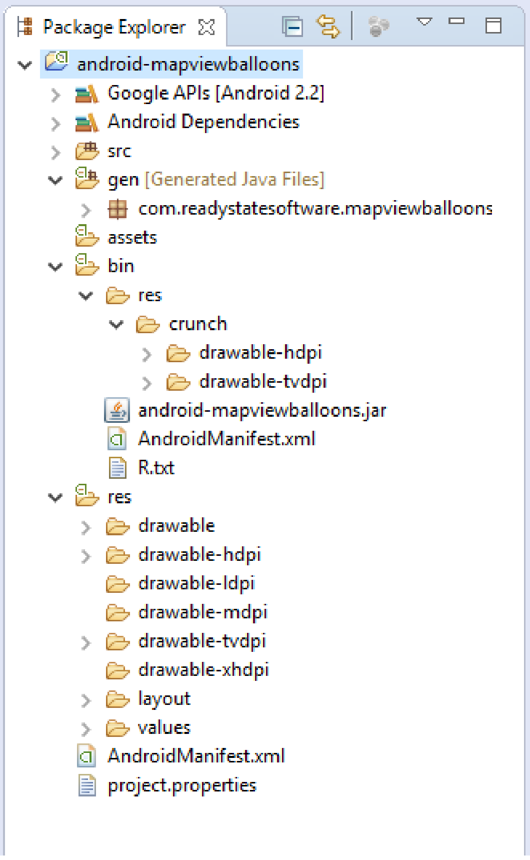
Overview

Although .AAR files are increasingly becoming the norm for Android library distribution, in some cases it is necessary to create a binding for an *Android library project*. Android library projects are special Android projects that contain shareable code and resources that can be referenced by Android application projects. Typically, you bind to an Android library project when the library is created in the Eclipse IDE. This walkthrough provides examples of how to create an Android library project .ZIP from the directory structure of an Eclipse project.

Android library projects are different from regular Android projects in that they are not compiled into an APK and are not, on their own, deployable to a device. Instead, an Android library project is meant to be referenced by an Android application project. When an Android application project is built, the Android library project is compiled first. The Android application project will then be absorbed into the compiled Android library project and include the code and resources into the APK for distribution. Because of this difference, creating a binding for an Android library project is slightly different than creating a binding for a Java .JAR or .AAR file.

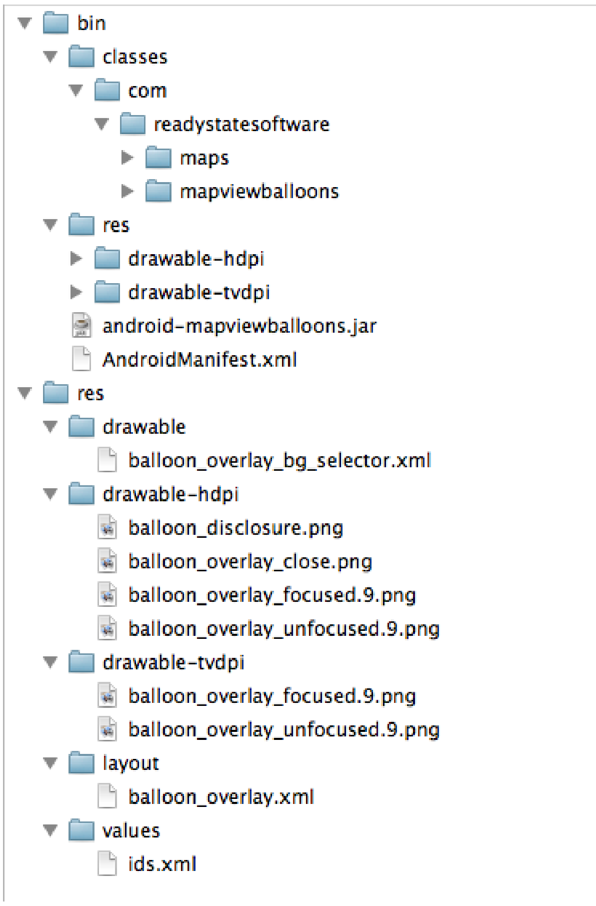
Walkthrough

To use an Android library project in a Xamarin.Android Java Binding project it is first necessary to build the Android library project in Eclipse. The following screenshot shows an example of one Android library project after compilation:

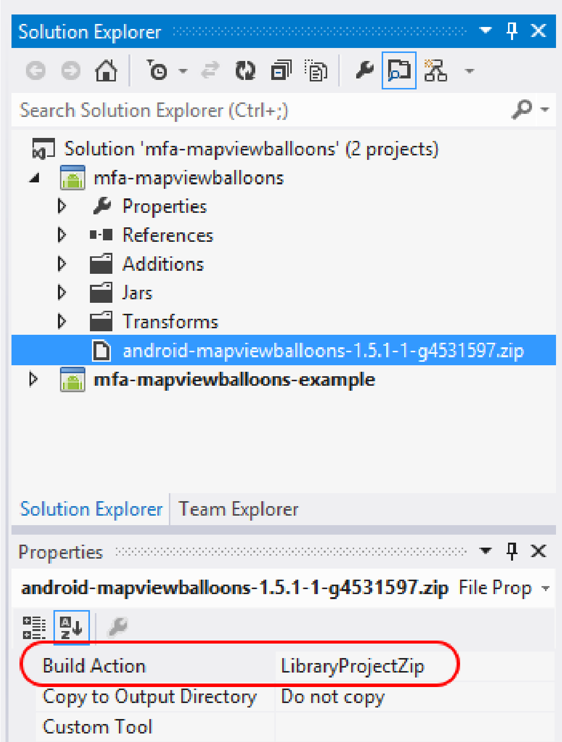


Notice that the source code from the Android library project has been compiled to a temporary .JAR file named **android-mapviewballoons.jar**, and that the resources have been copied to the **bin/res/crunch** folder.

Once the Android library project has been compiled in Eclipse, it can then be bound using a Xamarin.Android Java Binding project. First a .ZIP file must be created which contains the **bin** and **res** folders of the Android library project. It is important that you remove the intervening **crunch** subdirectory so that the resources reside in **bin/res**. The following screenshot shows the contents of one such .ZIP file:

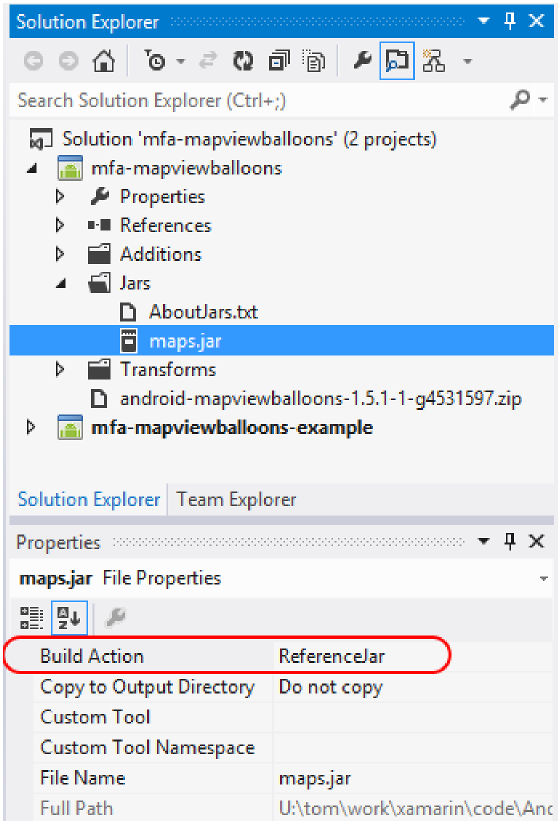


This .ZIP file is then added to Xamarin.Android Java Binding project, as shown in the following screenshot:



Notice that the Build Action of the .ZIP file has been automatically set to **LibraryProjectZip**.

If there are any .JAR files that are required by the Android library project, they should be added to the **Jars** folder of the Java Binding Library project and the **Build Action** set to **ReferenceJar**. An example of this can be seen in the screenshot below:



Once these steps are complete, the Xamarin.Android Java Binding project can be used as described earlier on in this document.

> **NOTE**
>
> Compiling the Android library projects in other IDEs is not supported at this time. Other IDEs may not create the same directory structure or files in the **bin** folder as Eclipse.

Summary

In this article, we walked through the process of binding an Android library project. We built the Android library project in Eclipse, then we created a zip file from the **bin** and **res** folders of the Android library project. Next, we used this zip to create a Xamarin.Android Java Binding project.

Customizing Bindings

10/28/2019 • 2 minutes to read • Edit Online

You can customize an Xamarin.Android binding by editing the metadata that controls the binding process. These manual modifications are often necessary for resolving build errors and for shaping the resulting API so that it is more consistent with C#/.NET. These guides explain the structure of this metadata, how to modify the metadata, and how to use JavaDoc to recover the names of method parameters.

Overview

Xamarin.Android automates much of the binding process; however, in some cases manual modification is required to address the following issues:

- Resolving build errors caused by missing types, obfuscated types, duplicate names, class visibility issues, and other situations that cannot be resolved by the Xamarin.Android tooling.
- Changing the mapping that Xamarin.Android uses to bind the Android API to different types in C# (for example, many developers prefer to map Java `int` constants to C# `enum` constants).
- Removing unused types that do not need to be bound.
- Adding types that have no counterpart in the underlying Java API.

You can make some or all of these changes by modifying the metadata that controls the binding process.

Guides

The following guides describe the metadata that controls the binding process and explain how to modify this metadata to address these issues:

- Java Bindings Metadata provides an overview of the metadata that goes into a Java binding. It describes the various manual steps that are sometimes required to complete a Java binding library, and it explains how to shape an API exposed by a binding to more closely follow .NET design guidelines.
- Naming Parameters with Javadoc explains how to recover parameter names in a Java Binding Project by using Javadoc produced from the bound Java project.

Java Bindings Metadata

7/8/2021 • 10 minutes to read • Edit Online

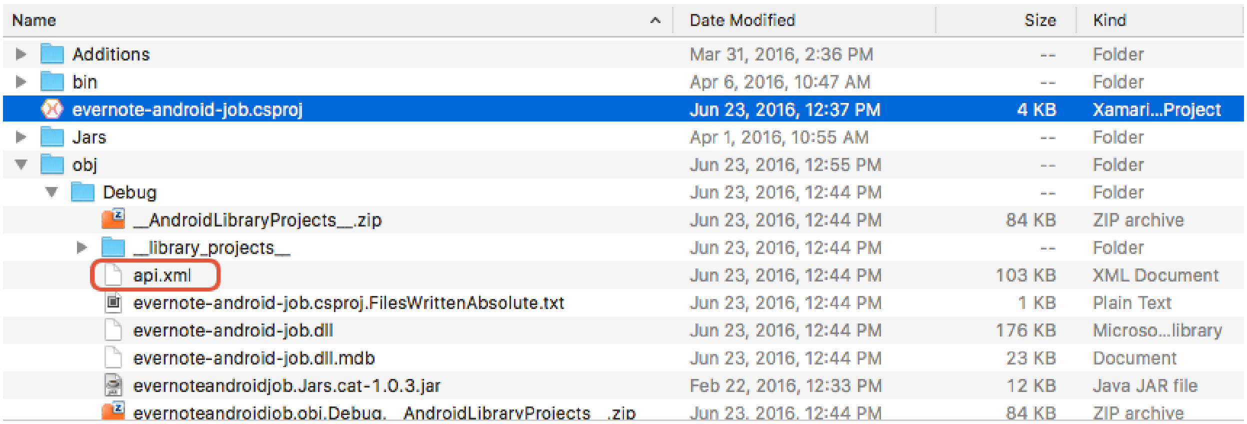
IMPORTANT

We're currently investigating custom binding usage on the Xamarin platform. Please take **this survey** to inform future development efforts.

C# code in Xamarin.Android calls Java libraries through bindings, which are a mechanism that abstracts the low-level details that are specified in Java Native Interface (JNI). Xamarin.Android provides a tool that generates these bindings. This tooling lets the developer control how a binding is created by using metadata, which allows procedures such as modifying namespaces and renaming members. This document discusses how metadata works, summarizes the attributes that metadata supports, and explains how to resolve binding problems by modifying this metadata.

Overview

A Xamarin.Android **Java Binding Library** tries to automate much of the work necessary for binding an existing Android library with the help of a tool sometimes known as the *Bindings Generator*. When binding a Java library, Xamarin.Android will inspect the Java classes and generate a list of all the packages, types, and members which to be bound. This list of APIs is stored in an XML file that can be found at **{project directory}\obj\Release\api.xml** for a **RELEASE** build and at **{project directory}\obj\Debug\api.xml** for a **DEBUG** build.



The Bindings Generator will use the **api.xml** file as a guideline for generating the necessary C# wrapper classes. The contents of this XML file are a variation of Google's *Android Open Source Project* format. The following snippet is an example of the contents of **api.xml**:

```
<api>
    <package name="android">
        <class abstract="false" deprecated="not deprecated" extends="java.lang.Object"
            extends-generic-aware="java.lang.Object"
            final="true"
            name="Manifest"
            static="false"
            visibility="public">
            <constructor deprecated="not deprecated" final="false"
                name="Manifest" static="false" type="android.Manifest"
                visibility="public">
            </constructor>
        </class>
...
</api>
```

In this example, **api.xml** declares a class in the `android` package named `Manifest` that extends the `java.lang.Object`.

In many cases, human assistance is required to make the Java API feel more ".NET like" or to correct issues that prevent the binding assembly from compiling. For example, it may be necessary to change Java package names to .NET namespaces, rename a class, or change the return type of a method.

These changes are not achieved by modifying **api.xml** directly. Instead, changes are recorded in special XML files that are provided by the Java Binding Library template. When compiling the Xamarin.Android binding assembly, the Bindings Generator will be influenced by these mapping files when creating the binding assembly

These XML mapping files may be found in the **Transforms** folder of the project:

- **MetaData.xml** – Allows changes to be made to the final API, such as changing the namespace of the generated binding.
- **EnumFields.xml** – Contains the mapping between Java `int` constants and C# `enums` .
- **EnumMethods.xml** – Allows changing method parameters and return types from Java `int` constants to C# `enums` .

The **MetaData.xml** file is the most import of these files as it allows general-purpose changes to the binding such as:

- Renaming namespaces, classes, methods, or fields so they follow .NET conventions.
- Removing namespaces, classes, methods, or fields that aren't needed.
- Moving classes to different namespaces.
- Adding additional support classes to make the design of the binding follow .NET framework patterns.

Lets move on to discuss **Metadata.xml** in more detail.

Metadata.xml Transform File

As we've already learned, the file **Metadata.xml** is used by the Bindings Generator to influence the creation of the binding assembly. The metadata format uses XPath syntax and is nearly identical to the *GAPI Metadata* described in GAPI Metadata guide. This implementation is almost a complete implementation of XPath 1.0 and thus supports items in the 1.0 standard. This file is a powerful XPath based mechanism to change, add, hide, or move any element or attribute in the API file. All of the rule elements in the metadata spec include a path attribute to identify the node to which the rule is to be applied. The rules are applied in the following order:

- **add-node** – Appends a child node to the node specified by the path attribute.

- **attr** – Sets the value of an attribute of the element specified by the path attribute.
- **remove-node** – Removes nodes matching a specified XPath.

The following is an example of a **Metadata.xml** file:

```
<metadata>
    <!-- Normalize the namespace for .NET -->
    <attr path="/api/package[@name='com.evernote.android.job']"
        name="managedName">Evernote.AndroidJob</attr>

    <!-- Don't  need these packages for the Xamarin binding/public API -->
    <remove-node path="/api/package[@name='com.evernote.android.job.v14']" />
    <remove-node path="/api/package[@name='com.evernote.android.job.v21']" />

    <!-- Change a parameter name from the generic p0 to a more meaningful one. -->
    <attr
path="/api/package[@name='com.evernote.android.job']/class[@name='JobManager']/method[@name='forceApi']/para
meter[@name='p0']"
        name="name">api</attr>
</metadata>
```

The following lists some of the more commonly used XPath elements for the Java API's:

- `interface` – Used to locate a Java interface. e.g. `/interface[@name='AuthListener']`.
- `class` – Used to locate a class . e.g. `/class[@name='MapView']`.
- `method` – Used to locate a method on a Java class or interface. e.g. `/class[@name='MapView']/method[@name='setTitleSource']`.
- `parameter` – Identify a parameter for a method. e.g. `/parameter[@name='p0']`

Adding Types

The `add-node` element will tell the Xamarin.Android binding project to add a new wrapper class to **api.xml**. For example, the following snippet will direct the Binding Generator to create a class with a constructor and a single field:

```
<add-node path="/api/package[@name='org.alljoyn.bus']">
    <class abstract="false" deprecated="not deprecated" final="false" name="AuthListener.AuthRequest"
static="true" visibility="public" extends="java.lang.Object">
        <constructor deprecated="not deprecated" final="false" name="AuthListener.AuthRequest"
static="false" type="org.alljoyn.bus.AuthListener.AuthRequest" visibility="public" />
        <field name="p0" type="org.alljoyn.bus.AuthListener.Credentials" />
    </class>
</add-node>
```

Removing Types

It is possible to instruct the Xamarin.Android Bindings Generator to ignore a Java type and not bind it. This is done by adding a `remove-node` XML element to the **metadata.xml** file:

```
<remove-node path="/api/package[@name='{package_name}']/class[@name='{name}']" />
```

Renaming Members

Renaming members cannot be done by directly editing the **api.xml** file because Xamarin.Android requires the original Java Native Interface (JNI) names. Therefore, the `//class/@name` attribute cannot be altered; if it is, the binding will not work.

Consider the case where we want to rename a type, `android.Manifest`. To accomplish this, we might try to directly edit **api.xml** and rename the class like so:

```
<attr path="/api/package[@name='android']/class[@name='Manifest']"
    name="name">NewName</attr>
```

This will result in the Bindings Generator creating the following C# code for the wrapper class:

```
[Register ("android/NewName")]
public class NewName : Java.Lang.Object { ... }
```

Notice that the wrapper class has been renamed to `NewName`, while the original Java type is still `Manifest`. It is no longer possible for the Xamarin.Android binding class to access any methods on `android.Manifest`; the wrapper class is bound to a non-existent Java type.

To properly change the managed name of a wrapped type (or method), it is necessary to set the `managedName` attribute as shown in this example:

```
<attr path="/api/package[@name='android']/class[@name='Manifest']"
    name="managedName">NewName</attr>
```

Renaming `EventArg` Wrapper Classes

When the Xamarin.Android binding generator identifies an `onXXX` setter method for a *listener type*, a C# event and `EventArgs` subclass will be generated to support a .NET flavoured API for the Java-based listener pattern. As an example, consider the following Java class and method:

```
com.someapp.android.mpa.guidance.NavigationManager.on2DSignNextManuever(NextManueverListener listener);
```

Xamarin.Android will drop the prefix `on` from the setter method and instead use `2DSignNextManuever` as the basis for the name of the `EventArgs` subclass. The subclass will be named something similar to:

```
NavigationManager.2DSignNextManueverEventArgs
```

This is not a legal C# class name. To correct this problem, the binding author must use the `argsType` attribute and provide a valid C# name for the `EventArgs` subclass:

```
<attr path="/api/package[@name='com.someapp.android.mpa.guidance']/
    interface[@name='NavigationManager.Listener']/
    method[@name='on2DSignNextManeuver']"
    name="argsType">NavigationManager.TwoDSignNextManueverEventArgs</attr>
```

Supported Attributes

The following sections describe some of the attributes for transforming Java APIs.

argsType

This attribute is placed on setter methods to name the `EventArg` subclass that will be generated to support Java listeners. This is described in more detail below in the section Renaming EventArg Wrapper Classes later on in this guide.

eventName

Specifies a name for an event. If empty, it inhibits event generation. This is described in more detail in the section title Renaming EventArg Wrapper Classes.

managedName

This is used to change the name of a package, class, method, or parameter. For example to change the name of the Java class `MyClass` to `NewClassName`:

```
<attr path="/api/package[@name='com.my.application']/class[@name='MyClass']"
    name="managedName">NewClassName</attr>
```

The next example illustrates an XPath expression for renaming the method `java.lang.object.toString` to `Java.Lang.Object.NewManagedName`:

```
<attr path="/api/package[@name='java.lang']/class[@name='Object']/method[@name='toString']"
    name="managedName">NewMethodName</attr>
```

managedType

`managedType` is used to change the return type of a method. In some situations the Bindings Generator will incorrectly infer the return type of a Java method, which will result in a compile time error. One possible solution in this situation is to change the return type of the method.

For example, the Bindings Generator believes that the Java method `de.neom.neoreadersdk.resolution.compareTo()` should return an `int` and take `Object` as parameters, which results in the error message **Error CS0535: 'DE.Neom.Neoreadersdk.Resolution' does not implement interface member 'Java.Lang.IComparable.CompareTo(Java.Lang.Object)'**. The following snippet demonstrates how to change the first parameter's type of the generated C# method from a `DE.Neom.Neoreadersdk.Resolution` to a `Java.Lang.Object`:

```
<attr path="/api/package[@name='de.neom.neoreadersdk']/
    class[@name='Resolution']/
    method[@name='compareTo' and count(parameter)=1 and
    parameter[1][@type='de.neom.neoreadersdk.Resolution']]/
    parameter[1]" name="managedType">Java.Lang.Object</attr>
```

managedReturn

Changes the return type of a method. This does not change the return attribute (as changes to return attributes can result in incompatible changes to the JNI signature). In the following example, the return type of the `append` method is changed from `SpannableStringBuilder` to `IAppendable` (recall that C# does not support covariant return types):

```
<attr path="/api/package[@name='android.text']/
    class[@name='SpannableStringBuilder']/
    method[@name='append']"
    name="managedReturn">Java.Lang.IAppendable</attr>
```

obfuscated

Tools that obfuscate Java libraries may interfere with the Xamarin.Android Binding Generator and its ability to generate C# wrapper classes. Characteristics of obfuscated classes include:

- The class name includes a **$**, i.e. **a$.class**
- The class name is entirely compromised of lower case characters, i.e. **a.class**

This snippet is an example of how to generate an "un-obfuscated" C# type:

```
<attr path="/api/package[@name='{package_name}']/class[@name='{name}']"
    name="obfuscated">false</attr>
```

propertyName

This attribute can be used to change the name of a managed property.

A specialized case of using `propertyName` involves the situation where a Java class has only a getter method for a field. In this situation the Binding Generator would want to create a write-only property, something that is discouraged in .NET. The following snippet shows how to "remove" the .NET properties by setting the `propertyName` to an empty string:

```
<attr
path="/api/package[@name='org.java_websocket.handshake']/class[@name='HandshakeImpl1Client']/method[@name='s
etResourceDescriptor'
    and count(parameter)=1
    and parameter[1][@type='java.lang.String']]"
    name="propertyName"></attr>
<attr
path="/api/package[@name='org.java_websocket.handshake']/class[@name='HandshakeImpl1Client']/method[@name='g
etResourceDescriptor'
    and count(parameter)=0]"
    name="propertyName"></attr>
```

Note that the setter and getter methods will still be created by the Bindings Generator.

sender

Specifies which parameter of a method should be the `sender` parameter when the method is mapped to an event. The value can be `true` or `false`. For example:

```
<attr path="/api/package[@name='android.app']/
    interface[@name='TimePickerDialog.OnTimeSetListener']/
    method[@name='onTimeSet']/
    parameter[@name='view']"
    name="sender">true</ attr>
```

visibility

This attribute is used to change the visibility of a class, method, or property. For example, it may be necessary to promote a `protected` Java method so that it's corresponding C# wrapper is `public`:

```
<!-- Change the visibility of a class -->
<attr path="/api/package[@name='namespace']/class[@name='ClassName']" name="visibility">public</attr>

<!-- Change the visibility of a method -->
<attr path="/api/package[@name='namespace']/class[@name='ClassName']/method[@name='MethodName']"
name="visibility">public</attr>
```

EnumFields.xml and EnumMethods.xml

There are cases where Android libraries use integer constants to represent states that are passed to properties or methods of the libraries. In many cases, it is useful to bind these integer constants to enums in C#. To facilitate this mapping, use the **EnumFields.xml** and **EnumMethods.xml** files in your binding project.

Defining an Enum using EnumFields.xml

The **EnumFields.xml** file contains the mapping between Java `int` constants and C# `enums`. Let's take the following example of a C# enum being created for a set of `int` constants:

```
<mapping jni-class="com/skobbler/ngx/map/realreach/SKRealReachSettings" clr-enum-
type="Skobbler.Ngx.Map.RealReach.SKMeasurementUnit">
    <field jni-name="UNIT_SECOND" clr-name="Second" value="0" />
    <field jni-name="UNIT_METER" clr-name="Meter" value="1" />
    <field jni-name="UNIT_MILIWATT_HOURS" clr-name="MilliwattHour" value="2" />
</mapping>
```

Here we have taken the Java class `SKRealReachSettings` and defined a C# enum called `SKMeasurementUnit` in the namespace `Skobbler.Ngx.Map.RealReach`. The `field` entries defines the name of the Java constant (example `UNIT_SECOND`), the name of the enum entry (example `Second`), and the integer value represented by both entities (example `0`).

Defining Getter/Setter Methods using EnumMethods.xml

The **EnumMethods.xml** file allows changing method parameters and return types from Java `int` constants to C# `enums`. In other words, it maps the reading and writing of C# enums (defined in the **EnumFields.xml** file) to Java `int` constant `get` and `set` methods.

Given the `SKRealReachSettings` enum defined above, the following **EnumMethods.xml** file would define the getter/setter for this enum:

```
<mapping jni-class="com/skobbler/ngx/map/realreach/SKRealReachSettings">
    <method jni-name="getMeasurementUnit" parameter="return" clr-enum-
type="Skobbler.Ngx.Map.RealReach.SKMeasurementUnit" />
    <method jni-name="setMeasurementUnit" parameter="measurementUnit" clr-enum-
type="Skobbler.Ngx.Map.RealReach.SKMeasurementUnit" />
</mapping>
```

The first `method` line maps the return value of the Java `getMeasurementUnit` method to the `SKMeasurementUnit` enum. The second `method` line maps the first parameter of the `setMeasurementUnit` to the same enum.

With all of these changes in place, you can use the follow code in Xamarin.Android to set the `MeasurementUnit`:

```
realReachSettings.MeasurementUnit = SKMeasurementUnit.Second;
```

Summary

This article discussed how Xamarin.Android uses metadata to transform an API definition from the *Google AOSP format*. After covering the changes that are possible using *Metadata.xml*, it examined the limitations encountered when renaming members and it presented the list of supported XML attributes, describing when each attribute should be used.

Related Links

- Working with JNI
- Binding a Java Library
- GAPI Metadata

Naming Parameters With Javadoc

11/2/2020 • 2 minutes to read • Edit Online

> **IMPORTANT**
>
> We're currently investigating custom binding usage on the Xamarin platform. Please take **this survey** to inform future development efforts.

This article explains how to recover parameter names in an Java Binding Project by using Javadoc generated from the Java project.

Overview

When binding an existing Java library, some metadata about the bound API is lost. In particular the names of parameters to methods. Parameter names will appear as `p0`, `p1`, etc. This is because the Java `.class` files do not preserve the parameter names that were used in the Java source code.

A Xamarin.Android Java binding project can provide the parameter names if it has access to the Javadoc HTML from the original library.

Integrating Javadoc HTML into a Java Binding Project

Integrating the Javadoc HTML into a Java Binding project is a manual process consisting of the following steps:

1. Download the Javadoc for the library
2. Edit the `.csproj` file and add a `<JavaDocPaths>` property:
3. Clean and rebuild the project

Once this is done, the original Java parameter names should be present in the APIs bound by a Java Binding Project.

> **NOTE**
>
> There is a great deal of variance in the JavaDoc output. The .JAR binding toolchain does not support every single possible permutation and consequently some parameter may not be properly named.

Summary

This article covered how use Javadoc in a Java Binding Project to provide meaning parameter names for bound APIs.

Troubleshooting Bindings

7/8/2021 • 9 minutes to read • Edit Online

IMPORTANT

We're currently investigating custom binding usage on the Xamarin platform. Please take **this survey** to inform future development efforts.

This article summarizes serveral common errors that may occur when generating bindings, along with possible causes and suggested ways to resolve them.

Overview

Binding an Android library (an **.aar** or a **.jar**) file is seldom a straightforward affair; it usually requires additional effort to mitigate issues that result from the differences between Java and .NET. These issues will prevent Xamarin.Android from binding the Android library and present themselves as error messages in the build log. This guide will provide some tips for troubleshooting the issues, list some of the more common problems/scenarios, and provide possible solutions to successfully binding the Android library.

When binding an existing Android library, it is necessary to keep in mind the following points:

- **The external dependencies for the library** – Any Java dependencies required by the Android library must be included in the Xamarin.Android project as a **ReferenceJar** or as an **EmbeddedReferenceJar**.
- **The Android API level that the Android library is targetting** – It is not possible to "downgrade" the Android API level; ensure that the Xamarin.Android binding project is targeting the same API level (or higher) as the Android library.
- **The version of the Android JDK that was used to package the Android library** – Binding errors may occur if the Android library was built with a different version of JDK than the one in use by Xamarin.Android. If possible, recompile the Android library using the same version of the JDK that is used by your installation of Xamarin.Android.

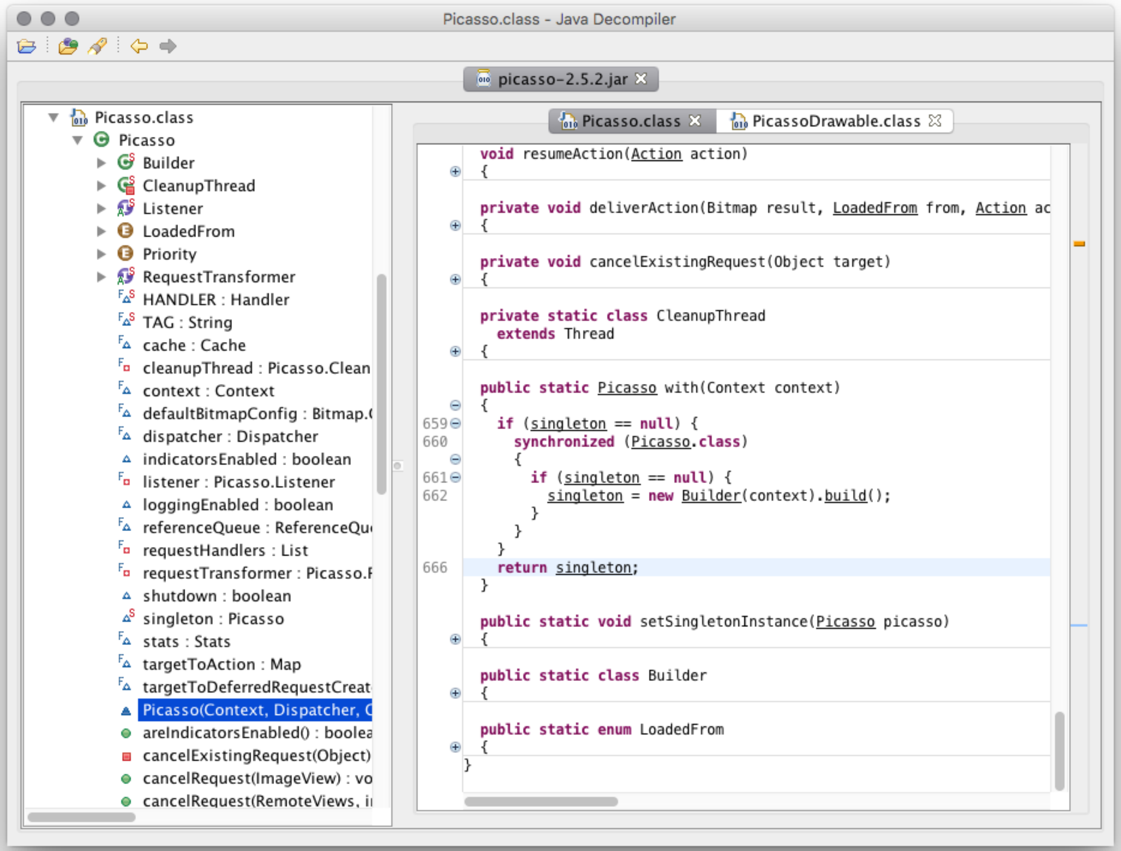
The first step to troubleshooting issues with binding a Xamarin.Android library is to enable diagnostic MSBuild output. After enabling the diagnostic output, rebuild the Xamarin.Android binding project and examine the build log to locate clues about what the cause of problem is.

It can also prove helpful to decompile the Android library and examine the types and methods that Xamarin.Android is trying to bind. This is covered in more detail later on in this guide.

Decompiling an Android Library

Inspecting the classes and methods of the Java classes can provide valuable information that will assist in binding a library. JD-GUI is a graphical utility that can display Java source code from the **CLASS** files contained in a JAR. It can be run as a stand alone application or as a plug-in for IntelliJ or Eclipse.

To decompile an Android library open the **.JAR** file with the Java decompiler. If the library is an **.AAR** file, it is necessary to extract the file **classes.jar** from the archive file. The following is a sample screenshot of using JD-GUI to analyze the Picasso JAR:



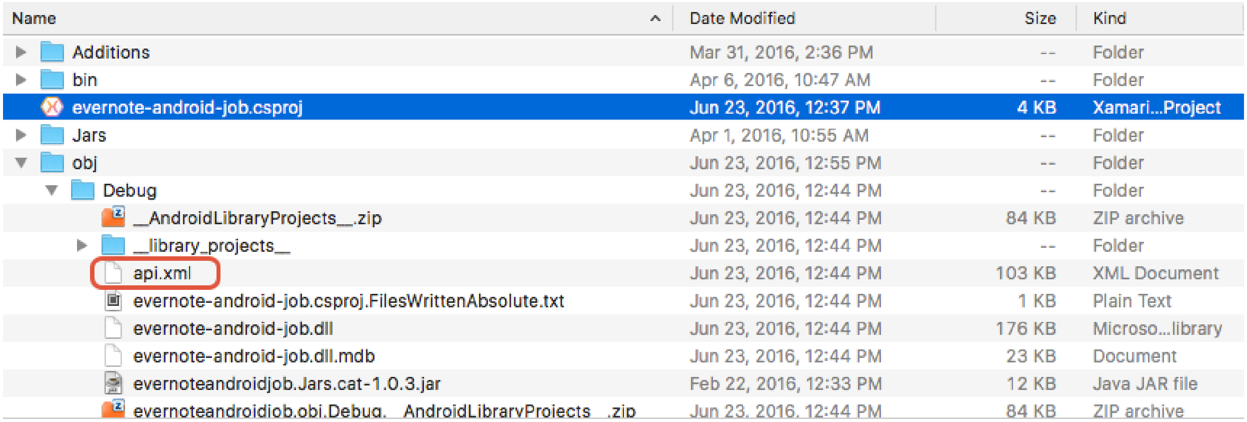
Once you have decompiled the Android library, examine the source code. Generally speaking, look for :

- **Classes that have characteristics of obfuscation** – Characteristics of obfuscated classes include:
  - The class name includes a **$**, i.e. **a$.class**
  - The class name is entirely compromised of lower case characters, i.e. **a.class**
- `import` **statements for unreferenced libraries** – Identify the unreferenced library and add those dependencies to the Xamarin.Android binding project with a **Build Action** of **ReferenceJar** or **EmbedddedReferenceJar**.

> **NOTE**
>
> Decompiling a Java library may be prohibited or subject to legal restrictions based on local laws or the license under which the Java library was published. If necessary, enlist the services of a legal professional before attempting to decompile a Java library and inspect the source code.

Inspect API.XML

As a part of building a binding project, Xamarin.Android will generate an XML file name **obj/Debug/api.xml**:



This file provides a list of all the Java APIs that Xamarin.Android is trying bind. The contents of this file can help identify any missing types or methods, duplicate binding. Although inspection of this file is tedious and time consuming, it can provide for clues on what might be causing any binding problems. For example, **api.xml** might reveal that a property is returning an inappropriate type, or that there are two types that share the same managed name.

Known Issues

This section will list some of the common error messages or symptoms that my occur when trying to bind an Android library.

Problem: Java Version Mismatch

Sometimes types will not be generated or unexpected crashes may occur because you are using either a newer or older version of Java compared to what the library was compiled with. Recompile the Android library with the same version of the JDK that your Xamarin.Android project is using.

Problem: At least one Java library is required

You receive the error "at least one Java library is required," even though a .JAR has been added.

Possible Causes:

Make sure the build action is set to `EmbeddedJar`. Since there are multiple build actions for .JAR files (such as `InputJar`, `EmbeddedJar`, `ReferenceJar` and `EmbeddedReferenceJar`), the binding generator cannot automatically guess which one to use by default. For more information about build actions, see Build Actions.

Problem: Binding tools cannot load the .JAR library

The binding library generator fails to load the .JAR library.

Possible Causes

Some .JAR libraries that use code obfuscation (via tools such as Proguard) cannot be loaded by the Java tools. Since our tool makes use of Java reflection and the ASM byte code engineering library, those dependent tools may reject the obfuscated libraries while Android runtime tools may pass. The workaround for this is to hand-bind these libraries instead of using the binding generator.

Problem: Missing C# types in generated output.

The binding **.dll** builds but misses some Java types, or the generated C# source does not build due to an error stating there are missing types.

Possible Causes:

This error may occur due to several reasons as listed below:

- The library being bound may reference a second Java library. If the public API for the bound library uses types from the second library, you must reference a managed binding for the second library as well.
- It is possible that a library was injected due to Java reflection, similar to the reason for the library load

error above, causing the unexpected loading of metadata. Xamarin.Android's tooling cannot currently resolve this situation. In such a case, the library must be manually bound.

- There was a bug in .NET 4.0 runtime that failed to load assemblies when it should have. This issue has been fixed in the .NET 4.5 runtime.

- Java allows deriving a public class from non-public class, but this is unsupported in .NET. Since the binding generator does not generate bindings for non-public classes, derived classes such as these cannot be generated correctly. To fix this, either remove the metadata entry for those derived classes using the remove-node in **Metadata.xml**, or fix the metadata that is making the non-public class public. Although the latter solution will create the binding so that the C# source will build, the non-public class should not be used.

  For example:

  ```
  <attr path="/api/package[@name='com.some.package']/class[@name='SomeClass']"
      name="visibility">public</attr>
  ```

- Tools that obfuscate Java libraries may interfere with the Xamarin.Android Binding Generator and its ability to generate C# wrapper classes. The following snippet shows how to update **Metadata.xml** to unobfuscate a class name:

  ```
  <attr path="/api/package[@name='{package_name}']/class[@name='{name}']"
      name="obfuscated">false</attr>
  ```

Problem: Generated C# source does not build due to parameter type mismatch

The generated C# source does not build. Overridden method's parameter types do not match.

Possible Causes:

Xamarin.Android includes a variety of Java fields that are mapped to enums in the C# bindings. These can cause type incompatibilities in the generated bindings. To resolve this, the method signatures created from the binding generator need to be modified to use the enums. For more information, please see Correcting Enums.

Problem: NoClassDefFoundError in packaging

`java.lang.NoClassDefFoundError` is thrown in the packaging step.

Possible Causes:

The most likely reason for this error is that a mandatory Java library needs to be added to the application project (**.csproj**). .JAR files are not automatically resolved. A Java library binding is not always generated against a user assembly that does not exist in the target device or emulator (such as Google Maps **maps.jar**). This is not the case for Android Library project support, as the library .JAR is embedded in the library dll. For example: Bug 4288

Problem: Duplicate custom EventArgs types

Build fails due to duplicate custom EventArgs types. An error like this occurs:

```
error CS0102: The type `Com.Google.Ads.Mediation.DismissScreenEventArgs' already contains a definition for
`p0'
```

Possible Causes:

This is because there is some conflict between event types that come from more than one interface "listener" type that shares methods having identical names. For example, if there are two Java interfaces as seen in the example below, the generator creates `DismissScreenEventArgs` for both `MediationBannerListener` and `MediationInterstitialListener`, resulting in the error.

```
// Java:
public interface MediationBannerListener {
    void onDismissScreen(MediationBannerAdapter p0);
}
public interface MediationInterstitialListener {
    void onDismissScreen(MediationInterstitialAdapter p0);
}
```

This is by design so that lengthy names on event argument types are avoided. To avoid these conflicts, some metadata transformation is required. Edit **Transforms\Metadata.xml** and add an `argsType` attribute on either of the interfaces (or on the interface method):

```
<attr path="/api/package[@name='com.google.ads.mediation']/
        interface[@name='MediationBannerListener']/method[@name='onDismissScreen']"
        name="argsType">BannerDismissScreenEventArgs</attr>

<attr path="/api/package[@name='com.google.ads.mediation']/
        interface[@name='MediationInterstitialListener']/method[@name='onDismissScreen']"
        name="argsType">IntersitionalDismissScreenEventArgs</attr>

<attr path="/api/package[@name='android.content']/
        interface[@name='DialogInterface.OnClickListener']"
        name="argsType">DialogClickEventArgs</attr>
```

Problem: Class does not implement interface method

An error message is produced indicating that a generated class does not implement a method that is required for an interface which the generated class implements. However, looking at the generated code, you can see that the method is implemented.

Here is an example of the error:

```
obj\Debug\generated\src\Oauth.Signpost.Basic.HttpURLConnectionRequestAdapter.cs(8,23):
error CS0738: 'Oauth.Signpost.Basic.HttpURLConnectionRequestAdapter' does not
implement interface member 'Oauth.Signpost.Http.IHttpRequest.Unwrap()'.
'Oauth.Signpost.Basic.HttpURLConnectionRequestAdapter.Unwrap()' cannot implement
'Oauth.Signpost.Http.IHttpRequest.Unwrap()' because it does not have the matching
return type of 'Java.Lang.Object'
```

Possible Causes:

This is a problem that occurs with binding Java methods with covariant return types. In this example, the method `Oauth.Signpost.Http.IHttpRequest.UnWrap()` needs to return `Java.Lang.Object`. However, the method `Oauth.Signpost.Basic.HttpURLConnectionRequestAdapter.UnWrap()` has a return type of `HttpURLConnection`. There are two ways to fix this issue:

- Add a partial class declaration for `HttpURLConnectionRequestAdapter` and explicitly implement `IHttpRequest.Unwrap()`:

```
namespace Oauth.Signpost.Basic {
    partial class HttpURLConnectionRequestAdapter {
        Java.Lang.Object OauthSignpost.Http.IHttpRequest.Unwrap() {
            return Unwrap();
        }
    }
}
```

- Remove the covariance from the generated C# code. This involves adding the following transform to **Transforms\Metadata.xml** which will cause the generated C# code to have a return type of

`Java.Lang.Object`:

```
<attr
path="/api/package[@name='oauth.signpost.basic']/class[@name='HttpURLConnectionRequestAdapter']/metho
d[@name='unwrap']"
    name="managedReturn">Java.Lang.Object
</attr>
```

Problem: Name Collisions on Inner Classes / Properties

Conflicting visibility on inherited objects.

In Java, it's not required that a derived class have the same visibility as its parent. Java will just fix that for you. In C#, that has to be explicit, so you need to make sure all classes in the hierarchy have the appropriate visibility. The following example shows how to change a Java package name from `com.evernote.android.job` to `Evernote.AndroidJob`:

```
<!-- Change the visibility of a class -->
<attr path="/api/package[@name='namespace']/class[@name='ClassName']" name="visibility">public</attr>

<!-- Change the visibility of a method -->
<attr path="/api/package[@name='namespace']/class[@name='ClassName']/method[@name='MethodName']"
name="visibility">public</attr>
```

Problem: A .so Library Required by the Binding is Not Loading

Some binding projects may also depend on functionality in a **.so** library. It is possible that Xamarin.Android will not automatically load the **.so** library. When the wrapped Java code executes, Xamarin.Android will fail to make the JNI call and the error message *java.lang.UnsatisfiedLinkError: Native method not found:* will appear in the logcat out for the application.

The fix for this is to manually load the **.so** library with a call to `Java.Lang.JavaSystem.LoadLibrary`. For example assuming that a Xamarin.Android project has shared library **libpocketsphinx_jni.so** included in the binding project with a build action of **EmbeddedNativeLibrary**, the following snippet (executed before using the shared library) will load the **.so** library:

```
Java.Lang.JavaSystem.LoadLibrary("pocketsphinx_jni");
```

Summary

In this article, we listed common troubleshooting issues associated with Java Bindings and explained how to resolve them.

Related Links

- Library Projects
- Working with JNI
- Enable Diagnostic Output
- Xamarin for Android Developers
- JD-GUI

Bind Android Kotlin libraries

11/2/2020 • 3 minutes to read • Edit Online

IMPORTANT

We're currently investigating custom binding usage on the Xamarin platform. Please take **this survey** to inform future development efforts.

The Android platform, along with its native languages and tooling, is constantly evolving and there are plenty of third-party libraries that have been developed using the latest offerings. Maximizing code and component reuse is one of the key goals of cross-platform development. The ability to reuse components built with Kotlin has become increasingly important to Xamarin developers as their popularity amongst developers continues to grow. You may already be familiar with the process of binding regular Java libraries. Additional documentation is now available describing the process of Binding a Kotlin Library, so they are consumable by a Xamarin application in the same manner. The purpose of this document is to describe a high-level approach to create a Kotlin Binding for Xamarin.

High-level approach

With Xamarin, you can bind any third-party native library to be consumable by a Xamarin application. Kotlin is the new language and creating binding for libraries built with this language requires some additional steps and tooling. This approach involves the following four steps:

1. Build the native library
2. Prepare the Xamarin metadata, which enables Xamarin tooling to generate C# classes
3. Build a Xamarin Binding Library using the native library and the metadata
4. Consume the Xamarin Binding Library in a Xamarin application

The following sections outline these steps with additional details.

Build the native library

The first step is to obtain a native Kotlin library (AAR package, which is an Android archive). You can either request it directly from a vendor or build it yourself.

Prepare the Xamarin metadata

The second step is to prepare the metadata transform file, which will be used by the Xamarin tools to generate the respective C# classes. In the best case scenario, this file could be empty where all classes are discovered and generated by the Xamarin tools. In some cases, metadata transformation must be applied to generate correct and/or desired C# code. In many cases, an AAR disassembler, such as Java Decompiler (JD), must be used to identify hidden dependencies and unwanted classes that you wish to exclude from the final list of C# classes to be generated. The final metadata should represent the public interface in which the referencing Xamarin.Android application will interact with.

Build a Xamarin.Android binding library

The third step is to create a special project - a Xamarin.Android Binding Library. It includes the Kotlin libraries as native references and the metadata transformation defined in the previous step. At time of writing, a separate Android Binding Library project is required for each AAR package being referenced. The Binding Library must add the Xamarin.Kotlin.StdLib package in order to support the Kotlin Standard Library.

Consume the Xamarin binding library

The fourth and the final step is to reference the binding library in a Xamarin.Android application. Adding a reference to the Xamarin.Android Binding Library enables your Xamarin application to use the exposed Kotlin classes from within that package.

Walkthrough

The approach above outlines the high-level steps required to create a Kotlin Binding for Xamarin. There are many lower-level steps involved and further details to consider when preparing these bindings in practice including adapting to changes in the native tools and languages. The intent is to help you to gain a deeper understanding of this concept and the high-level steps involved in this process. For a detailed step-by-step guide, refer to the Xamarin Kotlin Binding Walkthrough documentation.

Related links

- Android Studio
- Gradle Installation
- Visual Studio for Mac
- Java Decompiler
- BubblePicker Kotlin Library
- Binding Java Library
- XPath
- Java Binding Metadata
- Xamarin.Kotlin.StdLib NuGet
- Sample project repository

Walkthrough: Bind an Android Kotlin library

7/8/2021 • 11 minutes to read • Edit Online

> **IMPORTANT**
>
> We're currently investigating custom binding usage on the Xamarin platform. Please take **this survey** to inform future development efforts.

Xamarin enables mobile developers to create cross-platform native mobile apps using Visual Studio and C#. You can use the Android platform SDK components out of the box but in many cases you also want to use third-party SDKs written for that platform and Xamarin allows you to do it via bindings. In order to incorporate a third-party Android framework into your Xamarin.Android application, you need to create a Xamarin.Android binding for it before you can use it in your applications.

The Android platform, along with its native languages and tooling, are constantly evolving, including the recent introduction of the Kotlin language, which is set eventually to replace Java. There are a number of 3d party SDKs, which have already been migrated from Java to Kotlin and it presents us with new challenges. Even though the Kotlin binding process is similar to Java, it requires additional steps and configuration settings to successfully build and run as part of a Xamarin.Android application.

The goal of this document is to outline a high-level approach for addressing this scenario and provide a detailed step-by-step guide with a simple example.

Background

Kotlin was released in February 2016 and was positioned as an alternative to the standard Java compiler into Android Studio by 2017. Later in 2019, Google announced that the Kotlin programming language would became its preferred language for Android app developers. The high-level binding approach is similar to the binding process of regular Java libraries with a few important Kotlin specific steps.

Prerequisites

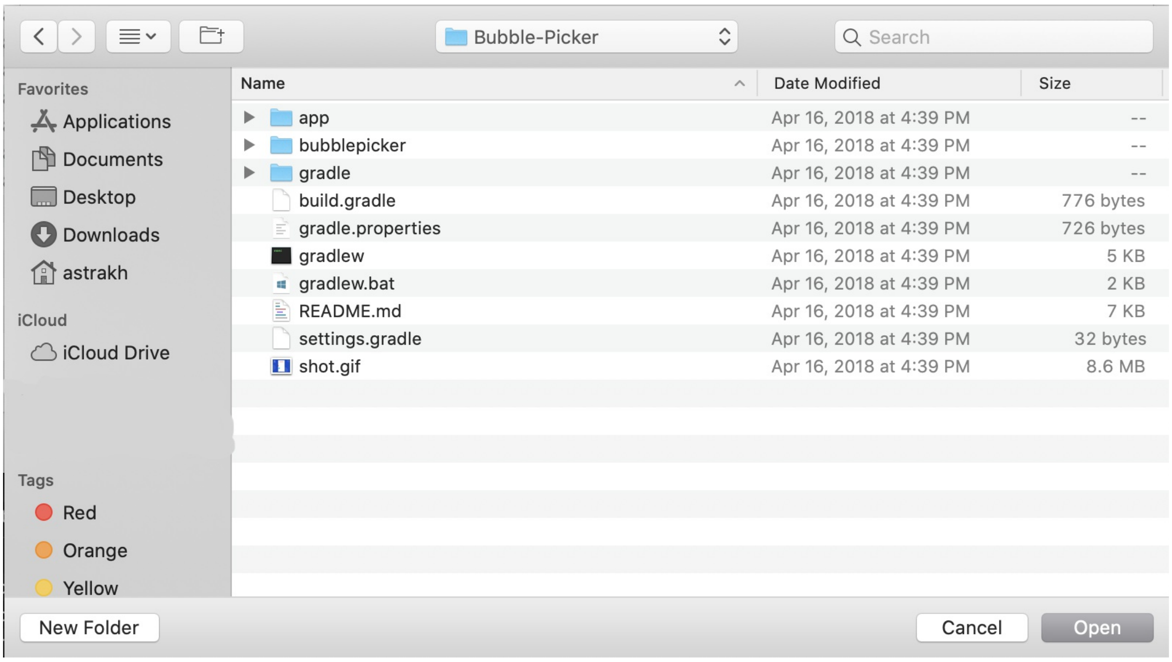
In order to complete this walkthrough, you will need:

- Android Studio
- Visual Studio for Mac
- Java Decompiler

Build a native library

The first step is to build a native Kotlin library using Android Studio. The library is usually provided by a third-party developer or available at the Google's Maven repository and other remote repositories. As an example, in this tutorial a binding for the Bubble Picker Kotlin Library is created:

1. Download the source code from GitHub for the library and unpack it to a local folder **Bubble-Picker**.
2. Launch the Android Studio and select **Open an existing Android Studio project** menu option choosing the Bubble-Picker local folder:



3. Verify that the Android Studio is up to date including Gradle. The source code can be successfully built on Android Studio v3.5.3, Gradle v5.4.1. Instructions on how to update Gradle to the latest Gradle version could be found here.
4. Verify that required Android SDK is installed. The source code requires Android SDK v25. Open **Tools > SDK Manager** menu option to install SDK components.

5. Update and synchronize the main **build.gradle** configuration file located at the root of the project folder:

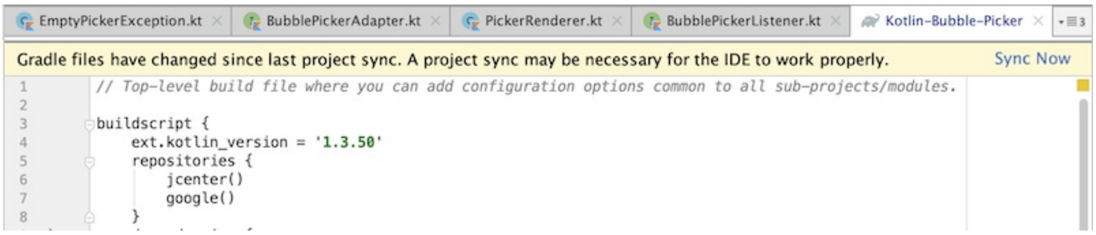
   - Set the Kotlin version to 1.3.10

```
buildscript {
    ext.kotlin_version = '1.3.10'
}
```

   - Register default Google's Maven repository so the support library dependency could be resolved:

```
allprojects {
    repositories {
        jcenter()
        maven {
            url "https://maven.google.com"
        }
    }
}
```

   - Once the configuration file is updated, it's out of sync and Gradle shows the **Sync Now** button, press it and wait for the synchronization process to be completed:

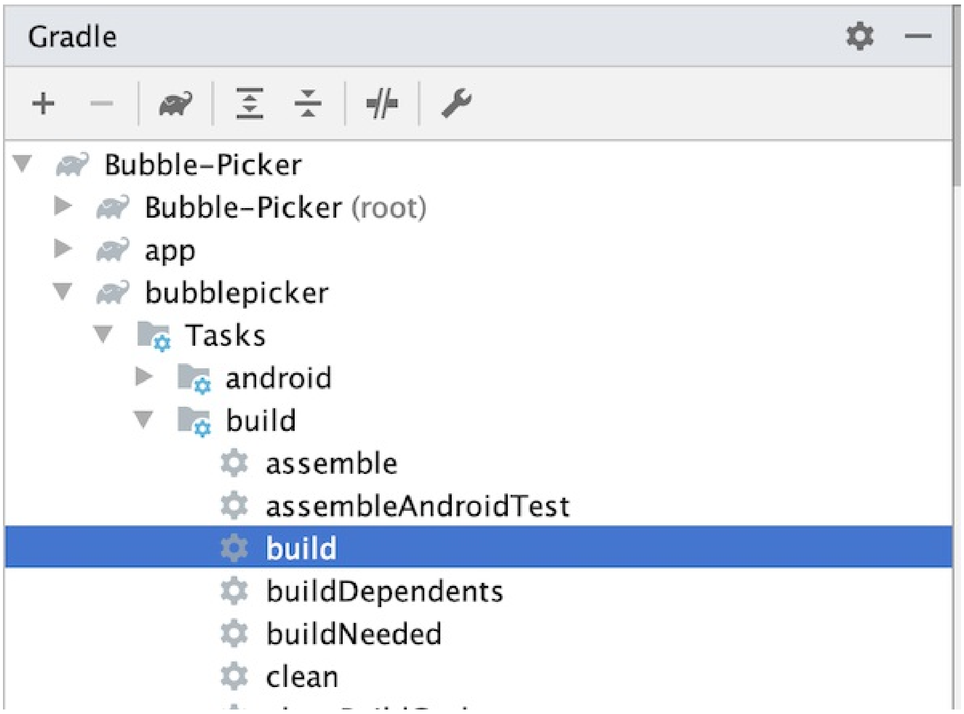


> **TIP**
>
> Gradle's dependency cache may be corrupt, this sometimes occurs after a network connection timeout. Redownload dependencies and sync project (requires network).

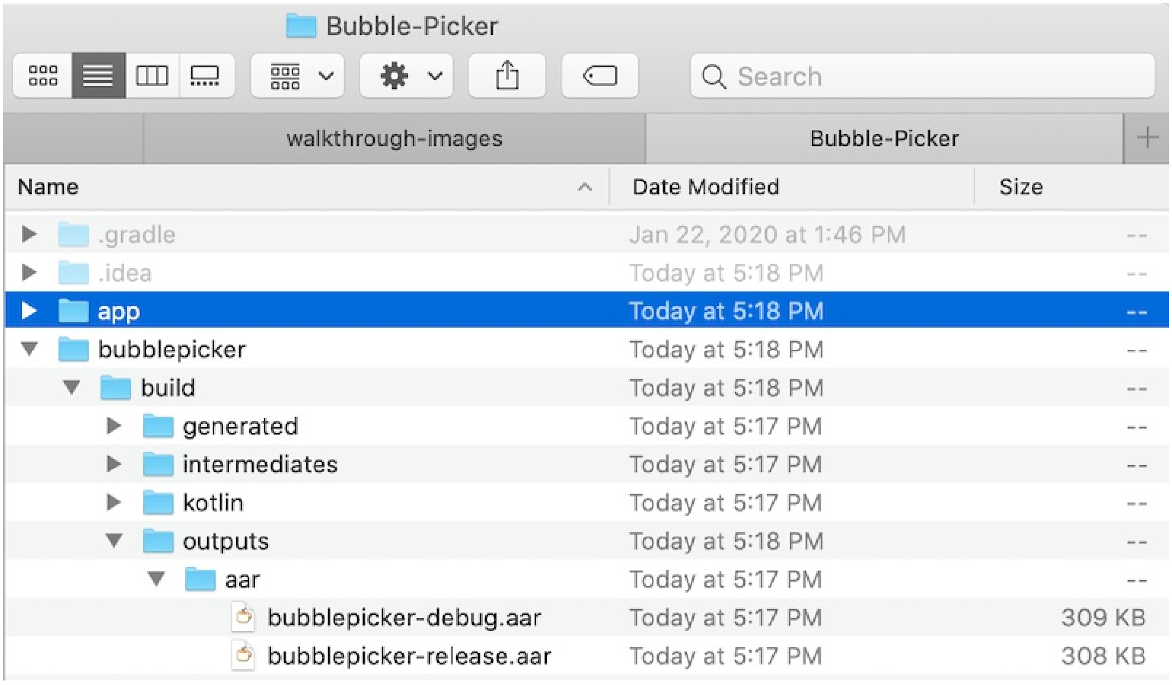
> **TIP**
>
> The state of a Gradle build process (daemon) may be corrupt. Stopping all Gradle daemons may solve this problem. Stop Gradle build processes (requires restart). In the case of corrupt Gradle processes, you can also try closing the IDE and then killing all Java processes.

> **TIP**
>
> Your project may be using a third-party plugin, which is not compatible with the other plugins in the project or the version of Gradle requested by the project.

6. Open the Gradle menu on the right, navigate to the **bubblepicker** > **Tasks** menu, execute the **build** task by double tapping on it, and wait for the build process to complete:



7. Open the root folder files browser and navigate to the build folder: **Bubble-Picker -> bubblepicker -> build -> outputs -> aar**, save the **bubblepicker-release.aar** file as **bubblepicker-v1.0.aar**, this file will be used later in the binding process:



The AAR file is an Android archive, which contains the compiled Kotlin source code and assets, required by Android to run an application using this SDK.

Prepare metadata

The second step is to prepare the metadata transformation file, which is used by Xamarin.Android to generate respective C# classes. A Xamarin.Android Binding Project will discover all native classes and members from a given Android archive subsequently generating an XML file with the appropriate metadata. The manually created metadata transformation file is then applied to the previously generated baseline to create the final XML definition file used to generate the C# code.

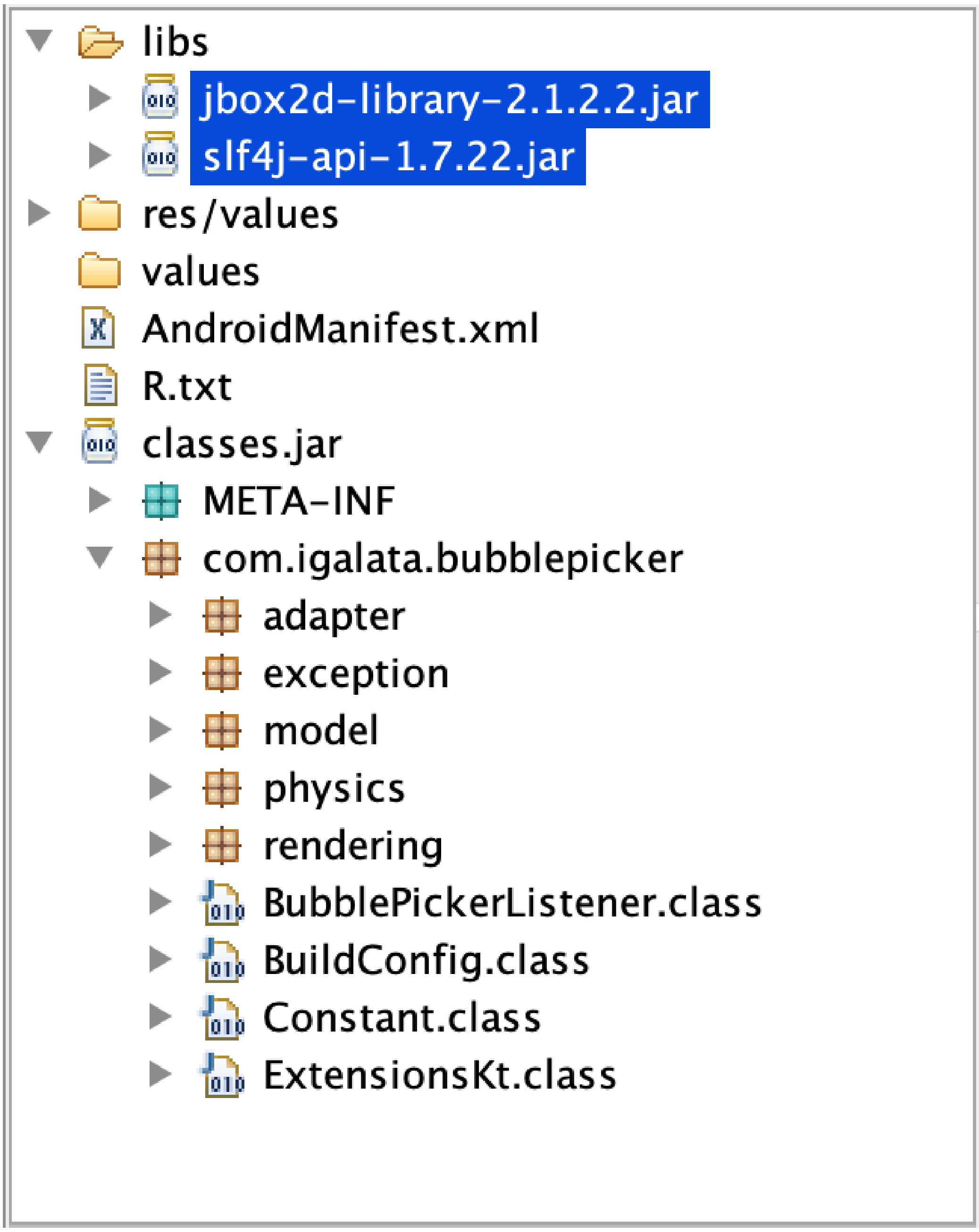
The metadata uses XPath syntax and is used by the Bindings Generator to influence the creation of the binding assembly. The Java Binding Metadata article provides more information on transformations, which could be applied:

1. Create an empty **Metadata.xml** file:

```
<?xml version="1.0" encoding="UTF-8"?>
<metadata>
</metadata>
```

2. Define xml transformations:

- The native Kotlin library has two dependencies, which you don't want to expose to C# world, define two transformations to ignore them completely. Important to say, the native members won't be stripped from the resulting binary, only C# classes won't be generated. Java Decompiler can be used to identify the dependencies. Run the tool and open the AAR file created earlier, as a result the structure of the Android archive will be shown, reflecting all dependencies, values, resources, manifest, and classes:



The transformations to skip processing these packages are defined using XPath instructions:

```
<remove-node path="/api/package[starts-with(@name,'org.jbox2d')]" />
<remove-node path="/api/package[starts-with(@name,'org.slf4j')]" />
```

- The native `BubblePicker` class has two methods `getBackgroundColor` and `setBackgroundColor` and the following transformation will change it into a C# `BackgroundColor` property:

```
<attr
path="/api/package[@name='com.igalata.bubblepicker.rendering']/class[@name='BubblePicker']/method[@na
me='getBackground' and count(parameter)=0]" name="propertyName">BackgroundColor</attr>
<attr
path="/api/package[@name='com.igalata.bubblepicker.rendering']/class[@name='BubblePicker']/method[@na
me='setBackground' and count(parameter)=1 and parameter[1][@type='int']]"
name="propertyName">BackgroundColor</attr>
```

- Unsigned types `UInt, UShort, ULong, UByte` require special handling. For these types Kotlin changes method names and parameters types automatically, which is reflected in the generated code:

```
public open fun fooUIntMethod(value: UInt) : String {
    return "fooUIntMethod${value}"
}
```

This code is compiled into the following Java byte code:

```
@NotNull
public String fooUIntMethod-WZ4Q5Ns(int value) {
return "fooUIntMethod" + UInt.toString-impl(value);
}
```

Moreover, related types such as `UIntArray, UShortArray, ULongArray, UByteArray` are also affected by Kotlin. The method name is changed to include an additional suffix and parameters are changed to an array of elements of signed versions of the same types. In the example below a parameter of type `UIntArray` is converted automatically into `int[]` and the method name is changed from `fooUIntArrayMethod` to `fooUIntArrayMethod--ajY-9A`. The latter is discovered by Xamarin.Android tools and generated as a valid method name:

```
public open fun fooUIntArrayMethod(value: UIntArray) : String {
    return "fooUIntArrayMethod${value.size}"
}
```

This code is compiled into the following Java byte code:

```
@NotNull
public String fooUIntArrayMethod--ajY-9A(@NotNull int[] value) {
    Intrinsics.checkParameterIsNotNull(value, "value");
    return "fooUIntArrayMethod" + UIntArray.getSize-impl(value);
}
```

In order to give it a meaningful name, the following metadata can be added to the **Metadata.xml**, which will update the name back to originally defined in the Kotlin code:

```
<attr
path="/api/package[@name='com.microsoft.simplekotlinlib']/class[@name='FooClass']/method[@name='fooUI
ntArrayMethod--ajY-9A']" name="managedName">fooUIntArrayMethod</attr>
```

In the BubblePicker sample, there are no members using unsigned types thus no additional changes are required.

- Kotlin members with generic parameters by default transformed into parameters of Java. `Lang.Object` type. For example, a Kotlin method has a generic parameter <T>:

```
public open fun <T>fooGenericMethod(value: T) : String {
return "fooGenericMethod${value}"
}
```

Once a Xamarin.Android binding is generated, the method is exposed to C# as below:

```
[Register ("fooGenericMethod", "(Ljava/lang/Object;)Ljava/lang/String;",
"GetFooGenericMethod_Ljava_lang_Object_Handler")]
[JavaTypeParameters (new string[] {
    "T"
})]

public virtual string FooGenericMethod (Java.Lang.Object value);
```

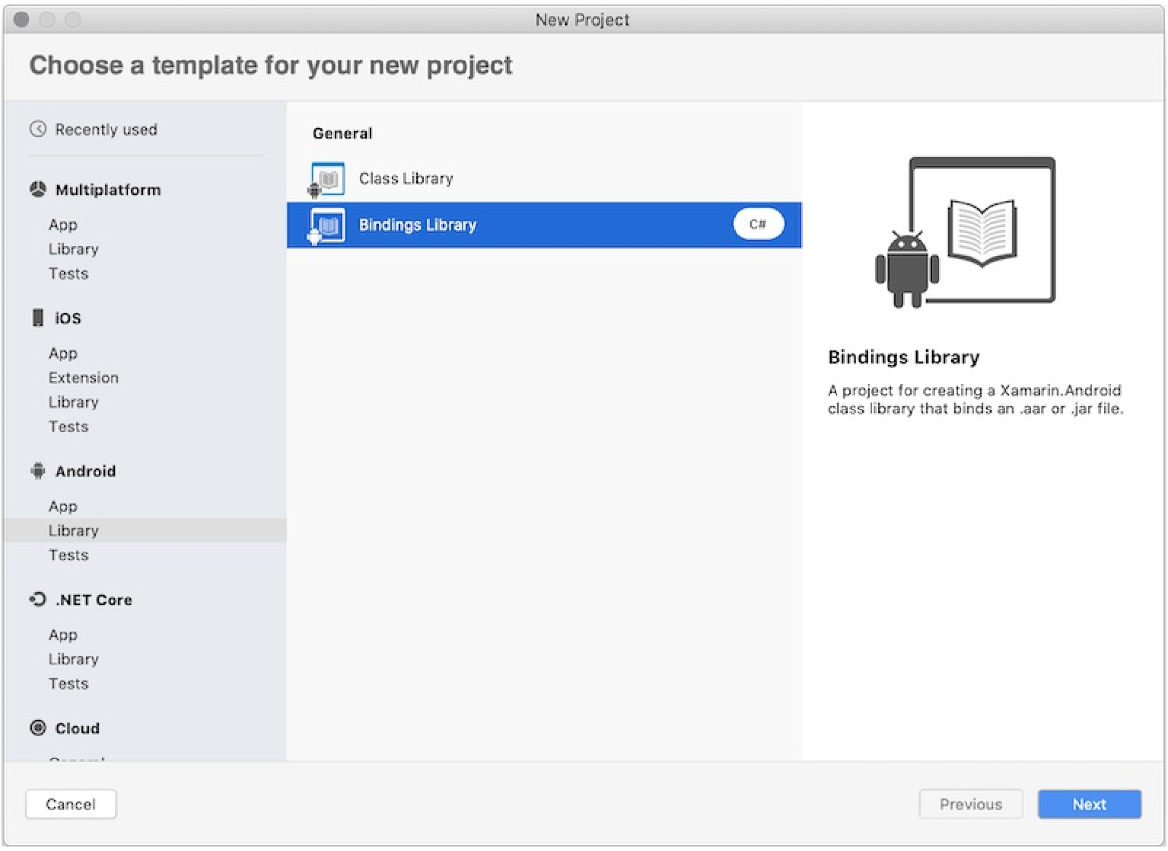
Java and Kotlin generics are not supported by Xamarin.Android bindings, thus a generalized C# method to access the generic API is created. As a work-around you can create a wrapper Kotlin library and expose required APIs in a strong-typed manner without generics. Alternatively, you can create helpers on C# side to address the issue in the same way via strong-typed APIs.

> **TIP**
> By transforming the metadata, any changes could be applied to the generated binding. The Binding Java Library article explains in details how the metadata is generated and processed.

Build a binding library

The next step is to create a Xamarin.Android binding project using the Visual Studio binding template, add required metadata, native references and then build the project to produce a consumable library:

1. Open Visual Studio for Mac and create a new Xamarin.Android Binding Library project, give it a name, in this case **testBubblePicker.Binding** and complete the wizard. The Xamarin.Android binding template is located by the following path: **Android > Library > Binding Library**:

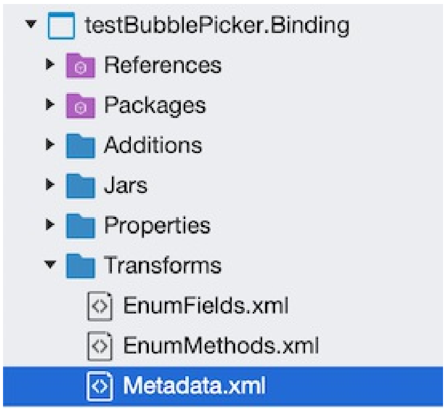


In the Transformations folder there are three main transformation files:

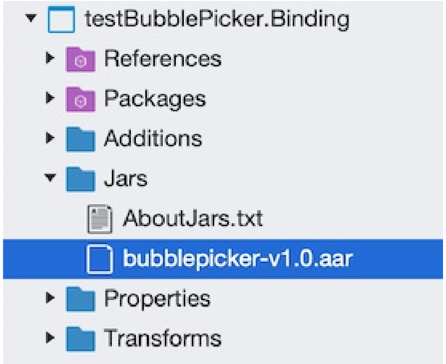
- **Metadata.xml** – Allows changes to be made to the final API, such as changing the namespace of the generated binding.
- **EnumFields.xml** – Contains the mapping between Java int constants and C# enums.
- **EnumMethods.xml** – Allows changing method parameters and return types from Java int constants to C# enums.

Keep empty the **EnumFields.xml** and **EnumMethods.xml** files and update the **Metadata.xml** to define your transformations.

2. Replace the existing **Transformations/Metadata.xml** file with the **Metadata.xml** file created at the previous step. In the properties window, verify that the file **Build Action** is set to **TransformationFile**:



3. Add the **bubblepicker-v1.0.aar** file you built in Step 1 to the binding project as a native reference. To add native library references, open finder and navigate to the folder with the Android archive. Drag and drop the archive into the Jars folder in Solution Explorer. Alternatively, you can use the **Add** context menu option on the Jars folder and choose **Existing Files…**. Choose to copy the file to the directory for the purposes of this walkthrough. Be sure to verify that the **Build Action** is set to **LibraryProjectZip**:



4. Add a reference to the Xamarin.Kotlin.StdLib NuGet package. This package is a binding for Kotlin Standard Library. Without this package, the binding will only work if the Kotlin library doesn't use any Kotlin specific types, otherwise all these members will not be exposed to C# and any app that tries to consume the binding will crash at runtime.

   TIP

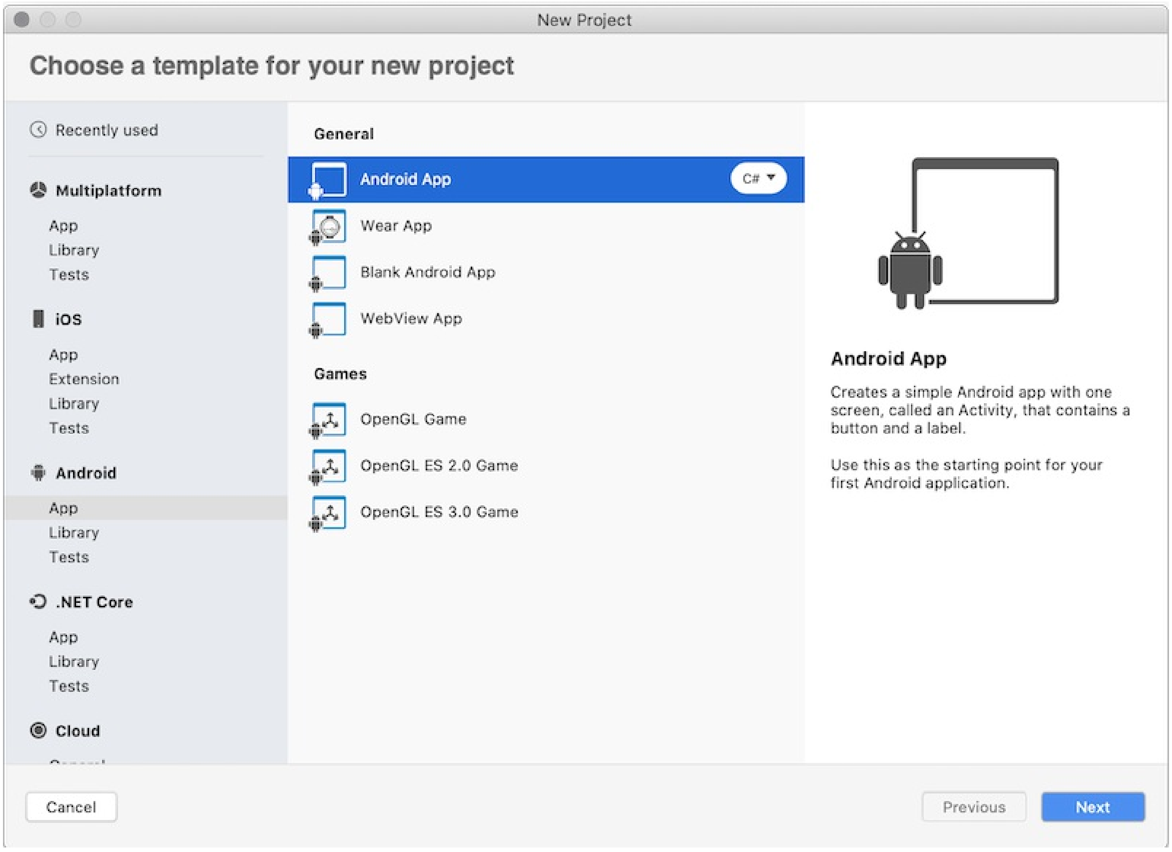
   Due to a limitation of the Xamarin.Android, binding tools only a single Android archive (AAR) can be added per binding project. If multiple AAR files need to be included, then multiple Xamarin.Android projects are required, one per each AAR. If this were the case for this walkthrough, then the previous four actions of this step would have to be repeated for each archive. As an alternative option, it is possible to manually merge multiple Android archives as a single archive and as a result you could use a single Xamarin.Android binding project.

5. The final action is to build the library and make don't have any compilation errors. In case of compilation errors, they can be addressed and handled using the Metadata.xml file, which you created earlier by adding xml transformation metadata, which will add, remove, or rename library members.

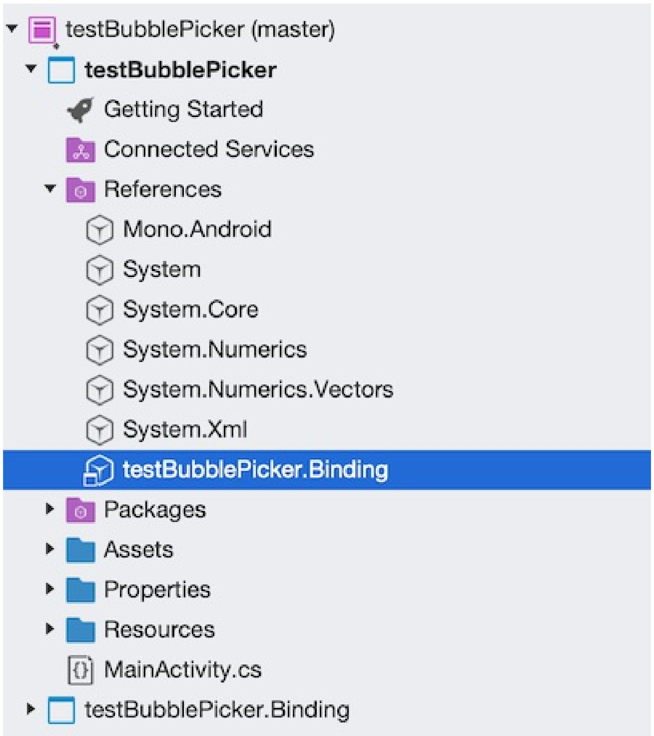
Consume the binding library

The final step is to consume the Xamarin.Android binding library in a Xamarin.Android application. Create a new Xamarin.Android project, add reference to the binding library and render Bubble Picker UI:

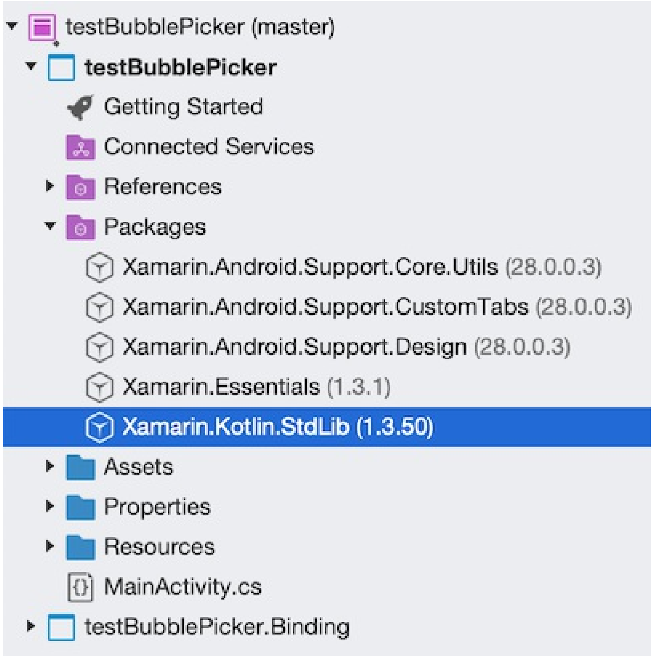
1. Create Xamarin.Android project. Use the **Android** > **App** > **Android App** as a starting point and select **Latest and Greatest** as you Target Platforms option to avoid compatibility issues. All the following steps target this project:



2. Add a project reference to the binding project or add a reference the DLL created previously:



3. Add a reference to the Xamarin.Kotlin.StdLib NuGet package, that you added to the Xamarin.Android binding project earlier. It adds support to any Kotlin specific types that need handing in runtime. Without this package the app can be compiled but will crash at runtime:



4. Add the `BubblePicker` control to the Android layout for `MainActivity`. Open **testBubblePicker/Resources/layout/content_main.xml** file and append the BubblePicker control node as the last element of the root RelativeLayout control:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout …>
    …
    <com.igalata.bubblepicker.rendering.BubblePicker
        android:id="@+id/picker"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:backgroundColor="@android:color/white" />
</RelativeLayout>
```

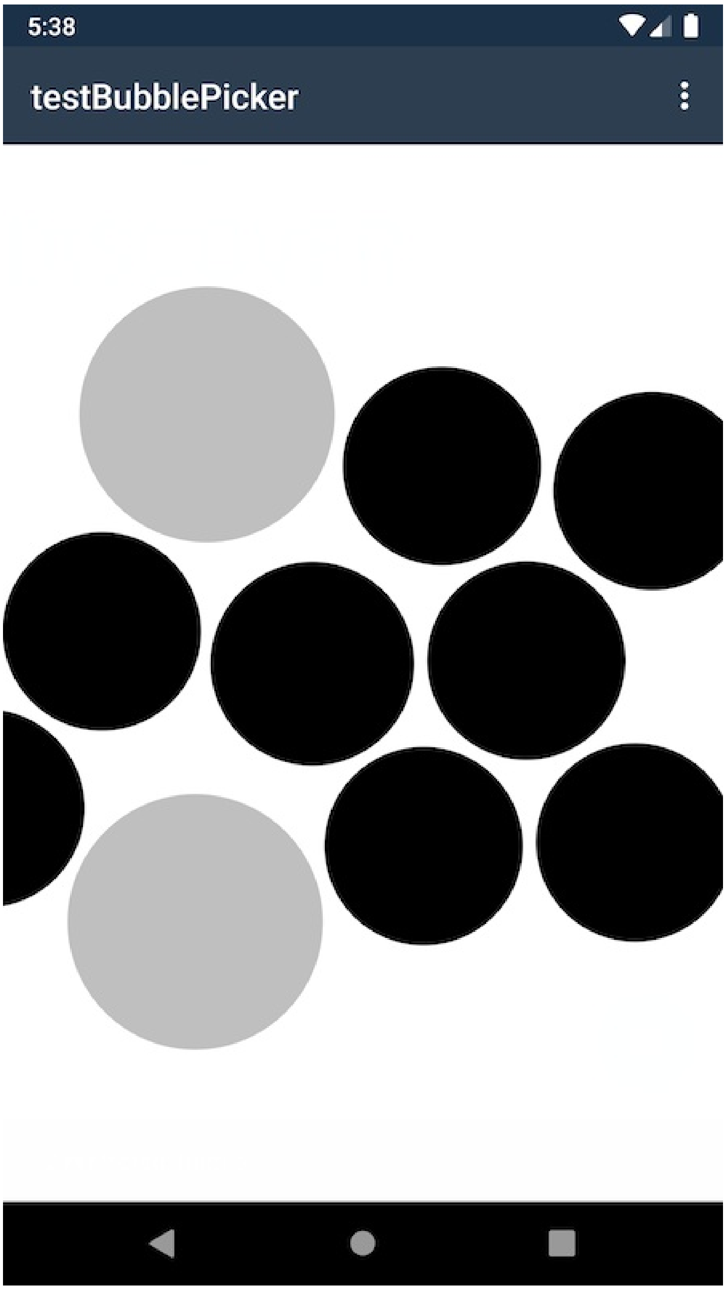
5. Update the source code of the app and add the initialization logic to the `MainActivity`, which activates the Bubble Picker SDK:

```
protected override void OnCreate(Bundle savedInstanceState)
{
    ...
    var picker = FindViewById<BubblePicker>(Resource.Id.picker);
    picker.BubbleSize = 20;
    picker.Adapter = new BubblePickerAdapter();
    picker.Listener = new BubblePickerListener(picker);
    ...
}
```

`BubblePickerAdapter` and `BubblePickerListener` are two classes to be created from scratch, which handle the bubbles data and control interaction:

```
public class BubblePickerAdapter : Java.Lang.Object, IBubblePickerAdapter
{
    private List<string> _bubbles = new List<string>();
    public int TotalCount => _bubbles.Count;
    public BubblePickerAdapter()
    {
        for (int i = 0; i < 10; i++)
        {
            _bubbles.Add($"Item {i}");
        }
    }

    public PickerItem GetItem(int itemIndex)
    {
        if (itemIndex < 0 || itemIndex >= _bubbles.Count)
            return null;

        var result = _bubbles[itemIndex];
        var item = new PickerItem(result);
        return item;
    }
}

public class BubblePickerListener : Java.Lang.Object, IBubblePickerListener
{
    public View Picker { get; }
    public BubblePickerListener(View picker)
    {
        Picker = picker;
    }

    public void OnBubbleDeselected(PickerItem item)
    {
        Snackbar.Make(Picker, $"Deselected: {item.Title}", Snackbar.LengthLong)
            .SetAction("Action", (Android.Views.View.IOnClickListener)null)
            .Show();
    }

    public void OnBubbleSelected(PickerItem item)
    {
        Snackbar.Make(Picker, $"Selected: {item.Title}", Snackbar.LengthLong)
        .SetAction("Action", (Android.Views.View.IOnClickListener)null)
        .Show();
    }
}
```

6. Run the app, which should render the Bubble Picker UI:



The sample requires additional code to render elements style and handle interactions but the `BubblePicker` control has been successfully created and activated.

Congratulations! You have successfully created a Xamarin.Android app and a binding library, which consumes a Kotlin library.

You should now have a basic Xamarin.Android application that uses a native Kotlin library via a Xamarin.Android binding library. This walkthrough intentionally uses a basic example to better emphasize the key concepts being introduced. In real world scenarios, you will likely be required to expose a greater number of APIs and apply metadata transformations to them.

Related links

- Android Studio
- Gradle Installation
- Visual Studio for Mac
- Java Decompiler
- BubblePicker Kotlin Library
- Binding Java Library

- XPath
- Java Binding Metadata
- Xamarin.Kotlin.StdLib NuGet
- Sample project repository

Using Native Libraries

4/12/2021 • 2 minutes to read • Edit Online

Xamarin.Android supports the use of native libraries via the standard PInvoke mechanism. You can also bundle additional native libraries which are not part of the OS into your .apk.

To deploy a native library with a Xamarin.Android application, add the library binary to the project and set its **Build Action** to **AndroidNativeLibrary**.

To deploy a native library with a Xamarin.Android library project, add the library binary to the project and set its **Build Action** to **EmbeddedNativeLibrary**.

Note that since Android supports multiple Application Binary Interfaces (ABIs), Xamarin.Android must know which ABI the native library is built for. There are two ways this can be done:

1. Path "sniffing"
2. By using an `AndroidNativeLibrary/Abi` element within the project file

With path sniffing, the parent directory name of the native library is used to specify the ABI that the library targets. Thus, if you add `lib/armeabi/libfoo.so` to the project, then the ABI will be "sniffed" as `armeabi`.

Alternatively, you can edit your project file to explicitly specify the ABI to use:

```
<ItemGroup>
    <AndroidNativeLibrary Include="path/to/libfoo.so">
        <Abi>armeabi</Abi>
    </AndroidNativeLibrary>
</ItemGroup>
```

For more information about using native libraries, see Interop with native libraries.

Debugging Native Code with Visual Studio

If you're using *Visual Studio 2019* or *Visual Studio 2017*, you don't have to modify your project files as described above. You can build and debug C++ inside your Xamarin.Android solution by adding a project reference to a C++ **Dynamic Shared Library (Android)** project.

To debug native C++ code in your project, follow these steps:

1. Double-click project **Properties** and select the **Android Options** page.
2. Scroll down to **Debugging options**.
3. In the **Debugger** dropdown menu, select **C++** (instead of the default **.NET (Xamarin)**).

Visual Studio C++ developers can see the SanAngeles_NativeDebug sample to try debugging C++ from Visual Studio 2019 or Visual Studio 2017 with Xamarin; and refer to our blog post for more information.

Related Links

- SanAngeles_NativeDebug (sample)
- Developing Xamarin Android Native Applications

An Introduction to Renderscript

7/8/2021 • 7 minutes to read • Edit Online

This guide introduces Renderscript and explains how to use the intrinsic Renderscript API's in a Xamarin.Android application that targets API level 17 or higher.

Overview

Renderscript is a programming framework created by Google for the purpose of improving the performance of Android applications that require extensive computational resources. It is a low level, high performance API based on C99. Because it is a low level API that will run on CPUs, GPUs, or DSPs, Renderscript is well suited for Android apps that may need to perform any of the following:

- Graphics
- Image Processing
- Encryption
- Signal Processing
- Mathematical Routines

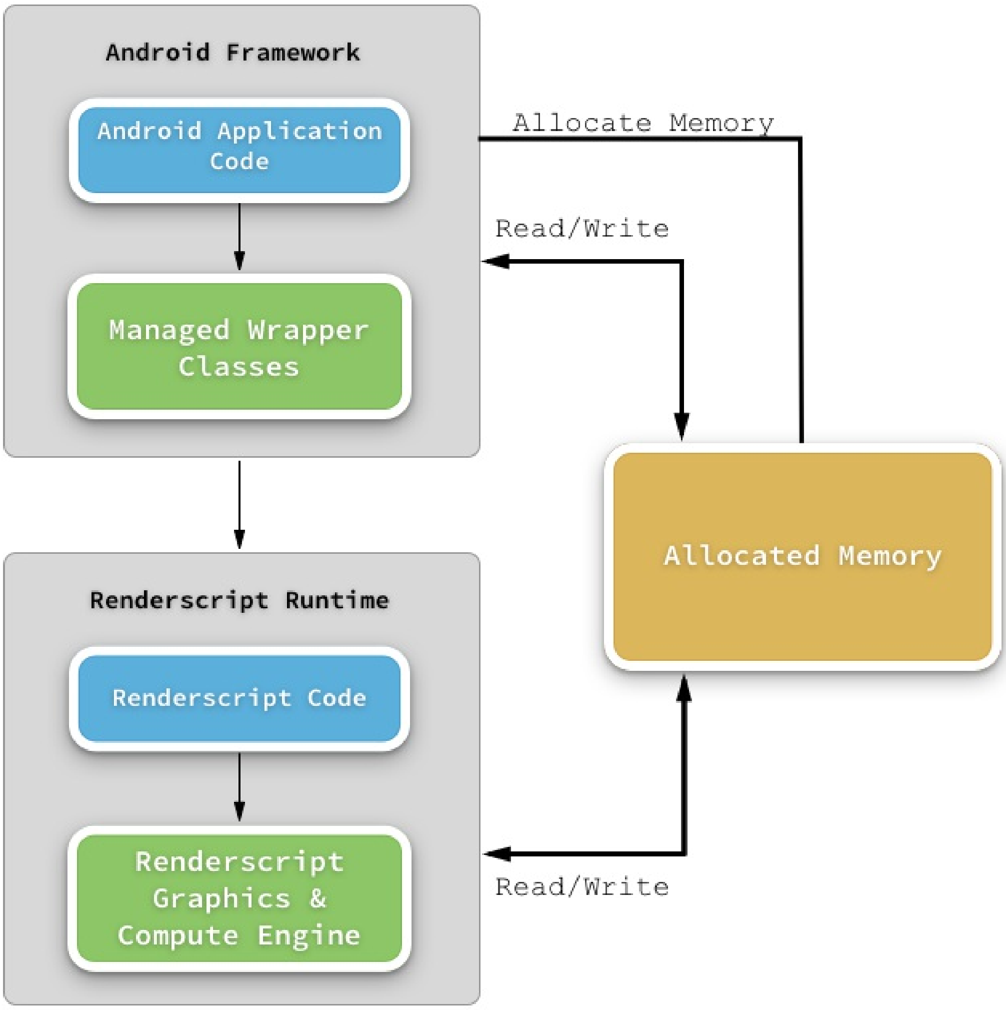
Renderscript will use `clang` and compile the scripts to LLVM byte code which is bundled into the APK. When the app is run for the first time, the LLVM byte code will be compiled into machine code for the processors on the device. This architecture allows an Android application to exploit the advantages of machine code without the developers themselves having to write it for each processor on the device themselves.

There are two components to a Renderscript routine:

1. **The Renderscript runtime** – This is the native APIs that are responsible for executing the Renderscript. This includes any Renderscripts written for the application.

2. **Managed Wrappers from the Android Framework** – Managed classes that allow an Android app to control and interact with the Renderscript runtime and scripts. In addition to the framework provided classes for controlling the Renderscript runtime, the Android toolchain will examine the Renderscript source code and generate managed wrapper classes for use by the Android application.

The following diagram illustrates how these components relate:



There are three important concepts for using Renderscripts in an Android application:

1. **A context** – A managed API provided by the Android SDK that allocates resources to Renderscript and allows the Android app to pass and receive data from the Renderscript.

2. **A *compute kernel*** – Also known as the *root kernel* or *kernel*, this a routine that does the work. The kernel is very similar to a C function; it is a parallelizable routine that will be run over all the data in allocated memory .

3. **Allocated Memory** – Data is passed to and from a kernel through an *Allocation*. A kernel may have one input and/or one output Allocation.

The Android.Renderscripts namespace contains the classes for interacting with the Renderscript runtime. In particular, the `Renderscript` class will manage the lifecycle and resources of the Renderscript engine. The Android app must initialize one or more `Android.Renderscripts.Allocation` objects. An Allocation is a managed API that is responsible for allocation and accessing the memory that is shared between the Android app and the Renderscript runtime. Typically, one Allocation is created for input, and optionally another Allocation is created to hold the output of the kernel. The Renderscript runtime engine and the associated managed wrapper classes will manage access to the memory held by the Allocations, there is no need for an Android app developer to do any extra work.

An Allocation will contain one or more Android.Renderscripts.Elements. Elements are a specialized type that describe data in each Allocation. The Element types of the output Allocation must match the types of the input Element. When executing, a Renderscript will iterate over each Element in the input Allocation in parallel, and write the results to the output Allocation. There are two types of Elements:

- **simple type** – Conceptually this is the same as a C data type, `float` or a `char`.

- **complex type** – This type is similar to a C `struct`.

The Renderscript engine will perform a runtime check to ensure that the Elements in each Allocation are compatible with what is required by the kernel. If the data type of the Elements in the Allocation do not match the data type that the kernel is expecting, an exception will be thrown.

All Renderscript kernels will be wrapped by a type that is a descendant of the `Android.Renderscripts.Script` class. The `Script` class is used to set parameters for a Renderscript, set the appropriate `Allocations`, and run the Renderscript. There are two `Script` subclasses in the Android SDK:

- `Android.Renderscripts.ScriptIntrinsic` – Some of the more common Renderscript tasks are bundled in the Android SDK and are accessible by a subclass of the ScriptIntrinsic class. There is no need for a developer take any extra steps to use these scripts in their application as they are already provided.

- `ScriptC_XXXXX` – Also known as *user scripts*, these are scripts that are written by developers and packaged in the APK. At compile time, the Android toolchain will generate managed wrapper classes that will allow the scripts to be used in the Android app. The name of these generated classes is the name of the Renderscript file, prefixed with `ScriptC_`. Writing and incorporating user scripts is not officially supported by Xamarin.Android and beyond the scope of this guide.

Of these two types, only the `StringIntrinsic` is supported by Xamarin.Android. This guide will discuss how to use intrinsic scripts in a Xamarin.Android application.

Requirements

This guide is for Xamarin.Android applications that target API level 17 or higher. The use of *user scripts* is not covered in this guide.

The Xamarin.Android V8 Support Library backports the instrinsic Renderscript API's for apps that target older versions of the Android SDK. Adding this package to a Xamarin.Android project should allow apps that target older versions of the Android SDK to leverage the intrinsic scripts.

Using Intrinsic Renderscripts in Xamarin.Android

The intrinsic scripts are a great way to perform intensive computing tasks with a minimal amount of additional code. They have been hand tuned to offer optimal performance on a large cross section of devices. It is not uncommon for an intrinsic script to run 10x faster than managed code and 2-3x times after than a custom C implementation. Many of the typical processing scenarios are covered by the intrinsic scripts. This list of the intrinsic scripts describes the current scripts in Xamarin.Android:

- ScriptIntrinsic3DLUT – Converts RGB to RGBA using a 3D lookup table.

- ScriptIntrinsicBLAS – Provideshigh performance Renderscript APIs to BLAS. The BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations.

- ScriptIntrinsicBlend – Blends two Allocations together.

- ScriptIntrinsicBlur – Applies a Gaussian blur to an Allocation.

- ScriptIntrinsicColorMatrix – Applies a color matrix to an Allocation (i.e. change colours, adjust hue).

- ScriptIntrinsicConvolve3x3 – Applies a 3x3 color matrix to an Allocation.

- ScriptIntrinsicConvolve5x5 – Applies a 5x5 color matrix to an Allocation.

- ScriptIntrinsicHistogram – An intrinsic histogram filter.

- ScriptIntrinsicLUT – Applies a per-channel lookup table to a buffer.
- ScriptIntrinsicResize – Script for performing the resize of a 2D allocation.
- ScriptIntrinsicYuvToRGB – Converts a YUV buffer to RGB.

Please consult the API documentation for details on each of the intrinsic scripts.

The basic steps for using Renderscript in an Android application are described next.

Create a Renderscript Context – The `Renderscript` class is a managed wrapper around the Renderscript context and will control initialization, resource management, and clean up. The Renderscript object is created using the `RenderScript.Create` factory method, which takes an Android Context (such as an Activity) as a parameter. The following line of code demonstrates how to initialize the Renderscript context:

```
Android.Renderscripts.RenderScript renderScript = RenderScript.Create(this);
```

Create Allocations – Depending on the intrinsic script, it may be necessary to create one or two `Allocation` s. The `Android.Renderscripts.Allocation` class has several factory methods to help with instantiating an allocation for an intrinsic. As an example, the following code snippet demonstrates how to create Allocation for Bitmaps.

```
Android.Graphics.Bitmap originalBitmap;
Android.Renderscripts.Allocation inputAllocation = Allocation.CreateFromBitmap(renderScript,
                                                     originalBitmap,
                                                     Allocation.MipmapControl.MipmapFull,
                                                     AllocationUsage.Script);
```

Often, it will be necessary to create an `Allocation` to hold the output data of a script. This following snippet shows how to use the `Allocation.CreateTyped` helper to instantiate a second `Allocation` that the same type as the original:

```
Android.Renderscripts.Allocation outputAllocation = Allocation.CreateTyped(renderScript,
inputAllocation.Type);
```

Instantiate the Script wrapper – Each of the intrinsic script wrapper classes should have helper methods (typically called `Create`)for instantiating a wrapper object for that script. The following code snippet is an example of how to instantiate a `ScriptIntrinsicBlur` blur object. The `Element.U8_4` helper method will create an Element that describes a data type that is 4 fields of 8-bit, unsigned integer values, suitable for holding the data of a `Bitmap` object:

```
Android.Renderscripts.ScriptIntrinsicBlur blurScript = ScriptIntrinsicBlur.Create(renderScript,
Element.U8_4(renderScript));
```

Assign Allocation(s), Set Parameters, & Run Script – The `Script` class provides a `ForEach` method to actually run the Renderscript. This method will iterate over each `Element` in the `Allocation` holding the input data. In some cases, it may be necessary to provide an `Allocation` that holds the output. `ForEach` will overwrite the contents of the output Allocation. To carry on with the code snippets from the previous steps, this example shows how to assign an input Allocation, set a parameter, and then finally run the script (copying the results to the output Allocation):

```
blurScript.SetInput(inputAllocation);
blurScript.SetRadius(25);  // Set a pamaeter
blurScript.ForEach(outputAllocation);
```

You may wish to check out the Blur an Image with Renderscript recipe, it is a complete example of how to use an intrinsic script in Xamarin.Android.

Summary

This guide introduced Renderscript and how to use it in a Xamarin.Android application. It briefly discussed what Renderscript is and how it works in an Android application. It described some of the key components in Renderscript and the difference between *user scripts* and *instrinsic scripts*. Finally, this guide discussed the steps in using an intrinsic script in a Xamarin.Android application.

Related Links

- Android.Renderscripts namespace
- Blur an Image with Renderscript
- Renderscript
- Tutorial: Getting Started with Renderscript

Xamarin.Essentials

3/5/2021 • 2 minutes to read • Edit Online

Xamarin.Essentials provides developers with cross-platform APIs for their mobile applications.

Android, iOS, and UWP offer unique operating system and platform APIs that developers have access to all in C# leveraging Xamarin. Xamarin.Essentials provides a single cross-platform API that works with any Xamarin.Forms, Android, iOS, or UWP application that can be accessed from shared code no matter how the user interface is created.

Get Started with Xamarin.Essentials

Follow the getting started guide to install the **Xamarin.Essentials** NuGet package into your existing or new Xamarin.Forms, Android, iOS, or UWP projects.

Feature Guides

Follow the guides to integrate these Xamarin.Essentials features into your applications:

- Accelerometer – Retrieve acceleration data of the device in three dimensional space.
- App Actions – Get and set shortcuts for the application.
- App Information – Find out information about the application.
- App Theme – Detect the current theme requested for the application.
- Barometer – Monitor the barometer for pressure changes.
- Battery – Easily detect battery level, source, and state.
- Clipboard – Quickly and easily set or read text on the clipboard.
- Color Converters – Helper methods for System.Drawing.Color.
- Compass – Monitor compass for changes.
- Connectivity – Check connectivity state and detect changes.
- Contacts – Retrieve information about a contact on the device.
- Detect Shake – Detect a shake movement of the device.
- Device Display Information – Get the device's screen metrics and orientation.
- Device Information – Find out about the device with ease.
- Email – Easily send email messages.
- File Picker – Allow user to pick files from the device.
- File System Helpers – Easily save files to app data.
- Flashlight – A simple way to turn the flashlight on/off.
- Geocoding – Geocode and reverse geocode addresses and coordinates.
- Geolocation – Retrieve the device's GPS location.
- Gyroscope – Track rotation around the device's three primary axes.
- Haptic Feedback – Control click and long press haptics.
- Launcher – Enables an application to open a URI by the system.
- Magnetometer – Detect device's orientation relative to Earth's magnetic field.
- MainThread – Run code on the application's main thread.
- Maps – Open the maps application to a specific location.
- Media Picker – Allow user to pick or take photos and videos.

- Open Browser – Quickly and easily open a browser to a specific website.
- Orientation Sensor – Retrieve the orientation of the device in three dimensional space.
- Permissions – Check and request permissions from users.
- Phone Dialer – Open the phone dialer.
- Platform Extensions – Helper methods for converting Rect, Size, and Point.
- Preferences – Quickly and easily add persistent preferences.
- Screenshot – Take a capture of the current display of the application.
- Secure Storage – Securely store data.
- Share – Send text and website links to other apps.
- SMS – Create an SMS message for sending.
- Text-to-Speech – Vocalize text on the device.
- Unit Converters – Helper methods to convert units.
- Version Tracking – Track the applications version and build numbers.
- Vibrate – Make the device vibrate.
- Web Authenticator - Start web authentication flows and listen for a callback.

Troubleshooting

Find help if you are running into issues.

Xamarin.Essentials on Q&A

Ask questions about accessing native features with Xamarin.Essentials.

Release Notes

Find full release notes for each release of Xamarin.Essentials.

API Documentation

Browse the API documentation for every feature of Xamarin.Essentials.

Get Started with Xamarin.Essentials

3/5/2021 • 2 minutes to read • Edit Online

Xamarin.Essentials provides a single cross-platform API that works with any iOS, Android, or UWP application that can be accessed from shared code no matter how the user interface is created. See the platform & feature support guide for more information on supported operating systems.

Installation

Xamarin.Essentials is available as a NuGet package and is included in every new project in Visual Studio. It can also be added to any existing projects using Visual Studio with the following steps.

1. Download and install Visual Studio with the Visual Studio tools for Xamarin.
2. Open an existing project, or create a new project using the Blank App template under **Visual Studio C#** (Android, iPhone & iPad, or Cross-Platform).

   > **IMPORTANT**
   > If adding to a UWP project ensure Build 16299 or higher is set in the project properties.

3. Add the **Xamarin.Essentials** NuGet package to each project:

   - Visual Studio
   - Visual Studio for Mac

   In the Solution Explorer panel, right click on the solution name and select **Manage NuGet Packages**. Search for **Xamarin.Essentials** and install the package into **ALL** projects including Android, iOS, UWP, and .NET Standard libraries.

4. Add a reference to Xamarin.Essentials in any C# class to reference the APIs.

   ```
   using Xamarin.Essentials;
   ```

5. Xamarin.Essentials requires platform-specific setup:

   - Android
   - iOS
   - UWP

   Xamarin.Essentials supports a minimum Android version of 4.4, corresponding to API level 19, but the target Android version for compiling must be 9.0 or 10.0, corresponding to API level 28 and level 29. (In Visual Studio, these two versions are set in the Project Properties dialog for the Android project, in the Android Manifest tab. In Visual Studio for Mac, they're set in the Project Options dialog for the Android project, in the Android Application tab.)

   When compiling against Android 9.0, Xamarin.Essentials installs version 28.0.0.3 of the Xamarin.Android.Support libraries that it requires. Any other Xamarin.Android.Support libraries that your application requires should also be updated to version 28.0.0.3 using the NuGet package manager. All Xamarin.Android.Support libraries used by your application should be the same, and should be at least version 28.0.0.3. Refer to the troubleshooting page if you have issues adding the Xamarin.Essentials NuGet or updating NuGets in your solution.

Starting with version 1.5.0 when compiling against Android 10.0, Xamarin.Essentials install AndroidX support libraries that it requires. Read through the AndroidX documentation if you have not made the transition yet.

In the Android project's `MainLauncher` or any `Activity` that is launched, Xamarin.Essentials must be initialized in the `OnCreate` method:

```
protected override void OnCreate(Bundle savedInstanceState) {
    //...
    base.OnCreate(savedInstanceState);
    Xamarin.Essentials.Platform.Init(this, savedInstanceState); // add this line to your code, it may
also be called: bundle
    //...
```

To handle runtime permissions on Android, Xamarin.Essentials must receive any `OnRequestPermissionsResult`. Add the following code to all `Activity` classes:

```
public override void OnRequestPermissionsResult(int requestCode, string[] permissions,
Android.Content.PM.Permission[] grantResults)
{
    Xamarin.Essentials.Platform.OnRequestPermissionsResult(requestCode, permissions, grantResults);

    base.OnRequestPermissionsResult(requestCode, permissions, grantResults);
}
```

6. Follow the Xamarin.Essentials guides that enable you to copy and paste code snippets for each feature.

Xamarin.Essentials - Cross-Platform APIs for Mobile Apps (video)

Other Resources

We recommend developers new to Xamarin visit getting started with Xamarin development.

Visit the Xamarin.Essentials GitHub Repository to see the current source code, what is coming next, run samples, and clone the repository. Community contributions are welcome!

Browse through the API documentation for every feature of Xamarin.Essentials.

Platform Support

7/8/2021 • 2 minutes to read • Edit Online

Xamarin.Essentials supports the following platforms and operating systems:

PLATFORM	VERSION
Android	4.4 (API 19) or higher
iOS	10.0 or higher
Tizen	4.0 or higher
tvOS	10.0 or higher
watchOS	4.0 or higher
UWP	10.0.16299.0 or higher
macOS	10.12.6 (Sierra) or higher

NOTE

- Tizen is officially supported by the Samsung development team.
- tvOS & watchOS have limited API coverage, please see the feature guide for more information.
- macOS support is in preview.

Feature Support

Xamarin.Essentials always tries to bring features to every platform, however sometimes there are limitations based on the device. Below is a guide of what features are supported on each platform.

Icon Guide:

- ✅ - Full support
- ⚠️ - Limited support
- ❌ - Not supported

FEATURE	ANDROID	IOS	UWP	WATCHOS	TVOS	TIZEN	MACOS
Accelerometer	✅	✅	✅	✅	❌	✅	❌
App Actions	✅	✅	✅	❌	❌	❌	❌
App Information	✅	✅	✅	❌	✅	✅	✅

FEATURE	ANDROID	IOS	UWP	WATCHOS	TVOS	TIZEN	MACOS
App Theme	✔	✔	✔	✔	✖	✔	✔
Barometer	✔	✔	✔	✔	✖	✔	✖
Battery	✔	✔	✔	⚠	✖	⚠	✔
Clipboard	✔	✔	✔	✖	✖	✖	✔
Color Converters	✔	✔	✔	✔	✔	✔	✔
Compass	✔	✔	✔	✖	✖	✔	✖
Connectivity	✔	✔	✔	✖	✔	✔	✔
Contacts	✔	✔	✔	✖	✖	✔	✖
Detect Shake	✔	✔	✔	✔	✔	✔	✖
Device Display Information	✔	✔	✔	✖	✖	✖	✔
Device Information	✔	✔	✔	✔	✔	✔	✔
Email	✔	✔	✔	✖	✖	✔	✔
File Picker	✔	✔	✔	✖	✖	✔	✔
File System Helpers	✔	✔	✔	✔	✔	✔	✔
Flashlight	✔	✔	✔	✖	✖	✔	✖
Geocoding	✔	✔	✔	✔	✔	✔	✔
Geolocation	✔	✔	✔	✖	✖	✔	✔
Gyroscope	✔	✔	✔	✔	✖	✔	✖
Haptic Feedback	✔	✔	✔	✖	✖	✔	✔
Launcher	✔	✔	✔	✖	✖	✔	✔
Magnetometer	✔	✔	✔	✔	✖	✔	✖

FEATURE	ANDROID	IOS	UWP	WATCHOS	TVOS	TIZEN	MACOS
MainThread	✔️	✔️	✔️	✔️	✔️	✔️	✔️
Maps	✔️	✔️	✔️	✔️	❌	✔️	✔️
Media Picker	✔️	✔️	✔️	❌	❌	✔️	⚠️
Open Browser	✔️	✔️	✔️	❌	❌	✔️	✔️
Orientation Sensor	✔️	✔️	✔️	✔️	❌	✔️	❌
Permissions	✔️	✔️	✔️	✔️	✔️	✔️	✔️
Phone Dialer	✔️	✔️	✔️	❌	❌	✔️	✔️
Platform Extensions	✔️	✔️	✔️	✔️	✔️	✔️	✔️
Preferences	✔️	✔️	✔️	✔️	✔️	✔️	✔️
Screenshot	✔️	✔️	✔️	❌	❌	❌	❌
Secure Storage	✔️	✔️	✔️	✔️	✔️	✔️	✔️
Share	✔️	✔️	✔️	❌	❌	✔️	✔️
SMS	✔️	✔️	✔️	❌	❌	✔️	✔️
Text-to-Speech	✔️	✔️	✔️	✔️	✔️	✔️	✔️
Unit Converters	✔️	✔️	✔️	✔️	✔️	✔️	✔️
Version Tracking	✔️	✔️	✔️	✔️	✔️	✔️	✔️
Vibrate	✔️	✔️	✔️	❌	❌	✔️	❌
Web Authenticator	✔️	✔️	✔️	❌	✔️	❌	✔️

Xamarin.Essentials: Accelerometer

11/2/2020 • 2 minutes to read • Edit Online

The **Accelerometer** class lets you monitor the device's accelerometer sensor, which indicates the acceleration of the device in three-dimensional space.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using Accelerometer

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Accelerometer functionality works by calling the `Start` and `Stop` methods to listen for changes to the acceleration. Any changes are sent back through the `ReadingChanged` event. Here is sample usage:

```
public class AccelerometerTest
{
    // Set speed delay for monitoring changes.
    SensorSpeed speed = SensorSpeed.UI;

    public AccelerometerTest()
    {
        // Register for reading changes, be sure to unsubscribe when finished
        Accelerometer.ReadingChanged += Accelerometer_ReadingChanged;
    }

    void Accelerometer_ReadingChanged(object sender, AccelerometerChangedEventArgs e)
    {
        var data = e.Reading;
        Console.WriteLine($"Reading: X: {data.Acceleration.X}, Y: {data.Acceleration.Y}, Z:
{data.Acceleration.Z}");
        // Process Acceleration X, Y, and Z
    }

    public void ToggleAccelerometer()
    {
        try
        {
            if (Accelerometer.IsMonitoring)
              Accelerometer.Stop();
            else
              Accelerometer.Start(speed);
        }
        catch (FeatureNotSupportedException fnsEx)
        {
            // Feature not supported on device
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}
```

Accelerometer readings are reported back in G. A G is a unit of gravitation force equal to that exerted by the earth's gravitational field (9.81 m/s^2).

The coordinate-system is defined relative to the screen of the phone in its default orientation. The axes are not swapped when the device's screen orientation changes.

The X axis is horizontal and points to the right, the Y axis is vertical and points up and the Z axis points towards the outside of the front face of the screen. In this system, coordinates behind the screen have negative Z values.

Examples:

- When the device lies flat on a table and is pushed on its left side toward the right, the x acceleration value is positive.
- When the device lies flat on a table, the acceleration value is +1.00 G or (+9.81 m/s^2), which correspond to the acceleration of the device (0 m/s^2) minus the force of gravity (-9.81 m/s^2) and normalized as in G.
- When the device lies flat on a table and is pushed toward the sky with an acceleration of A m/s^2, the acceleration value is equal to A+9.81 which corresponds to the acceleration of the device (+A m/s^2) minus the force of gravity (-9.81 m/s^2) and normalized in G.

Sensor Speed

- **Fastest** – Get the sensor data as fast as possible (not guaranteed to return on UI thread).
- **Game** – Rate suitable for games (not guaranteed to return on UI thread).
- **Default** – Default rate suitable for screen orientation changes.
- **UI** – Rate suitable for general user interface.

If your event handler is not guaranteed to run on the UI thread, and if the event handler needs to access user-interface elements, use the `MainThread.BeginInvokeOnMainThread` method to run that code on the UI thread.

API

- Accelerometer source code
- Accelerometer API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: App Actions

7/8/2021 • 2 minutes to read • Edit Online

The **AppActions** class lets you create and respond to app shortcuts from the app icon.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **AppActions** functionality the following platform specific setup is required.

- Android
- iOS
- UWP

Add the intent filter to your `MainActivity` class:

```
[IntentFilter(
        new[] { Xamarin.Essentials.Platform.Intent.ActionAppAction },
        Categories = new[] { Android.Content.Intent.CategoryDefault })]
public class MainActivity : global::Xamarin.Forms.Platform.Android.FormsAppCompatActivity
{
    ...
```

Then add the following logic to handle actions:

```
protected override void OnResume()
{
    base.OnResume();

    Xamarin.Essentials.Platform.OnResume(this);
}

protected override void OnNewIntent(Android.Content.Intent intent)
{
    base.OnNewIntent(intent);

    Xamarin.Essentials.Platform.OnNewIntent(intent);
}
```

Create Actions

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

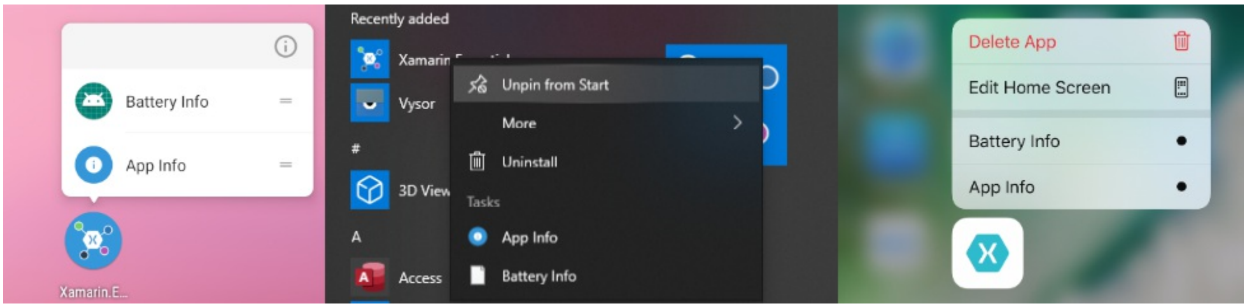
App Actions can be created at any time, but are often created when an application starts. Call the `SetAsync` method to create the list of actions for your app.

```
try
{
    await AppActions.SetAsync(
        new AppAction("app_info", "App Info", icon: "app_info_action_icon"),
        new AppAction("battery_info", "Battery Info"));
}
catch (FeatureNotSupportedException ex)
{
    Debug.WriteLine("App Actions not supported");
}
```

If App Actions are not supported on the specific version of the operating system a `FeatureNotSupportedException` will be thrown.

The following properties can be set on an `AppAction`:

- Id: A unique identifier used to respond to the action tap.
- Title: the visible title to display.
- Subtitle: If supported a sub-title to display under the title.
- Icon: Must match icons in the corresponding resources directory on each platform.



Responding To Actions

When your application starts register for the `OnAppAction` event. When an app action is selected the event will be sent with information as to which action was selected.

```
public App()
{
    //...
    AppActions.OnAppAction += AppActions_OnAppAction;
}

void AppActions_OnAppAction(object sender, AppActionEventArgs e)
{
    // Don't handle events fired for old application instances
    // and cleanup the old instance's event handler
    if (Application.Current != this && Application.Current is App app)
    {
        AppActions.OnAppAction -= app.AppActions_OnAppAction;
        return;
    }
    MainThread.BeginInvokeOnMainThread(async () =>
    {
        await Shell.Current.GoToAsync($"//{e.AppAction.Id}");
    });
}
```

GetActions

You can get the current list of App Actions by calling `AppActions.GetAsync()`.

API

- AppActions source code
- AppActions API documentation

Related Video

Xamarin.Essentials: App Information

11/2/2020 • 2 minutes to read • Edit Online

The **AppInfo** class provides information about your application.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using AppInfo

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

Obtaining Application Information:

The following information is exposed through the API:

```
// Application Name
var appName = AppInfo.Name;

// Package Name/Application Identifier (com.microsoft.testapp)
var packageName = AppInfo.PackageName;

// Application Version (1.0.0)
var version = AppInfo.VersionString;

// Application Build Number (1)
var build = AppInfo.BuildString;
```

Displaying Application Settings

The **AppInfo** class can also display a page of settings maintained by the operating system for the application:

```
// Display settings page
AppInfo.ShowSettingsUI();
```

This settings page allows the user to change application permissions and perform other platform-specific tasks.

Platform Implementation Specifics

- Android
- iOS
- UWP

App information is taken from the `AndroidManifest.xml` for the following fields:

- **Build** – `android:versionCode` in `manifest` node

- **Name** - `android:label` in the `application` node
- **PackageName**: `package` in the `manifest` node
- **VersionString** – `android:versionName` in the `application` node

API

- AppInfo source code
- AppInfo API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: App Theme

11/2/2020 • 2 minutes to read • Edit Online

The **RequestedTheme** API is part of the `AppInfo` class and provides information as to what theme is requested for your running app by the system.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using RequestedTheme

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

Obtaining Theme Information

The requested application theme can be detected with the following API:

```
AppTheme appTheme = AppInfo.RequestedTheme;
```

This will provide the current requested theme by the system for your application. The return value will be one of the following:

- Unspecified
- Light
- Dark

Unspecified will be returned when the operating system does not have a specific user interface style to request. An example of this is on devices running versions of iOS older than 13.0.

Platform Implementation Specifics

- Android
- iOS
- UWP

Android uses configuration modes to specify the type of theme to request from the user. Based on the version of Android, it can be changed by the user or is changed when battery saver mode is enabled.

You can read more on the official Android documentation for Dark Theme.

API

- AppInfo source code
- AppInfo API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Barometer

11/2/2020 • 2 minutes to read • Edit Online

The **Barometer** class lets you monitor the device's barometer sensor, which measures pressure.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using Barometer

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Barometer functionality works by calling the `Start` and `Stop` methods to listen for changes to the barometer's pressure reading in hectopascals. Any changes are sent back through the `ReadingChanged` event. Here is sample usage:

```
public class BarometerTest
{
    // Set speed delay for monitoring changes.
    SensorSpeed speed = SensorSpeed.UI;

    public BarometerTest()
    {
        // Register for reading changes.
        Barometer.ReadingChanged += Barometer_ReadingChanged;
    }

    void Barometer_ReadingChanged(object sender, BarometerChangedEventArgs e)
    {
        var data = e.Reading;
        // Process Pressure
        Console.WriteLine($"Reading: Pressure: {data.PressureInHectopascals} hectopascals");
    }

    public void ToggleBarometer()
    {
        try
        {
            if (Barometer.IsMonitoring)
              Barometer.Stop();
            else
              Barometer.Start(speed);
        }
        catch (FeatureNotSupportedException fnsEx)
        {
            // Feature not supported on device
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}
```

Sensor Speed

- **Fastest** – Get the sensor data as fast as possible (not guaranteed to return on UI thread).
- **Game** – Rate suitable for games (not guaranteed to return on UI thread).
- **Default** – Default rate suitable for screen orientation changes.
- **UI** – Rate suitable for general user interface.

If your event handler is not guaranteed to run on the UI thread, and if the event handler needs to access user-interface elements, use the `MainThread.BeginInvokeOnMainThread` method to run that code on the UI thread.

Platform Implementation Specifics

- Android
- iOS
- UWP

No platform-specific implementation details.

API

- Barometer source code

- Barometer API documentation

Xamarin.Essentials: Battery

4/27/2021 • 3 minutes to read • Edit Online

The **Battery** class lets you check the device's battery information and monitor for changes and provides information about the device's energy-saver status, which indicates if the device is running in a low-power mode. Applications should avoid background processing if the device's energy-saver status is on.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **Battery** functionality the following platform specific setup is required.

- Android
- iOS
- UWP

The `Battery` permission is required and must be configured in the Android project. This can be added in the following ways:

Open the **AssemblyInfo.cs** file under the **Properties** folder and add:

```
[assembly: UsesPermission(Android.Manifest.Permission.BatteryStats)]
```

OR Update Android Manifest:

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node.

```
<uses-permission android:name="android.permission.BATTERY_STATS" />
```

Or right click on the Android project and open the project's properties. Under **Android Manifest** find the **Required permissions:** area and check the **Battery** permission. This will automatically update the **AndroidManifest.xml** file.

Using Battery

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

Check current battery information:

```
var level = Battery.ChargeLevel; // returns 0.0 to 1.0 or 1.0 when on AC or no battery.

var state = Battery.State;

switch (state)
{
    case BatteryState.Charging:
        // Currently charging
        break;
    case BatteryState.Full:
        // Battery is full
        break;
    case BatteryState.Discharging:
    case BatteryState.NotCharging:
        // Currently discharging battery or not being charged
        break;
    case BatteryState.NotPresent:
        // Battery doesn't exist in device (desktop computer)
        break;
    case BatteryState.Unknown:
        // Unable to detect battery state
        break;
}

var source = Battery.PowerSource;

switch (source)
{
    case BatteryPowerSource.Battery:
        // Being powered by the battery
        break;
    case BatteryPowerSource.AC:
        // Being powered by A/C unit
        break;
    case BatteryPowerSource.Usb:
        // Being powered by USB cable
        break;
    case BatteryPowerSource.Wireless:
        // Powered via wireless charging
        break;
    case BatteryPowerSource.Unknown:
        // Unable to detect power source
        break;
}
```

Whenever any of the battery's properties change an event is triggered:

```
public class BatteryTest
{
    public BatteryTest()
    {
        // Register for battery changes, be sure to unsubscribe when needed
        Battery.BatteryInfoChanged += Battery_BatteryInfoChanged;
    }

    void Battery_BatteryInfoChanged(object sender, BatteryInfoChangedEventArgs   e)
    {
        var level = e.ChargeLevel;
        var state = e.State;
        var source = e.PowerSource;
        Console.WriteLine($"Reading: Level: {level}, State: {state}, Source: {source}");
    }
}
```

Devices that run on batteries can be put into a low-power energy-saver mode. Sometimes devices are switched into this mode automatically, for example, when the battery drops below 20% capacity. The operating system responds to energy-saver mode by reducing activities that tend to deplete the battery. Applications can help by avoiding background processing or other high-power activities when energy-saver mode is on.

You can also obtain the current energy-saver status of the device using the static `Battery.EnergySaverStatus` property:

```
// Get energy saver status
var status = Battery.EnergySaverStatus;
```

This property returns a member of the `EnergySaverStatus` enumeration, which is either `On`, `Off`, or `Unknown`. If the property returns `On`, the application should avoid background processing or other activities that might consume a lot of power.

The application should also install an event handler. The **Battery** class exposes an event that is triggered when the energy-saver status changes:

```
public class EnergySaverTest
{
    public EnergySaverTest()
    {
        // Subscribe to changes of energy-saver status
        Battery.EnergySaverStatusChanged += OnEnergySaverStatusChanged;
    }

    private void OnEnergySaverStatusChanged(EnergySaverStatusChangedEventArgs e)
    {
        // Process change
        var status = e.EnergySaverStatus;
    }
}
```

If the energy-saver status changes to `On`, the application should stop performing background processing. If the status changes to `Unknown` or `Off`, the application can resume background processing.

Platform Differences

- Android
- iOS
- UWP

No platform differences.

API

- Battery source code
- Battery API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Clipboard

11/2/2020 • 2 minutes to read • Edit Online

The **Clipboard** class lets you copy and paste text to the system clipboard between applications.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using Clipboard

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

To check if the **Clipboard** has text currently ready to be pasted:

```
var hasText = Clipboard.HasText;
```

To set text to the **Clipboard**:

```
await Clipboard.SetTextAsync("Hello World");
```

To read text from the **Clipboard**:

```
var text = await Clipboard.GetTextAsync();
```

Whenever any of the clipboard's content has changed an event is triggered:

```
public class ClipboardTest
{
    public ClipboardTest()
    {
        // Register for clipboard changes, be sure to unsubscribe when needed
        Clipboard.ClipboardContentChanged += OnClipboardContentChanged;
    }

    void OnClipboardContentChanged(object sender, EventArgs    e)
    {
        Console.WriteLine($"Last clipboard change at {DateTime.UtcNow:T}";);
    }
}
```

TIP

Access to the Clipboard must be done on the main user interface thread. See the MainThread API to see how to invoke methods on the main user interface thread.

API

- Clipboard source code
- Clipboard API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Color Converters

11/2/2020 • 2 minutes to read • Edit Online

The **ColorConverters** class in Xamarin.Essentials provides several helper methods for System.Drawing.Color.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using Color Converters

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

When working with `System.Drawing.Color` you can use the built in converters of Xamarin.Forms to create a color from Hsl, Hex, or UInt.

```
var blueHex = ColorConverters.FromHex("#3498db");
var blueHsl = ColorConverters.FromHsl(204, 70, 53);
var blueUInt = ColorConverters.FromUInt(3447003);
```

Using Color Extensions

Extension methods on `System.Drawing.Color` enable you to apply different properties:

```
var blue = ColorConverters.FromHex("#3498db");

// Multiplies the current alpha by 50%
var blueWithAlpha = blue.MultiplyAlpha(.5f);
```

There are several other extension methods including:

- GetComplementary
- MultiplyAlpha
- ToUInt
- WithAlpha
- WithHue
- WithLuminosity
- WithSaturation

Using Platform Extensions

Additionally, you can convert System.Drawing.Color to the platform specific color structure. These methods can only be called from the iOS, Android, and UWP projects.

```
var system = System.Drawing.Color.FromArgb(255, 52, 152, 219);

// Extension to convert to Android.Graphics.Color, UIKit.UIColor, or Windows.UI.Color
var platform = system.ToPlatformColor();
```

```
var platform = new Android.Graphics.Color(52, 152, 219, 255);

// Back to System.Drawing.Color
var system = platform.ToSystemColor();
```

The `ToSystemColor` method applies to Android.Graphics.Color, UIKit.UIColor, and Windows.UI.Color.

API

- Color Converters source code
- Color Converters API documentation
- Color Extensions source code
- Color Extensions API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Compass

11/2/2020 • 2 minutes to read • Edit Online

The **Compass** class lets you monitor the device's magnetic north heading.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using Compass

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Compass functionality works by calling the `Start` and `Stop` methods to listen for changes to the compass. Any changes are sent back through the `ReadingChanged` event. Here is an example:

```
public class CompassTest
{
    // Set speed delay for monitoring changes.
    SensorSpeed speed = SensorSpeed.UI;

    public CompassTest()
    {
        // Register for reading changes, be sure to unsubscribe when finished
        Compass.ReadingChanged += Compass_ReadingChanged;
    }

    void Compass_ReadingChanged(object sender, CompassChangedEventArgs e)
    {
        var data = e.Reading;
        Console.WriteLine($"Reading: {data.HeadingMagneticNorth} degrees");
        // Process Heading Magnetic North
    }

    public void ToggleCompass()
    {
        try
        {
            if (Compass.IsMonitoring)
              Compass.Stop();
            else
              Compass.Start(speed);
        }
        catch (FeatureNotSupportedException fnsEx)
        {
            // Feature not supported on device
        }
        catch (Exception ex)
        {
            // Some other exception has occurred
        }
    }
}
```

Sensor Speed

- **Fastest** – Get the sensor data as fast as possible (not guaranteed to return on UI thread).
- **Game** – Rate suitable for games (not guaranteed to return on UI thread).
- **Default** – Default rate suitable for screen orientation changes.
- **UI** – Rate suitable for general user interface.

If your event handler is not guaranteed to run on the UI thread, and if the event handler needs to access user-interface elements, use the `MainThread.BeginInvokeOnMainThread` method to run that code on the UI thread.

Platform Implementation Specifics

- Android

Android does not provide an API for retrieving the compass heading. We utilize the accelerometer and magnetometer to calculate the magnetic north heading, which is recommended by Google.

In rare instances, you maybe see inconsistent results because the sensors need to be calibrated, which involves moving your device in a figure-8 motion. The best way of doing this is to open Google Maps, tap on the dot for your location, and select **Calibrate compass**.

Running multiple sensors from your app at the same time may adjust the sensor speed.

Low Pass Filter

Due to how the Android compass values are updated and calculated there may be a need to smooth out the values. A *Low Pass Filter* can be applied that averages the sine and cosine values of the angles and can be turned on by using the `Start` method overload, which accepts the `bool applyLowPassFilter` parameter:

```
Compass.Start(SensorSpeed.UI, applyLowPassFilter: true);
```

This is only applied on the Android platform, and the parameter is ignored on iOS and UWP. More information can be read here.

API

- Compass source code
- Compass API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Connectivity

11/2/2020 • 2 minutes to read • Edit Online

The **Connectivity** class lets you monitor for changes in the device's network conditions, check the current network access, and how it is currently connected.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **Connectivity** functionality the following platform specific setup is required.

- Android
- iOS
- UWP

The `AccessNetworkState` permission is required and must be configured in the Android project. This can be added in the following ways:

Open the **AssemblyInfo.cs** file under the **Properties** folder and add:

```
[assembly: UsesPermission(Android.Manifest.Permission.AccessNetworkState)]
```

OR Update Android Manifest:

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node.

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Or right click on the Android project and open the project's properties. Under **Android Manifest** find the **Required permissions:** area and check the **Access Network State** permission. This will automatically update the **AndroidManifest.xml** file.

Using Connectivity

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

Check current network access:

```
var current = Connectivity.NetworkAccess;

if (current == NetworkAccess.Internet)
{
    // Connection to internet is available
}
```

Network access falls into the following categories:

- **Internet** – Local and internet access.
- **ConstrainedInternet** – Limited internet access. Indicates captive portal connectivity, where local access to a web portal is provided, but access to the Internet requires that specific credentials are provided via a portal.
- **Local** – Local network access only.
- **None** – No connectivity is available.
- **Unknown** – Unable to determine internet connectivity.

You can check what type of connection profile the device is actively using:

```
var profiles = Connectivity.ConnectionProfiles;
if (profiles.Contains(ConnectionProfile.WiFi))
{
    // Active Wi-Fi connection.
}
```

Whenever the connection profile or network access changes you can receive an event when triggered:

```
public class ConnectivityTest
{
    public ConnectivityTest()
    {
        // Register for connectivity changes, be sure to unsubscribe when finished
        Connectivity.ConnectivityChanged += Connectivity_ConnectivityChanged;
    }

    void Connectivity_ConnectivityChanged(object sender, ConnectivityChangedEventArgs e)
    {
        var access = e.NetworkAccess;
        var profiles = e.ConnectionProfiles;
    }
}
```

Limitations

It is important to note that it is possible that `Internet` is reported by `NetworkAccess` but full access to the web is not available. Due to how connectivity works on each platform it can only guarantee that a connection is available. For instance the device may be connected to a Wi-Fi network, but the router is disconnected from the internet. In this instance Internet may be reported, but an active connection is not available.

API

- Connectivity source code
- Connectivity API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Contacts

3/5/2021 • 2 minutes to read • Edit Online

The **Contacts** class lets a user pick a contact and retrieve information about it.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **Contacts** functionality the following platform specific setup is required.

- Android
- iOS
- UWP

The `ReadContacts` permission is required and must be configured in the Android project. This can be added in the following ways:

Open the **AssemblyInfo.cs** file under the **Properties** folder and add:

```
[assembly: UsesPermission(Android.Manifest.Permission.ReadContacts)]
```

OR Update Android Manifest:

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node.

```
<uses-permission android:name="android.permission.READ_CONTACTS" /> />
```

Or right click on the Android project and open the project's properties. Under **Android Manifest** find the **Required permissions:** area and check this permission. This will automatically update the **AndroidManifest.xml** file.

Pick a contact

By calling `Contacts.PickContactAsync()` the contact dialog will appear and allow the user to receive information about the user.

```
try
{
    var contact = await Contacts.PickContactAsync();

    if(contact == null)
        return;

    var id = contact.Id;
    var namePrefix = contact.NamePrefix;
    var givenName = contact.GivenName;
    var middleName = contact.MiddleName;
    var familyName = contact.FamilyName;
    var nameSuffix = contact.NameSuffix;
    var displayName = contact.DisplayName;
    var phones = contact.Phones; // List of phone numbers
    var emails = contact.Emails; // List of email addresses
}
catch (Exception ex)
{
    // Handle exception here.
}
```

Get all contacts

```
ObservableCollection<Contact> contactsCollect = new ObservableCollection<Contact>();

try
{
    // cancellationToken parameter is optional
    var cancellationToken = default(CancellationToken);
    var contacts = await Contacts.GetAllAsync(cancellationToken);

    if (contacts == null)
        return;

    foreach (var contact in contacts)
        contactsCollect.Add(contact);
}
catch (Exception ex)
{
    // Handle exception here.
}
```

Platform differences

- Android
- iOS
- UWP

- The `cancellationToken` parameter in the `GetAllAsync` method is only used on UWP.

API

- Contacts source code
- Contacts API documentation

Xamarin.Essentials: Detect Shake

11/2/2020 • 2 minutes to read • Edit Online

The **Accelerometer** class lets you monitor the device's accelerometer sensor, which indicates the acceleration of the device in three-dimensional space. Additionally, it enables you to register for events when the user shakes the device.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using Detect Shake

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

To detect a shake of the device you must use the Accelerometer functionality by calling the `Start` and `Stop` methods to listen for changes to the acceleration and to detect a shake. Any time a shake is detected a `ShakeDetected` event will fire. It is recommended to use `Game` or faster for the `SensorSpeed`. Here is sample usage:

```csharp
public class DetectShakeTest
{
    // Set speed delay for monitoring changes.
    SensorSpeed speed = SensorSpeed.Game;

    public DetectShakeTest()
    {
        // Register for reading changes, be sure to unsubscribe when finished
        Accelerometer.ShakeDetected  += Accelerometer_ShakeDetected ;
    }

    void Accelerometer_ShakeDetected (object sender, EventArgs e)
    {
        // Process shake event
    }

    public void ToggleAccelerometer()
    {
        try
        {
            if (Accelerometer.IsMonitoring)
              Accelerometer.Stop();
            else
              Accelerometer.Start(speed);
        }
        catch (FeatureNotSupportedException fnsEx)
        {
            // Feature not supported on device
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}
```

Sensor Speed

- **Fastest** – Get the sensor data as fast as possible (not guaranteed to return on UI thread).
- **Game** – Rate suitable for games (not guaranteed to return on UI thread).
- **Default** – Default rate suitable for screen orientation changes.
- **UI** – Rate suitable for general user interface.

If your event handler is not guaranteed to run on the UI thread, and if the event handler needs to access user-interface elements, use the `MainThread.BeginInvokeOnMainThread` method to run that code on the UI thread.

Implementation Details

The detect shake API uses raw readings from the accelerometer to calculate acceleration. It uses a simple queue mechanism to detect if 3/4ths of the recent accelerometer events occurred in the last half second. Acceleration is calculated by adding the square of the X, Y, and Z readings from the accelerometer and comparing it to a specific threashold.

API

- Accelerometer source code
- Accelerometer API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Device Display Information

3/11/2021 • 2 minutes to read • Edit Online

The **DeviceDisplay** class provides information about the device's screen metrics the application is running on and can request to keep the screen from falling asleep when the application is running.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using DeviceDisplay

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

Main Display Info

In addition to basic device information the **DeviceDisplay** class contains information about the device's screen and orientation.

```
// Get Metrics
var mainDisplayInfo = DeviceDisplay.MainDisplayInfo;

// Orientation (Landscape, Portrait, Square, Unknown)
var orientation = mainDisplayInfo.Orientation;

// Rotation (0, 90, 180, 270)
var rotation = mainDisplayInfo.Rotation;

// Width (in pixels)
var width = mainDisplayInfo.Width;

// Height (in pixels)
var height = mainDisplayInfo.Height;

// Screen density
var density = mainDisplayInfo.Density;
```

The **DeviceDisplay** class also exposes an event that can be subscribed to that is triggered whenever any screen metric changes:

```
public class DisplayInfoTest
{
    public DisplayInfoTest()
    {
        // Subscribe to changes of screen metrics
        DeviceDisplay.MainDisplayInfoChanged += OnMainDisplayInfoChanged;
    }

    void OnMainDisplayInfoChanged(object sender, DisplayInfoChangedEventArgs  e)
    {
        // Process changes
        var displayInfo = e.DisplayInfo;
    }
}
```

Keep Screen On

The **DeviceDisplay** class exposes a `bool` property called `KeepScreenOn` that can be set to attempt to keep the device's display from turning off or locking.

```
public class KeepScreenOnTest
{
    public void ToggleScreenLock()
    {
        DeviceDisplay.KeepScreenOn = !DeviceDisplay.KeepScreenOn;
    }
}
```

Platform Differences

- Android
- iOS
- UWP

No differences.

API

- DeviceDisplay source code
- DeviceDisplay API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Device Information

11/2/2020 • 2 minutes to read • Edit Online

The **DeviceInfo** class provides information about the device the application is running on.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using DeviceInfo

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The following information is exposed through the API:

```
// Device Model (SMG-950U, iPhone10,6)
var device = DeviceInfo.Model;

// Manufacturer (Samsung)
var manufacturer = DeviceInfo.Manufacturer;

// Device Name (Motz's iPhone)
var deviceName = DeviceInfo.Name;

// Operating System Version Number (7.0)
var version = DeviceInfo.VersionString;

// Platform (Android)
var platform = DeviceInfo.Platform;

// Idiom (Phone)
var idiom = DeviceInfo.Idiom;

// Device Type (Physical)
var deviceType = DeviceInfo.DeviceType;
```

Platforms

`DeviceInfo.Platform` correlates to a constant string that maps to the operating system. The values can be checked with the `DevicePlatform` struct:

- **DevicePlatform.iOS** – iOS
- **DevicePlatform.Android** – Android
- **DevicePlatform.UWP** – UWP
- **DevicePlatform.Unknown** – Unknown

Idioms

`DeviceInfo.Idiom` correlates a constant string that maps to the type of device the application is running on. The values can be checked with the `DeviceIdiom` struct:

- **DeviceIdiom.Phone** – Phone
- **DeviceIdiom.Tablet** – Tablet
- **DeviceIdiom.Desktop** – Desktop
- **DeviceIdiom.TV** – TV
- **DeviceIdiom.Watch** – Watch
- **DeviceIdiom.Unknown** – Unknown

Device Type

`DeviceInfo.DeviceType` correlates an enumeration to determine if the application is running on a physical or virtual device. A virtual device is a simulator or emulator.

Platform Implementation Specifics

- iOS

iOS does not expose an API for developers to get the model of the specific iOS device. Instead a hardware identifier is returned such as *iPhone10,6* which refers to the iPhone X. A mapping of these identifiers are not provided by Apple, but can be found on these (non-official sources) The iPhone Wiki and Get iOS Model.

API

- DeviceInfo source code
- DeviceInfo API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Email

11/2/2020 • 2 minutes to read • Edit Online

The **Email** class enables an application to open the default email application with a specified information including subject, body, and recipients (TO, CC, BCC).

To access the **Email** functionality the following platform specific setup is required.

- Android
- iOS
- UWP

If your project's Target Android version is set to **Android 11 (R API 30)** you must update your Android Manifest with queries that are used with the new package visibility requirements.

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node:

```
<queries>
  <intent>
    <action android:name="android.intent.action.SENDTO" />
    <data android:scheme="mailto" />
  </intent>
</queries>
```

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

> **TIP**
> To use the Email API on iOS you must run it on a physical device, else an exception will be thrown.

Using Email

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Email functionality works by calling the `ComposeAsync` method an `EmailMessage` that contains information about the email:

```
public class EmailTest
{
    public async Task SendEmail(string subject, string body, List<string> recipients)
    {
        try
        {
            var message = new EmailMessage
            {
                Subject = subject,
                Body = body,
                To = recipients,
                //Cc = ccRecipients,
                //Bcc = bccRecipients
            };
            await Email.ComposeAsync(message);
        }
        catch (FeatureNotSupportedException fbsEx)
        {
            // Email is not supported on this device
        }
        catch (Exception ex)
        {
            // Some other exception occurred
        }
    }
}
```

File Attachments

This feature enables an app to email files in email clients on the device. Xamarin.Essentials will automatically detect the file type (MIME) and request the file to be added as an attachment. Every email client is different and may only support specific file extensions, or none at all.

Here is a sample of writing text to disk and adding it as an email attachment:

```
var message = new EmailMessage
{
    Subject = "Hello",
    Body = "World",
};

var fn = "Attachment.txt";
var file = Path.Combine(FileSystem.CacheDirectory, fn);
File.WriteAllText(file, "Hello World");

message.Attachments.Add(new EmailAttachment(file));

await Email.ComposeAsync(message);
```

Platform Differences

- Android
- iOS
- UWP

Not all email clients for Android support `Html`, since there is no way to detect this we recommend using `PlainText` when sending emails.

API

- Email source code
- Email API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: File Picker

5/13/2021 • 2 minutes to read • Edit Online

The **FilePicker** class lets a user pick a single or multiple files from the device.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **FilePicker** functionality the following platform specific setup is required.

- Android
- iOS
- UWP

The `ReadExternalStorage` permission is required and must be configured in the Android project. This can be added in the following ways:

Open the **AssemblyInfo.cs** file under the **Properties** folder and add:

```
[assembly: UsesPermission(Android.Manifest.Permission.ReadExternalStorage)]
```

OR Update Android Manifest:

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node.

```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

Or right click on the Android project and open the project's properties. Under **Android Manifest** find the **Required permissions:** area and check this permission. This will automatically update the **AndroidManifest.xml** file.

> **TIP**
>
> All methods must be called on the UI thread because permission checks and requests are automatically handled by Xamarin.Essentials.

Pick File

`FilePicker.PickAsync()` method enables your user to pick a file from the device. You are able to specific different `PickOptions` when calling the method enabling you to specify the title to display and the file types the user is allowed to pick. By default

```
async Task<FileResult> PickAndShow(PickOptions options)
{
    try
    {
        var result = await FilePicker.PickAsync(options);
        if (result != null)
        {
            Text = $"File Name: {result.FileName}";
            if (result.FileName.EndsWith("jpg", StringComparison.OrdinalIgnoreCase) ||
                result.FileName.EndsWith("png", StringComparison.OrdinalIgnoreCase))
            {
                var stream = await result.OpenReadAsync();
                Image = ImageSource.FromStream(() => stream);
            }
        }

        return result;
    }
    catch (Exception ex)
    {
        // The user canceled or something went wrong
    }

    return null;
}
```

Default file types are provided with `FilePickerFileType.Images`, `FilePickerFileType.Png`, and `FilePickerFilerType.Videos`. You can specify custom files types when creating the `PickOptions` and they can be customized per platform. For example here is how you would specify specific comic file types:

```
var customFileType =
    new FilePickerFileType(new Dictionary<DevicePlatform, IEnumerable<string>>
    {
        { DevicePlatform.iOS, new[] { "public.my.comic.extension" } }, // or general UTType values
        { DevicePlatform.Android, new[] { "application/comics" } },
        { DevicePlatform.UWP, new[] { ".cbr", ".cbz" } },
        { DevicePlatform.Tizen, new[] { "*/*" } },
        { DevicePlatform.macOS, new[] { "cbr", "cbz" } }, // or general UTType values
    });
var options = new PickOptions
{
    PickerTitle = "Please select a comic file",
    FileTypes = customFileType,
};
```

Pick Multiple Files

If you desire your user to pick multiple files you can call the `FilePicker.PickMultipleAsync()` method. It also takes in `PickOptions` as a parameter to specify additional information. The results are the same as `PickAsync`, but instead of a single `FileResult` an `IEnumerable<FileResult>` is returned that can be iterated over.

TIP

The `FullPath` property does not always return the physical path to the file. To get the file, use the `OpenReadAsync` method.

Platform Differences

- Android

- iOS
- UWP

- No platform differences.

API

- FilePicker source code
- FilePicker API documentation

Related Video

Xamarin.Essentials: File System Helpers

11/2/2020 • 2 minutes to read • Edit Online

The **FileSystem** class contains a series of helpers to find the application's cache and data directories and open files inside of the app package.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using File System Helpers

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

To get the application's directory to store **cache data**. Cache data can be used for any data that needs to persist longer than temporary data, but should not be data that is required to properly operate, as the OS dictates when this storage is cleared.

```
var cacheDir = FileSystem.CacheDirectory;
```

To get the application's top-level directory for any files that are not user data files. These files are backed up with the operating system syncing framework. See Platform Implementation Specifics below.

```
var mainDir = FileSystem.AppDataDirectory;
```

To open a file that is bundled into the application package:

```
using (var stream = await FileSystem.OpenAppPackageFileAsync(templateFileName))
{
    using (var reader = new StreamReader(stream))
    {
        var fileContents = await reader.ReadToEndAsync();
    }
}
```

Platform Implementation Specifics

- Android
- iOS
- UWP

- **CacheDirectory** – Returns the CacheDir of the current context.
- **AppDataDirectory** – Returns the FilesDir of the current context and are backed up using Auto Backup starting on API 23 and above.

Add any file into the **Assets** folder in the Android project and mark the Build Action as **AndroidAsset** to use it with `OpenAppPackageFileAsync`.

API

- File System Helpers source code
- File System API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Flashlight

11/2/2020 • 2 minutes to read • Edit Online

The **Flashlight** class has the ability to turn on or off the device's camera flash to turn it into a flashlight.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **Flashlight** functionality the following platform specific setup is required.

- Android
- iOS
- UWP

The Flashlight and Camera permissions are required and must be configured in the Android project. This can be added in the following ways:

Open the **AssemblyInfo.cs** file under the **Properties** folder and add:

```
[assembly: UsesPermission(Android.Manifest.Permission.Flashlight)]
[assembly: UsesPermission(Android.Manifest.Permission.Camera)]
```

OR Update Android Manifest:

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node.

```
<uses-permission android:name="android.permission.FLASHLIGHT" />
<uses-permission android:name="android.permission.CAMERA" />
```

Or right click on the Android project and open the project's properties. Under **Android Manifest** find the **Required permissions:** area and check the **FLASHLIGHT** and **CAMERA** permissions. This will automatically update the **AndroidManifest.xml** file.

By adding these permissions Google Play will automatically filter out devices without specific hardware. You can get around this by adding the following to your AssemblyInfo.cs file in your Android project:

```
[assembly: UsesFeature("android.hardware.camera", Required = false)]
[assembly: UsesFeature("android.hardware.camera.autofocus", Required = false)]
```

This API uses runtime permissions on Android. Please ensure that Xamarin.Essentials is fully initialized and permission handling is setup in your app.

In the Android project's `MainLauncher` or any `Activity` that is launched Xamarin.Essentials must be initialized in the `OnCreate` method:

```
protected override void OnCreate(Bundle savedInstanceState)
{
    //...
    base.OnCreate(savedInstanceState);
    Xamarin.Essentials.Platform.Init(this, savedInstanceState); // add this line to your code, it may also
be called: bundle
    //...
}
```

To handle runtime permissions on Android, Xamarin.Essentials must receive any `OnRequestPermissionsResult`. Add the following code to all `Activity` classes:

```
public override void OnRequestPermissionsResult(int requestCode, string[] permissions,
Android.Content.PM.Permission[] grantResults)
{
    Xamarin.Essentials.Platform.OnRequestPermissionsResult(requestCode, permissions, grantResults);

    base.OnRequestPermissionsResult(requestCode, permissions, grantResults);
}
```

Using Flashlight

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The flashlight can be turned on and off through the `TurnOnAsync` and `TurnOffAsync` methods:

```
try
{
    // Turn On
    await Flashlight.TurnOnAsync();

    // Turn Off
    await Flashlight.TurnOffAsync();
}
catch (FeatureNotSupportedException fnsEx)
{
    // Handle not supported on device exception
}
catch (PermissionException pEx)
{
    // Handle permission exception
}
catch (Exception ex)
{
    // Unable to turn on/off flashlight
}
```

Platform Implementation Specifics

- Android
- iOS
- UWP

The Flashlight class has been optimized based on the device's operating system.

API Level 23 and Higher

On newer API levels, Torch Mode will be used to turn on or off the flash unit of the device.

API Level 22 and Lower

A camera surface texture is created to turn on or off the `FlashMode` of the camera unit.

API

- Flashlight source code
- Flashlight API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Geocoding

11/2/2020 • 2 minutes to read • Edit Online

The **Geocoding** class provides APIs to geocode a placemark to a positional coordinates and reverse geocode coordinates to a placemark.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **Geocoding** functionality the following platform specific setup is required.

- Android
- iOS
- UWP

No additional setup required.

Using Geocoding

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

Getting location coordinates for an address:

```
try
{
    var address =  "Microsoft Building 25 Redmond WA USA";
    var locations = await Geocoding.GetLocationsAsync(address);

    var location = locations?.FirstOrDefault();
    if (location != null)
    {
        Console.WriteLine($"Latitude: {location.Latitude}, Longitude: {location.Longitude}, Altitude:
{location.Altitude}");
    }
}
catch (FeatureNotSupportedException fnsEx)
{
    // Feature not supported on device
}
catch (Exception ex)
{
    // Handle exception that may have occurred in geocoding
}
```

The altitude isn't always available. If it is not available, the `Altitude` property might be `null` or the value might be zero. If the altitude is available, the value is in meters above sea level.

Using Reverse Geocoding

Reverse geocoding is the process of getting placemarks for an existing set of coordinates:

```
try
{
    var lat = 47.673988;
    var lon = -122.121513;

    var placemarks = await Geocoding.GetPlacemarksAsync(lat, lon);

    var placemark = placemarks?.FirstOrDefault();
    if (placemark != null)
    {
        var geocodeAddress =
            $"AdminArea:       {placemark.AdminArea}\n" +
            $"CountryCode:     {placemark.CountryCode}\n" +
            $"CountryName:     {placemark.CountryName}\n" +
            $"FeatureName:     {placemark.FeatureName}\n" +
            $"Locality:        {placemark.Locality}\n" +
            $"PostalCode:      {placemark.PostalCode}\n" +
            $"SubAdminArea:    {placemark.SubAdminArea}\n" +
            $"SubLocality:     {placemark.SubLocality}\n" +
            $"SubThoroughfare: {placemark.SubThoroughfare}\n" +
            $"Thoroughfare:    {placemark.Thoroughfare}\n";

        Console.WriteLine(geocodeAddress);
    }
}
catch (FeatureNotSupportedException fnsEx)
{
    // Feature not supported on device
}
catch (Exception ex)
{
    // Handle exception that may have occurred in geocoding
}
```

Distance between Two Locations

The `Location` and `LocationExtensions` classes define methods to calculate the distance between two locations. See the article **Xamarin.Essentials: Geolocation** for an example.

API

- Geocoding source code
- Geocoding API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Geolocation

7/28/2021 • 5 minutes to read • Edit Online

The **Geolocation** class provides APIs to retrieve the device's current geolocation coordinates.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **Geolocation** functionality, the following platform-specific setup is required:

- Android
- iOS
- UWP

Coarse and Fine Location permissions are required and must be configured in the Android project. Additionally, if your app targets Android 5.0 (API level 21) or higher, you must declare that your app uses the hardware features in the manifest file. This can be added in the following ways:

Open the **AssemblyInfo.cs** file under the **Properties** folder and add:

```
[assembly: UsesPermission(Android.Manifest.Permission.AccessCoarseLocation)]
[assembly: UsesPermission(Android.Manifest.Permission.AccessFineLocation)]
[assembly: UsesFeature("android.hardware.location", Required = false)]
[assembly: UsesFeature("android.hardware.location.gps", Required = false)]
[assembly: UsesFeature("android.hardware.location.network", Required = false)]
```

Or update the Android manifest:

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node:

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-feature android:name="android.hardware.location" android:required="false" />
<uses-feature android:name="android.hardware.location.gps" android:required="false" />
<uses-feature android:name="android.hardware.location.network" android:required="false" />
```

Or right-click on the Android project and open the project's properties. Under **Android Manifest** find the **Required permissions:** area and check the **ACCESS_COARSE_LOCATION** and **ACCESS_FINE_LOCATION** permissions. This will automatically update the **AndroidManifest.xml** file.

If your application is targeting Android 10 - Q (API Level 29 or higher) and is requesting **LocationAlways**, you must also add the following permission into **AssemblyInfo.cs**:

```
[assembly: UsesPermission(Manifest.Permission.AccessBackgroundLocation)]
```

Or directly into your **AndroidManifest.xml**:

```
<uses-permission android:name="android.permission.ACCESS_BACKGROUND_LOCATION" />
```

If it recommended to read Android documentation on background location updates as there are many restrictions that need to be considered.

This API uses runtime permissions on Android. Please ensure that Xamarin.Essentials is fully initialized and permission handling is setup in your app.

In the Android project's `MainLauncher` or any `Activity` that is launched Xamarin.Essentials must be initialized in the `OnCreate` method:

```
protected override void OnCreate(Bundle savedInstanceState)
{
    //...
    base.OnCreate(savedInstanceState);
    Xamarin.Essentials.Platform.Init(this, savedInstanceState); // add this line to your code, it may also
be called: bundle
    //...
}
```

To handle runtime permissions on Android, Xamarin.Essentials must receive any `OnRequestPermissionsResult`. Add the following code to all `Activity` classes:

```
public override void OnRequestPermissionsResult(int requestCode, string[] permissions,
Android.Content.PM.Permission[] grantResults)
{
    Xamarin.Essentials.Platform.OnRequestPermissionsResult(requestCode, permissions, grantResults);

    base.OnRequestPermissionsResult(requestCode, permissions, grantResults);
}
```

Using Geolocation

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Geolocation API will also prompt the user for permissions when necessary.

You can get the last known location of the device by calling the `GetLastKnownLocationAsync` method. This is often faster then doing a full query, but can be less accurate and may return `null` if no cached location exists.

```
try
{
    var location = await Geolocation.GetLastKnownLocationAsync();

    if (location != null)
    {
        Console.WriteLine($"Latitude: {location.Latitude}, Longitude: {location.Longitude}, Altitude:
{location.Altitude}");
    }
}
catch (FeatureNotSupportedException fnsEx)
{
    // Handle not supported on device exception
}
catch (FeatureNotEnabledException fneEx)
{
    // Handle not enabled on device exception
}
catch (PermissionException pEx)
{
    // Handle permission exception
}
catch (Exception ex)
{
    // Unable to get location
}
```

To query the current device's location coordinates, the `GetLocationAsync` can be used. It is best to pass in a full `GeolocationRequest` and `CancellationToken` since it may take some time to get the device's location.

```
CancellationTokenSource cts;

async Task GetCurrentLocation()
{
    try
    {
        var request = new GeolocationRequest(GeolocationAccuracy.Medium, TimeSpan.FromSeconds(10));
        cts = new CancellationTokenSource();
        var location = await Geolocation.GetLocationAsync(request, cts.Token);

        if (location != null)
        {
            Console.WriteLine($"Latitude: {location.Latitude}, Longitude: {location.Longitude}, Altitude: 
{location.Altitude}");
        }
    }
    catch (FeatureNotSupportedException fnsEx)
    {
        // Handle not supported on device exception
    }
    catch (FeatureNotEnabledException fneEx)
    {
        // Handle not enabled on device exception
    }
    catch (PermissionException pEx)
    {
        // Handle permission exception
    }
    catch (Exception ex)
    {
        // Unable to get location
    }
}

protected override void OnDisappearing()
{
    if (cts != null && !cts.IsCancellationRequested)
        cts.Cancel();
    base.OnDisappearing();
}
```

Note all values may be available due to how each device queries geolocation through different providers. For example, the `Altitude` property might be `null`, have a value of 0, or have a positive value, which is in meters above sea level. Other values that may not be present include `Speed` and `Course`.

Geolocation Accuracy

The following table outlines accuracy per platform:

Lowest

PLATFORM	DISTANCE (IN METERS)
Android	500
iOS	3000
UWP	1000 - 5000

Low

PLATFORM	DISTANCE (IN METERS)
Android	500
iOS	1000
UWP	300 - 3000

Medium (Default)

PLATFORM	DISTANCE (IN METERS)
Android	100 - 500
iOS	100
UWP	30-500

High

PLATFORM	DISTANCE (IN METERS)
Android	0 - 100
iOS	10
UWP	<= 10

Best

PLATFORM	DISTANCE (IN METERS)
Android	0 - 100
iOS	~0
UWP	<= 10

Detecting Mock Locations

Some devices may return a mock location from the provider or by an application that provides mock locations. You can detect this by using the `IsFromMockProvider` on any `Location`.

```
var request = new GeolocationRequest(GeolocationAccuracy.Medium);
var location = await Geolocation.GetLocationAsync(request);

if (location != null)
{
    if(location.IsFromMockProvider)
    {
        // location is from a mock provider
    }
}
```

Distance between Two Locations

The `Location` and `LocationExtensions` classes define `CalculateDistance` methods that allow you to calculate the distance between two geographic locations. This calculated distance does not take roads or other pathways into account, and is merely the shortest distance between the two points along the surface of the Earth, also known as the *great-circle distance* or colloquially, the distance "as the crow flies."

Here's an example:

```
Location boston = new Location(42.358056, -71.063611);
Location sanFrancisco = new Location(37.783333, -122.416667);
double miles = Location.CalculateDistance(boston, sanFrancisco, DistanceUnits.Miles);
```

The `Location` constructor has latitude and longitude arguments in that order. Positive latitude values are north of the equator, and positive longitude values are east of the Prime Meridian. Use the final argument to `CalculateDistance` to specify miles or kilometers. The `UnitConverters` class also defines `KilometersToMiles` and `MilesToKilometers` methods for converting between the two units.

Platform Differences

Altitude is calculated differently on each platform.

- Android
- iOS
- UWP

On Android, altitude, if available, is returned in meters above the WGS 84 reference ellipsoid. If this location does not have an altitude then 0.0 is returned.

API

- Geolocation source code
- Geolocation API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Gyroscope

11/2/2020 • 2 minutes to read • Edit Online

The **Gyroscope** class lets you monitor the device's gyroscope sensor which is the rotation around the device's three primary axes.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using Gyroscope

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Gyroscope functionality works by calling the `Start` and `Stop` methods to listen for changes to the gyroscope. Any changes are sent back through the `ReadingChanged` event in rad/s. Here is sample usage:

```
public class GyroscopeTest
{
    // Set speed delay for monitoring changes.
    SensorSpeed speed = SensorSpeed.UI;

    public GyroscopeTest()
    {
        // Register for reading changes.
        Gyroscope.ReadingChanged += Gyroscope_ReadingChanged;
    }

    void Gyroscope_ReadingChanged(object sender, GyroscopeChangedEventArgs e)
    {
        var data = e.Reading;
        // Process Angular Velocity X, Y, and Z reported in rad/s
        Console.WriteLine($"Reading: X: {data.AngularVelocity.X}, Y: {data.AngularVelocity.Y}, Z:
{data.AngularVelocity.Z}");
    }

    public void ToggleGyroscope()
    {
        try
        {
            if (Gyroscope.IsMonitoring)
              Gyroscope.Stop();
            else
              Gyroscope.Start(speed);
        }
        catch (FeatureNotSupportedException fnsEx)
        {
            // Feature not supported on device
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}
```

Sensor Speed

- **Fastest** – Get the sensor data as fast as possible (not guaranteed to return on UI thread).
- **Game** – Rate suitable for games (not guaranteed to return on UI thread).
- **Default** – Default rate suitable for screen orientation changes.
- **UI** – Rate suitable for general user interface.

If your event handler is not guaranteed to run on the UI thread, and if the event handler needs to access user-interface elements, use the `MainThread.BeginInvokeOnMainThread` method to run that code on the UI thread.

API

- Gyroscope source code
- Gyroscope API documentation

Xamarin.Essentials: Haptic Feedback

3/5/2021 • 2 minutes to read • Edit Online

The **HapticFeedback** class lets you control haptic feedback on device.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **HapticFeedback** functionality the following platform specific setup is required.

- Android
- iOS
- UWP

The Vibrate permission is required and must be configured in the Android project. This can be added in the following ways:

Open the **AssemblyInfo.cs** file under the **Properties** folder and add:

```
[assembly: UsesPermission(Android.Manifest.Permission.Vibrate)]
```

OR Update Android Manifest:

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node.

```
<uses-permission android:name="android.permission.VIBRATE" />
```

Or right click on the Android project and open the project's properties. Under **Android Manifest** find the **Required permissions:** area and check the **VIBRATE** permission. This will automatically update the **AndroidManifest.xml** file.

Using Haptic Feedback

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Haptic Feedback functionality can be performed with a `Click` or `LongPress` feedback type.

```
try
{
    // Perform click feedback
    HapticFeedback.Perform(HapticFeedbackType.Click);

    // Or use long press
    HapticFeedback.Perform(HapticFeedbackType.LongPress);
}
catch (FeatureNotSupportedException ex)
{
    // Feature not supported on device
}
catch (Exception ex)
{
    // Other error has occurred.
}
```

API

- HapticFeedback source code
- HapticFeedback API documentation

Xamarin.Essentials: Launcher

3/5/2021 • 3 minutes to read • Edit Online

The **Launcher** class enables an application to open a URI by the system. This is often used when deep linking into another application's custom URI schemes. If you are looking to open the browser to a website then you should refer to the **Browser** API.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using Launcher

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

To use the Launcher functionality call the `OpenAsync` method and pass in a `string` or `Uri` to open. Optionally, the `CanOpenAsync` method can be used to check if the URI schema can be handled by an application on the device.

```
public class LauncherTest
{
    public async Task OpenRideShareAsync()
    {
        var supportsUri = await Launcher.CanOpenAsync("lyft://");
        if (supportsUri)
            await Launcher.OpenAsync("lyft://ridetype?id=lyft_line");
    }
}
```

This can be combined into a single call with `TryOpenAsync`, which checks if the parameter can be opened and if so open it.

```
public class LauncherTest
{
    public async Task<bool> OpenRideShareAsync()
    {
        return await Launcher.TryOpenAsync("lyft://ridetype?id=lyft_line");
    }
}
```

Additional Platform Setup

- Android
- iOS
- UWP

No additional setup.

Files

This features enables an app to request other apps to open and view a file. Xamarin.Essentials will automatically detect the file type (MIME) and request the file to be opened.

Here is a sample of writing text to disk and requesting it be opened:

```
var fn = "File.txt";
var file = Path.Combine(FileSystem.CacheDirectory, fn);
File.WriteAllText(file, "Hello World");

await Launcher.OpenAsync(new OpenFileRequest
{
    File = new ReadOnlyFile(file)
});
```

Presentation Location When Opening Files

When requesting a share or opening launcher on iPadOS you have the ability to present in a pop over control. This specifies where the pop over will appear and point an arrow directly to. This location is often the control that launched the action. You can specify the location using the `PresentationSourceBounds` property:

```
await Share.RequestAsync(new ShareFileRequest
{
    Title = Title,
    File = new ShareFile(file),
    PresentationSourceBounds = DeviceInfo.Platform== DevicePlatform.iOS && DeviceInfo.Idiom ==
DeviceIdiom.Tablet
                            ? new System.Drawing.Rectangle(0, 20, 0, 0)
                            : System.Drawing.Rectangle.Empty
});
```

```
await Launcher.OpenAsync(new OpenFileRequest
{
    File = new ReadOnlyFile(file),
    PresentationSourceBounds = DeviceInfo.Platform== DevicePlatform.iOS && DeviceInfo.Idiom ==
DeviceIdiom.Tablet
                            ? new System.Drawing.Rectangle(0, 20, 0, 0)
                            : System.Drawing.Rectangle.Empty
});
```

Everything described here works equally for `Share` and `Launcher`.

If you are using Xamarin.Forms you are able to pass in a `View` and calculate the bounds:

```
public static class ViewHelpers
{
    public static Rectangle GetAbsoluteBounds(this Xamarin.Forms.View element)
    {
        Element looper = element;

        var absoluteX = element.X + element.Margin.Top;
        var absoluteY = element.Y + element.Margin.Left;

        // Add logic to handle titles, headers, or other non-view bars

        while (looper.Parent != null)
        {
            looper = looper.Parent;
            if (looper is Xamarin.Forms.View v)
            {
                absoluteX += v.X + v.Margin.Top;
                absoluteY += v.Y + v.Margin.Left;
            }
        }

        return new Rectangle(absoluteX, absoluteY, element.Width, element.Height);
    }

    public static System.Drawing.Rectangle ToSystemRectangle(this Rectangle rect) =>
        new System.Drawing.Rectangle((int)rect.X, (int)rect.Y, (int)rect.Width, (int)rect.Height);
}
```

This can then be used when calling `RequestAsync`:

```
public Command<Xamarin.Forms.View> ShareCommand { get; } = new Command<Xamarin.Forms.View>(Share);
async void Share(Xamarin.Forms.View element)
{
    try
    {
        Analytics.TrackEvent("ShareWithFriends");
        var bounds = element.GetAbsoluteBounds();

        await Share.RequestAsync(new ShareTextRequest
        {
            PresentationSourceBounds = bounds.ToSystemRectangle(),
            Title = "Title",
            Text = "Text"
        });
    }
    catch (Exception)
    {
        // Handle exception that share failed
    }
}
```

You can pass in the calling element when the `Command` is triggered:

```
<Button Text="Share"
        Command="{Binding ShareWithFriendsCommand}"
        CommandParameter="{Binding Source={RelativeSource Self}}"/>
```

Platform Differences

- Android
- iOS

- UWP

The Task returned from `CanOpenAsync` completes immediately.

API

- Launcher source code
- Launcher API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Magnetometer

11/2/2020 • 2 minutes to read • Edit Online

The **Magnetometer** class lets you monitor the device's magnetometer sensor which indicates the device's orientation relative to Earth's magnetic field.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using Magnetometer

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Magnetometer functionality works by calling the `Start` and `Stop` methods to listen for changes to the magnetometer. Any changes are sent back through the `ReadingChanged` event. Here is sample usage:

```csharp
public class MagnetometerTest
{
    // Set speed delay for monitoring changes.
    SensorSpeed speed = SensorSpeed.UI;

    public MagnetometerTest()
    {
        // Register for reading changes.
        Magnetometer.ReadingChanged += Magnetometer_ReadingChanged;
    }

    void Magnetometer_ReadingChanged(object sender, MagnetometerChangedEventArgs e)
    {
        var data = e.Reading;
        // Process MagneticField X, Y, and Z
        Console.WriteLine($"Reading: X: {data.MagneticField.X}, Y: {data.MagneticField.Y}, Z:
{data.MagneticField.Z}");
    }

    public void ToggleMagnetometer()
    {
        try
        {
            if (Magnetometer.IsMonitoring)
              Magnetometer.Stop();
            else
              Magnetometer.Start(speed);
        }
        catch (FeatureNotSupportedException fnsEx)
        {
            // Feature not supported on device
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}
```

All data is returned in µT (microteslas).

Sensor Speed

- **Fastest** – Get the sensor data as fast as possible (not guaranteed to return on UI thread).
- **Game** – Rate suitable for games (not guaranteed to return on UI thread).
- **Default** – Default rate suitable for screen orientation changes.
- **UI** – Rate suitable for general user interface.

If your event handler is not guaranteed to run on the UI thread, and if the event handler needs to access user-interface elements, use the `MainThread.BeginInvokeOnMainThread` method to run that code on the UI thread.

API

- Magnetometer source code
- Magnetometer API documentation

Xamarin.Essentials: MainThread

11/2/2020 • 3 minutes to read • Edit Online

The **MainThread** class allows applications to run code on the main thread of execution, and to determine if a particular block of code is currently running on the main thread.

Background

Most operating systems — including iOS, Android, and the Universal Windows Platform — use a single-threading model for code involving the user interface. This model is necessary to properly serialize user-interface events, including keystrokes and touch input. This thread is often called the *main thread* or the *user-interface thread* or the *UI thread*. The disadvantage of this model is that all code that accesses user interface elements must run on the application's main thread.

Applications sometimes need to use events that call the event handler on a secondary thread of execution. (The Xamarin.Essentials classes `Accelerometer`, `Compass`, `Gyroscope`, `Magnetometer`, and `OrientationSensor` all might return information on a secondary thread when used with faster speeds.) If the event handler needs to access user-interface elements, it must run that code on the main thread. The **MainThread** class allows the application to run this code on the main thread.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Running Code on the Main Thread

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

To run code on the main thread, call the static `MainThread.BeginInvokeOnMainThread` method. The argument is an `Action` object, which is simply a method with no arguments and no return value:

```
MainThread.BeginInvokeOnMainThread(() =>
{
    // Code to run on the main thread
});
```

It is also possible to define a separate method for the code that must run on the main thread:

```
void MyMainThreadCode()
{
    // Code to run on the main thread
}
```

You can then run this method on the main thread by referencing it in the `BeginInvokeOnMainThread` method:

```
MainThread.BeginInvokeOnMainThread(MyMainThreadCode);
```

> **NOTE**
>
> Xamarin.Forms has a method called `Device.BeginInvokeOnMainThread(Action)` that does the same thing as `MainThread.BeginInvokeOnMainThread(Action)`. While you can use either method in a Xamarin.Forms app, consider whether or not the calling code has any other need for a dependency on Xamarin.Forms. If not, `MainThread.BeginInvokeOnMainThread(Action)` is likely a better option.

Determining if Code is Running on the Main Thread

The `MainThread` class also allows an application to determine if a particular block of code is running on the main thread. The `IsMainThread` property returns `true` if the code calling the property is running on the main thread. A program can use this property to run different code for the main thread or a secondary thread:

```
if (MainThread.IsMainThread)
{
    // Code to run if this is the main thread
}
else
{
    // Code to run if this is a secondary thread
}
```

You might wonder if you should check if code is running on a secondary thread before calling `BeginInvokeOnMainThread`, for example, like this:

```
if (MainThread.IsMainThread)
{
    MyMainThreadCode();
}
else
{
    MainThread.BeginInvokeOnMainThread(MyMainThreadCode);
}
```

You might suspect that this check might improve performance if the block of code is already running on the main thread.

However, this check is not necessary. The platform implementations of `BeginInvokeOnMainThread` themselves check if the call is made on the main thread. There is very little performance penalty if you call `BeginInvokeOnMainThread` when it's not really necessary.

Additional Methods

The `MainThread` class includes the following additional `static` methods that can be used to interact with user interface elements from backgrounds threads:

METHOD	ARGUMENTS	RETURNS	PURPOSE
`InvokeOnMainThreadAsync<T>`	`Func<T>`	`Task<T>`	Invokes a `Func<T>` on the main thread, and waits for it to complete.
`InvokeOnMainThreadAsync`	`Action`	`Task`	Invokes an `Action` on the main thread, and waits for it to complete.

METHOD	ARGUMENTS	RETURNS	PURPOSE
`InvokeOnMainThreadAsync<T>`	`Func<Task<T>>`	`Task<T>`	Invokes a `Func<Task<T>>` on the main thread, and waits for it to complete.
`InvokeOnMainThreadAsync`	`Func<Task>`	`Task`	Invokes a `Func<Task>` on the main thread, and waits for it to complete.
`GetMainThreadSynchronizationContextAsync`		`Task<SynchronizationContext>`	Returns the `SynchronizationContext` for the main thread.

API

- MainThread source code
- MainThread API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Map

11/2/2020 • 2 minutes to read • Edit Online

The **Map** class enables an application to open the installed map application to a specific location or placemark.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using Map

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Map functionality works by calling the `OpenAsync` method with the `Location` or `Placemark` to open with optional `MapLaunchOptions`.

```
public class MapTest
{
    public async Task NavigateToBuilding25()
    {
        var location = new Location(47.645160, -122.1306032);
        var options =  new MapLaunchOptions { Name = "Microsoft Building 25" };

        try
        {
            await Map.OpenAsync(location, options);
        }
        catch (Exception ex)
        {
            // No map application available to open
        }
    }
}
```

When opening with a `Placemark`, the following information is required:

- `CountryName`
- `AdminArea`
- `Thoroughfare`
- `Locality`

```
public class MapTest
{
    public async Task NavigateToBuilding25()
    {
        var placemark = new Placemark
            {
                CountryName = "United States",
                AdminArea = "WA",
                Thoroughfare = "Microsoft Building 25",
                Locality = "Redmond"
            };
        var options =  new MapLaunchOptions { Name = "Microsoft Building 25" };

        try
        {
            await Map.OpenAsync(placemark, options);
        }
        catch (Exception ex)
        {
            // No map application available to open or placemark can not be located
        }
    }
}
```

Extension Methods

If you already have a reference to a `Location` or `Placemark`, you can use the built-in extension method `OpenMapAsync` with optional `MapLaunchOptions`:

```
public class MapTest
{
    public async Task OpenPlacemarkOnMap(Placemark placemark)
    {
        try
        {
            await placemark.OpenMapAsync();
        }
        catch (Exception ex)
        {
            // No map application available to open
        }
    }
}
```

Directions Mode

If you call `OpenMapAsync` without any `MapLaunchOptions`, the map will launch to the location specified. Optionally, you can have a navigation route calculated from the device's current position. This is accomplished by setting the `NavigationMode` on the `MapLaunchOptions`:

```
public class MapTest
{
    public async Task NavigateToBuilding25()
    {
        var location = new Location(47.645160, -122.1306032);
        var options =  new MapLaunchOptions { NavigationMode = NavigationMode.Driving };

        await Map.OpenAsync(location, options);
    }
}
```

Platform Differences

- Android
- iOS
- UWP

- `NavigationMode` supports Bicycling, Driving, and Walking.

Platform Implementation Specifics

- Android
- iOS
- UWP

Android uses the `geo:` Uri scheme to launch the maps application on the device. This may prompt the user to select from an existing app that supports this Uri scheme. Xamarin.Essentials is tested with Google Maps, which supports this scheme.

API

- Map source code
- Map API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Media Picker

7/7/2021 • 2 minutes to read • Edit Online

The **MediaPicker** class lets a user pick or take a photo or video on the device.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **MediaPicker** functionality the following platform specific setup is required.

- Android
- iOS
- UWP

The following permissions are required and must be configured in the Android project. This can be added in the following ways:

Open the **AssemblyInfo.cs** file under the **Properties** folder and add:

```
// Needed for Picking photo/video
[assembly: UsesPermission(Android.Manifest.Permission.ReadExternalStorage)]

// Needed for Taking photo/video
[assembly: UsesPermission(Android.Manifest.Permission.WriteExternalStorage)]
[assembly: UsesPermission(Android.Manifest.Permission.Camera)]

// Add these properties if you would like to filter out devices that do not have cameras, or set to false to
make them optional
[assembly: UsesFeature("android.hardware.camera", Required = true)]
[assembly: UsesFeature("android.hardware.camera.autofocus", Required = true)]
```

OR Update Android Manifest:

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node.

```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.CAMERA" />
```

Or right click on the Android project and open the project's properties. Under **Android Manifest** find the **Required permissions:** area and check the these permissions. This will automatically update the **AndroidManifest.xml** file.

Using Media Picker

The `MediaPicker` class has the following methods that all return a `FileResult` that can be used to get the files location or read it as a `Stream`.

- `PickPhotoAsync`: Opens the media browser to select a photo.
- `CapturePhotoAsync`: Opens the camera to take a photo.

- `PickVideoAsync`: Opens the media browser to select a video.
- `CaptureVideoAsync`: Opens the camera to take a video.

Each method optionally takes in a `MediaPickerOptions` parameter that allows the `Title` to be set on some operating systems that is displayed to the users.

> **TIP**
> All methods must be called on the UI thread because permission checks and requests are automatically handled by Xamarin.Essentials.

General Usage

```
async Task TakePhotoAsync()
{
    try
    {
        var photo = await MediaPicker.CapturePhotoAsync();
        await LoadPhotoAsync(photo);
        Console.WriteLine($"CapturePhotoAsync COMPLETED: {PhotoPath}");
    }
    catch (FeatureNotSupportedException fnsEx)
    {
        // Feature is not supported on the device
    }
    catch (PermissionException pEx)
    {
        // Permissions not granted
    }
    catch (Exception ex)
    {
        Console.WriteLine($"CapturePhotoAsync THREW: {ex.Message}");
    }
}

async Task LoadPhotoAsync(FileResult photo)
{
    // canceled
    if (photo == null)
    {
        PhotoPath = null;
        return;
    }
    // save the file into local storage
    var newFile = Path.Combine(FileSystem.CacheDirectory, photo.FileName);
    using (var stream = await photo.OpenReadAsync())
    using (var newStream = File.OpenWrite(newFile))
        await stream.CopyToAsync(newStream);

    PhotoPath = newFile;
}
```

> **TIP**
> The `FullPath` property does not always return the physical path to the file. To get the file, use the `OpenReadAsync` method.

API

- MediaPicker source code
- MediaPicker API documentation

Xamarin.Essentials: Browser

7/8/2021 • 2 minutes to read • Edit Online

The **Browser** class enables an application to open a web link in the optimized system preferred browser or the external browser.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **Browser** functionality the following platform specific setup is required.

- Android
- iOS
- UWP

If your project's Target Android version is set to **Android 11 (R API 30)** you must update your Android Manifest with queries that are used with the new package visibility requirements.

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node:

```
<queries>
  <intent>
    <action android:name="android.intent.action.VIEW" />
    <data android:scheme="http"/>
  </intent>
  <intent>
    <action android:name="android.intent.action.VIEW" />
    <data android:scheme="https"/>
  </intent>
</queries>
```

Using Browser

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Browser functionality works by calling the `OpenAsync` method with the `Uri` and `BrowserLaunchMode`.

```
public class BrowserTest
{
    public async Task OpenBrowser(Uri uri)
    {
        try
        {
            await Browser.OpenAsync(uri, BrowserLaunchMode.SystemPreferred);
        }
        catch(Exception ex)
        {
            // An unexpected error occured. No browser may be installed on the device.
        }
    }
}
```

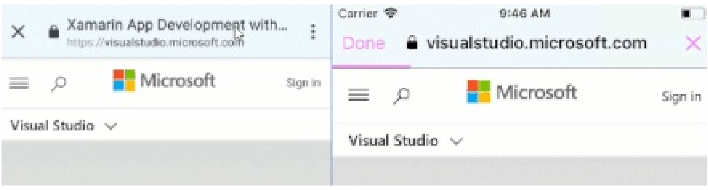
This method returns after the browser was *launched* and not necessarily *closed* by the user. The `bool` result indicates whether the launching was successful or not.

Customization

When using the system preferred browser there are several customization options available for iOS and Android. This includes a `TitleMode` (Android only), and preferred color options for the `Toolbar` (iOS and Android) and `Controls` (iOS only) that appear.

These options are specified using `BrowserLaunchOptions` when calling `OpenAsync`.

```
await Browser.OpenAsync(uri, new BrowserLaunchOptions
                {
                    LaunchMode = BrowserLaunchMode.SystemPreferred,
                    TitleMode = BrowserTitleMode.Show,
                    PreferredToolbarColor = Color.AliceBlue,
                    PreferredControlColor = Color.Violet
                });
```



Platform Implementation Specifics

- Android
- iOS
- UWP

The Launch Mode determines how the browser is launched:

System Preferred

Custom Tabs will attempted to be used to load the Uri and keep navigation awareness.

External

An `Intent` will be used to request the Uri be opened through the systems normal browser.

API

- Browser source code
- Browser API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: OrientationSensor

11/2/2020 • 3 minutes to read • Edit Online

The **OrientationSensor** class lets you monitor the orientation of a device in three dimensional space.

> **NOTE**
>
> This class is for determining the orientation of a device in 3D space. If you need to determine if the device's video display is in portrait or landscape mode, use the `Orientation` property of the `ScreenMetrics` object available from the `DeviceDisplay` class.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using OrientationSensor

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The `OrientationSensor` is enabled by calling the `Start` method to monitor changes to the device's orientation, and disabled by calling the `Stop` method. Any changes are sent back through the `ReadingChanged` event. Here is a sample usage:

```
public class OrientationSensorTest
{
    // Set speed delay for monitoring changes.
    SensorSpeed speed = SensorSpeed.UI;

    public OrientationSensorTest()
    {
        // Register for reading changes, be sure to unsubscribe when finished
        OrientationSensor.ReadingChanged += OrientationSensor_ReadingChanged;
    }

    void OrientationSensor_ReadingChanged(object sender, OrientationSensorChangedEventArgs e)
    {
        var data = e.Reading;
        Console.WriteLine($"Reading: X: {data.Orientation.X}, Y: {data.Orientation.Y}, Z:
{data.Orientation.Z}, W: {data.Orientation.W}");
        // Process Orientation quaternion (X, Y, Z, and W)
    }

    public void ToggleOrientationSensor()
    {
        try
        {
            if (OrientationSensor.IsMonitoring)
                OrientationSensor.Stop();
            else
                OrientationSensor.Start(speed);
        }
        catch (FeatureNotSupportedException fnsEx)
        {
            // Feature not supported on device
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}
```

`OrientationSensor` readings are reported back in the form of a `Quaternion` that describes the orientation of the device based on two 3D coordinate systems:

The device (generally a phone or tablet) has a 3D coordinate system with the following axes:

- The positive X axis points to the right of the display in portrait mode.
- The positive Y axis points to the top of the device in portrait mode.
- The positive Z axis points out of the screen.

The 3D coordinate system of the Earth has the following axes:

- The positive X axis is tangent to the surface of the Earth and points east.
- The positive Y axis is also tangent to the surface of the Earth and points north.
- The positive Z axis is perpendicular to the surface of the Earth and points up.

The `Quaternion` describes the rotation of the device's coordinate system relative to the Earth's coordinate system.

A `Quaternion` value is very closely related to rotation around an axis. If an axis of rotation is the normalized vector (a_x, a_y, a_z), and the rotation angle is Θ, then the (X, Y, Z, W) components of the quaternion are:

$(a_x \cdot \sin(\Theta/2), a_y \cdot \sin(\Theta/2), a_z \cdot \sin(\Theta/2), \cos(\Theta/2))$

These are right-hand coordinate systems, so with the thumb of the right hand pointed in the positive direction of the rotation axis, the curve of the fingers indicate the direction of rotation for positive angles.

Examples:

- When the device lies flat on a table with its screen facing up, with the top of the device (in portrait mode) pointing north, the two coordinate systems are aligned. The `Quaternion` value represents the identity quaternion (0, 0, 0, 1). All rotations can be analyzed relative to this position.
- When the device lies flat on a table with its screen facing up, and the top of the device (in portrait mode) pointing west, the `Quaternion` value is (0, 0, 0.707, 0.707). The device has been rotated 90 degrees around the Z axis of the Earth.
- When the device is held upright so that the top (in portrait mode) points towards the sky, and the back of the device faces north, the device has been rotated 90 degrees around the X axis. The `Quaternion` value is (0.707, 0, 0, 0.707).
- If the device is positioned so its left edge is on a table, and the top points north, the device has been rotated –90 degrees around the Y axis (or 90 degrees around the negative Y axis). The `Quaternion` value is (0, -0.707, 0, 0.707).

Sensor Speed

- **Fastest** – Get the sensor data as fast as possible (not guaranteed to return on UI thread).
- **Game** – Rate suitable for games (not guaranteed to return on UI thread).
- **Default** – Default rate suitable for screen orientation changes.
- **UI** – Rate suitable for general user interface.

If your event handler is not guaranteed to run on the UI thread, and if the event handler needs to access user-interface elements, use the `MainThread.BeginInvokeOnMainThread` method to run that code on the UI thread.

API

- OrientationSensor source code
- OrientationSensor API documentation

Xamarin.Essentials: Permissions

7/8/2021 • 6 minutes to read • Edit Online

The **Permissions** class provides the ability to check and request runtime permissions.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

This API uses runtime permissions on Android. Please ensure that Xamarin.Essentials is fully initialized and permission handling is setup in your app.

In the Android project's `MainLauncher` or any `Activity` that is launched Xamarin.Essentials must be initialized in the `OnCreate` method:

```
protected override void OnCreate(Bundle savedInstanceState)
{
    //...
    base.OnCreate(savedInstanceState);
    Xamarin.Essentials.Platform.Init(this, savedInstanceState); // add this line to your code, it may also
be called: bundle
    //...
}
```

To handle runtime permissions on Android, Xamarin.Essentials must receive any `OnRequestPermissionsResult`. Add the following code to all `Activity` classes:

```
public override void OnRequestPermissionsResult(int requestCode, string[] permissions,
Android.Content.PM.Permission[] grantResults)
{
    Xamarin.Essentials.Platform.OnRequestPermissionsResult(requestCode, permissions, grantResults);

    base.OnRequestPermissionsResult(requestCode, permissions, grantResults);
}
```

Using Permissions

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

Checking Permissions

To check the current status of a permission, use the `CheckStatusAsync` method along with the specific permission to get the status for.

```
var status = await Permissions.CheckStatusAsync<Permissions.LocationWhenInUse>();
```

A `PermissionException` is thrown if the required permission is not declared.

It's best to check the status of the permission before requesting it. Each operating system returns a different default state if the user has never been prompted. iOS returns `Unknown`, while others return `Denied`. If the status is `Granted` then there is no need to make other calls. On iOS if the status is `Denied` you should prompt the user to change the permission in the settings and on Android you can call `ShouldShowRationale` to detect if the user has already denied the permission in the past.

Requesting Permissions

To request a permission from the users, use the `RequestAsync` method along with the specific permission to request. If the user previously granted permission and hasn't revoked it, then this method will return `Granted` immediately and not display a dialog.

```
var status = await Permissions.RequestAsync<Permissions.LocationWhenInUse>();
```

A `PermissionException` is thrown if the required permission is not declared.

Note, that on some platforms a permission request can only be activated a single time. Further prompts must be handled by the developer to check if a permission is in the `Denied` state and ask the user to manually turn it on.

Permission Status

When using `CheckStatusAsync` or `RequestAsync` a `PermissionStatus` will be returned that can be used to determine the next steps:

- Unknown - The permission is in an unknown state
- Denied - The user denied the permission request
- Disabled - The feature is disabled on the device
- Granted - The user granted permission or is automatically granted
- Restricted - In a restricted state

Explain Why Permission Is Needed

It is best practice to explain why your application needs a specific permission. On iOS you must specify a string that is displayed to the user. Android does not have this ability and and also defaults permission status to Disabled. This limits the ability to know if the user denied the permission or if it is the first time prompting the user. The `ShouldShowRationale` method can be used to determine if an educational UI should be displayed. If the method returns `true` this is because the user has denied or disabled the permission in the past. Other platforms will always return `false` when calling this method.

Available Permissions

Xamarin.Essentials attempts to abstract as many permissions as possible. However, each operating system has a different set of runtime permissions. In addition there are differences when providing a single API for some permissions. Here is a guide to the currently available permissions:

Icon Guide:

- ✅ - Supported
- ❌ - Not supported/required

PERMISSION	ANDROID	IOS	UWP	WATCHOS	TVOS	TIZEN
CalendarRead	✅	✅	❌	✅	❌	❌

PERMISSION	ANDROID	IOS	UWP	WATCHOS	TVOS	TIZEN
CalendarWrite	✔️	✔️	❌	✔️	❌	❌
Camera	✔️	✔️	❌	❌	❌	✔️
ContactsRead	✔️	✔️	✔️	❌	❌	❌
ContactsWrite	✔️	✔️	✔️	❌	❌	❌
Flashlight	✔️	❌	❌	❌	❌	✔️
LocationWhenInUse	✔️	✔️	✔️	✔️	✔️	✔️
LocationAlways	✔️	✔️	✔️	✔️	❌	✔️
Media	❌	✔️	❌	❌	❌	❌
Microphone	✔️	✔️	✔️	❌	❌	✔️
Phone	✔️	✔️	❌	❌	❌	❌
Photos	❌	✔️	❌	❌	✔️	❌
Reminders	❌	✔️	❌	✔️	❌	❌
Sensors	✔️	✔️	✔️	✔️	❌	❌
Sms	✔️	✔️	❌	❌	❌	❌
Speech	✔️	✔️	❌	❌	❌	❌
StorageRead	✔️	❌	❌	❌	❌	❌
StorageWrite	✔️	❌	❌	❌	❌	❌

If a permission is marked as ❌ it will always return `Granted` when checked or requested.

General Usage

The following code presents the general usage pattern for determining whether a permission has been granted and requesting it if it has not. This code uses features that are available with Xamarin.Essentials version 1.6.0 or later.

```
public async Task<PermissionStatus> CheckAndRequestLocationPermission()
{
    var status = await Permissions.CheckStatusAsync<Permissions.LocationWhenInUse>();

    if (status == PermissionStatus.Granted)
        return status;

    if (status == PermissionStatus.Denied && DeviceInfo.Platform == DevicePlatform.iOS)
    {
        // Prompt the user to turn on in settings
        // On iOS once a permission has been denied it may not be requested again from the application
        return status;
    }

    if (Permissions.ShouldShowRationale<Permissions.LocationWhenInUse>())
    {
        // Prompt the user with additional information as to why the permission is needed
    }

    status = await Permissions.RequestAsync<Permissions.LocationWhenInUse>();

    return status;
}
```

Each permission type can have an instance of it created that the methods can be called directly.

```
public async Task GetLocationAsync()
{
    var status = await CheckAndRequestPermissionAsync(new Permissions.LocationWhenInUse());
    if (status != PermissionStatus.Granted)
    {
        // Notify user permission was denied
        return;
    }

    var location = await Geolocation.GetLocationAsync();
}

public async Task<PermissionStatus> CheckAndRequestPermissionAsync<T>(T permission)
            where T : BasePermission
{
    var status = await permission.CheckStatusAsync();
    if (status != PermissionStatus.Granted)
    {
        status = await permission.RequestAsync();
    }

    return status;
}
```

Extending Permissions

The Permissions API was created to be flexible and extensible for applications that require additional validation or permissions that aren't included in Xamarin.Essentials. Create a new class that inherits from `BasePermission` and implement the required abstract methods.

```
public class MyPermission : BasePermission
{
    // This method checks if current status of the permission
    public override Task<PermissionStatus> CheckStatusAsync()
    {
        throw new System.NotImplementedException();
    }

    // This method is optional and a PermissionException is often thrown if a permission is not declared
    public override void EnsureDeclared()
    {
        throw new System.NotImplementedException();
    }

    // Requests the user to accept or deny a permission
    public override Task<PermissionStatus> RequestAsync()
    {
        throw new System.NotImplementedException();
    }
}
```

When implementing a permission in a specific platform, the `BasePlatformPermission` class can be inherited from. This provides additional platform helper methods to automatically check the declarations. This can help when creating custom permissions that do groupings. For example, you can request both Read and Write access to storage on Android using the following custom permission.

```
public class ReadWriteStoragePermission : Xamarin.Essentials.Permissions.BasePlatformPermission
{
    public override (string androidPermission, bool isRuntime)[] RequiredPermissions => new List<(string
androidPermission, bool isRuntime)>
    {
        (Android.Manifest.Permission.ReadExternalStorage, true),
        (Android.Manifest.Permission.WriteExternalStorage, true)
    }.ToArray();
}
```

Then you can call your new permission from Android project.

```
await Permissions.RequestAsync<ReadWriteStoragePermission>();
```

If you wanted to call this API from your shared code you could create an interface and use a dependency service to register and get the implementation.

```
public interface IReadWritePermission
{
    Task<PermissionStatus> CheckStatusAsync();
    Task<PermissionStatus> RequestAsync();
}
```

Then implement the interface in your platform project:

```
public class ReadWriteStoragePermission : Xamarin.Essentials.Permissions.BasePlatformPermission,
IReadWritePermission
{
    public override (string androidPermission, bool isRuntime)[] RequiredPermissions => new List<(string
androidPermission, bool isRuntime)>
    {
        (Android.Manifest.Permission.ReadExternalStorage, true),
        (Android.Manifest.Permission.WriteExternalStorage, true)
    }.ToArray();
}
```

You can then register the specific implementation:

```
DependencyService.Register<IReadWritePermission, ReadWriteStoragePermission>();
```

Then from your shared project you can resolve and use it:

```
var readWritePermission = DependencyService.Get<IReadWritePermission>();
var status = await readWritePermission.CheckStatusAsync();
if (status != PermissionStatus.Granted)
{
    status = await readWritePermission.RequestAsync();
}
```

Platform Implementation Specifics

- Android
- iOS
- UWP

Permissions must have the matching attributes set in the Android Manifest file. Permission status defaults to Denied.

Read more on the Permissions in Xamarin.Android documentation.

API

- Permissions source code
- Permissions API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Phone Dialer

11/2/2020 • 2 minutes to read • Edit Online

The **PhoneDialer** class enables an application to open a phone number in the dialer.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

- Android
- iOS
- UWP

If your project's Target Android version is set to **Android 11 (R API 30)** you must update your Android Manifest with queries that are used with the new package visibility requirements.

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node:

```
<queries>
  <intent>
    <action android:name="android.intent.action.DIAL" />
    <data android:scheme="tel"/>
  </intent>
</queries>
```

Using Phone Dialer

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Phone Dialer functionality works by calling the `Open` method with a phone number to open the dialer with. When `Open` is requested the API will automatically attempt to format the number based on the country code if specified.

```
public class PhoneDialerTest
{
    public void PlacePhoneCall(string number)
    {
        try
        {
            PhoneDialer.Open(number);
        }
        catch (ArgumentNullException anEx)
        {
            // Number was null or white space
        }
        catch (FeatureNotSupportedException ex)
        {
            // Phone Dialer is not supported on this device.
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}
```

API

- Phone Dialer source code
- Phone Dialer API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Platform Extensions

11/2/2020 • 2 minutes to read • Edit Online

Xamarin.Essentials provides several platform extension methods when having to work with platform types such as Rect, Size, and Point. This means that you can convert between the `System` version of these types for their iOS, Android, and UWP specific types.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using Platform Extensions

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

All platform extensions can only be called from the iOS, Android, or UWP project.

Android Extensions

These extensions can only be accessed from an Android project.

Application Context & Activity

Using the platform extensions in the `Platform` class you can get access to the current `Context` or `Activity` for the running app.

```
var context = Platform.AppContext;

// Current Activity or null if not initialized or not started.
var activity = Platform.CurrentActivity;
```

If there is a situation where the `Activity` is needed, but the application hasn't fully started then the `WaitForActivityAsync` method should be used.

```
var activity = await Platform.WaitForActivityAsync();
```

Activity Lifecycle

In addition to getting the current Activity, you can also register for lifecycle events.

```
protected override void OnCreate(Bundle bundle)
{
    base.OnCreate(bundle);

    Xamarin.Essentials.Platform.Init(this, bundle);

    Xamarin.Essentials.Platform.ActivityStateChanged += Platform_ActivityStateChanged;
}

protected override void OnDestroy()
{
    base.OnDestroy();
    Xamarin.Essentials.Platform.ActivityStateChanged -= Platform_ActivityStateChanged;
}

void Platform_ActivityStateChanged(object sender, Xamarin.Essentials.ActivityStateChangedEventArgs e) =>
    Toast.MakeText(this, e.State.ToString(), ToastLength.Short).Show();
```

Activity states are the following:

- Created
- Resumed
- Paused
- Destroyed
- SaveInstanceState
- Started
- Stopped

Read the Activity Lifecycle documentation to learn more.

iOS Extensions

These extensions can only be accessed from an iOS project.

Current UIViewController

Gain access to the currently visible `UIViewController`:

```
var vc = Platform.GetCurrentUIViewController();
```

This method will return `null` if unable to detect a `UIViewController`.

Cross-platform Extensions

These extensions exist in every platform.

Point

```
var system = new System.Drawing.Point(x, y);

// Convert to CoreGraphics.CGPoint, Android.Graphics.Point, and Windows.Foundation.Point
var platform = system.ToPlatformPoint();

// Back to System.Drawing.Point
var system2 = platform.ToSystemPoint();
```

Size

```
var system = new System.Drawing.Size(width, height);

// Convert to CoreGraphics.CGSize, Android.Util.Size, and Windows.Foundation.Size
var platform = system.ToPlatformSize();

// Back to System.Drawing.Size
var system2 = platform.ToSystemSize();
```

Rectangle

```
var system = new System.Drawing.Rectangle(x, y, width, height);

// Convert to CoreGraphics.CGRect, Android.Graphics.Rect, and Windows.Foundation.Rect
var platform = system.ToPlatformRectangle();

// Back to System.Drawing.Rectangle
var system2 = platform.ToSystemRectangle();
```

API

- Converters source code
- Point Converters API documentation
- Rectangle Converters API documentation
- Size Converters API documentation

Xamarin.Essentials: Preferences

11/2/2020 • 2 minutes to read • Edit Online

The **Preferences** class helps to store application preferences in a key/value store.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using Preferences

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

To save a value for a given *key* in preferences:

```
Preferences.Set("my_key", "my_value");
```

To retrieve a value from preferences or a default if not set:

```
var myValue = Preferences.Get("my_key", "default_value");
```

To check if a given *key* exists in preferences:

```
bool hasKey = Preferences.ContainsKey("my_key");
```

To remove the *key* from preferences:

```
Preferences.Remove("my_key");
```

To remove all preferences:

```
Preferences.Clear();
```

> **TIP**
> The above methods take in an optional `string` parameter called `sharedName`. This parameter is used to create additional containers for preferences which are helpful in some use cases. One use case is when your application needs to share preferences across extensions or to a watch application. Please read the platform implementation specifics below.

Supported Data Types

The following data types are supported in **Preferences**:

- bool
- double
- int
- float
- long
- string
- DateTime

Integrate with System Settings

Preferences are stored natively, which allows you to integrate your settings into the native system settings. Follow the platform documentation and samples to integrate with the platform:

- Apple: Implementing an iOS Settings Bundle
- iOS Applicaton Preferences Sample
- watchOS Settings
- Android: Getting Started with Settings Screens

Implementation Details

Values of `DateTime` are stored in a 64-bit binary (long integer) format using two methods defined by the `DateTime` class: The `ToBinary` method is used to encode the `DateTime` value, and the `FromBinary` method decodes the value. See the documentation of these methods for adjustments that might be made to decoded values when a `DateTime` is stored that is not a Coordinated Universal Time (UTC) value.

Platform Implementation Specifics

- Android
- iOS
- UWP

All data is stored into Shared Preferences. If no `sharedName` is specified the default shared preferences are used, otherwise the name is used to get a **private** shared preferences with the specified name.

Persistence

Uninstalling the application will cause all *Preferences* to be removed, with the exception being apps that target and run on Android 6.0 (API level 23) or later that use **Auto Backup**. This feature is on by default and preserves app data including **Shared Preferences**, which is what the **Preferences** API utilizes. You can disable this by following Google's documentation.

Limitations

When storing a string, this API is intended to store small amounts of text. Performance may be subpar if you try to use it to store large amounts of text.

API

- Preferences source code
- Preferences API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Screenshot

4/14/2021 • 2 minutes to read • Edit Online

The **Screenshot** class lets you take a capture of the current displayed screen of the app.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using Screenshot

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

Then call `CaptureAsync` to take a screenshot of the current screen of the running application. This will return back a `ScreenshotResult` that can be used to get the `Width`, `Height`, and a `Stream` of the screenshot taken.

```
async Task CaptureScreenshot()
{
    var screenshot = await Screenshot.CaptureAsync();
    var stream = await screenshot.OpenReadAsync();

    Image = ImageSource.FromStream(() => stream);
}
```

Limitations

Not all views support being captured at a screen level such as an OpenGL view.

API

- Screenshot source code
- Screenshot API documentation

Xamarin.Essentials: Secure Storage

11/2/2020 • 4 minutes to read • Edit Online

The **SecureStorage** class helps securely store simple key/value pairs.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **SecureStorage** functionality, the following platform-specific setup is required:

- Android
- iOS
- UWP

> **TIP**
> Auto Backup for Apps is a feature of Android 6.0 (API level 23) and later that backs up user's app data (shared preferences, files in the app's internal storage, and other specific files). Data is restored when an app is re-installed or installed on a new device. This can impact `SecureStorage` which utilizes share preferences that are backed up and can not be decrypted when the restore occurs. Xamarin.Essentials automatically handles this case by removing the key so it can be reset, but you can take an additional step by disabling Auto Backup.

Enable or disable backup

You can choose to disable Auto Backup for your entire application by setting the `android:allowBackup` setting to false in the `AndroidManifest.xml` file. This approach is only recommended if you plan on restoring data in another way.

```
<manifest ... >
    ...
    <application android:allowBackup="false" ... >
        ...
    </application>
</manifest>
```

Selective Backup

Auto Backup can be configured to disable specific content from backing up. You can create a custom rule set to exclude `SecureStore` items from being backed up.

1. Set the `android:fullBackupContent` attribute in your **AndroidManifest.xml**:

   ```
   <application ...
       android:fullBackupContent="@xml/auto_backup_rules">
   </application>
   ```

2. Create a new XML file named **auto_backup_rules.xml** in the **Resources/xml** directory with the build action of **AndroidResource**. Then set the following content that includes all shared preferences except for `SecureStorage`:

```xml
<?xml version="1.0" encoding="utf-8"?>
<full-backup-content>
    <include domain="sharedpref" path="."/>
    <exclude domain="sharedpref" path="${applicationId}.xamarinessentials.xml"/>
</full-backup-content>
```

Using Secure Storage

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

To save a value for a given *key* in secure storage:

```
try
{
  await SecureStorage.SetAsync("oauth_token", "secret-oauth-token-value");
}
catch (Exception ex)
{
  // Possible that device doesn't support secure storage on device.
}
```

To retrieve a value from secure storage:

```
try
{
  var oauthToken = await SecureStorage.GetAsync("oauth_token");
}
catch (Exception ex)
{
  // Possible that device doesn't support secure storage on device.
}
```

NOTE

If there is no value associated with the requested key, `GetAsync` will return `null`.

To remove a specific key, call:

```
SecureStorage.Remove("oauth_token");
```

To remove all keys, call:

```
SecureStorage.RemoveAll();
```

TIP

It is possible that an exception is thrown when calling `GetAsync` or `SetAsync`. This can be caused by a device not supporting secure storage, encryption keys changing, or corruption of data. It is best to handle this by removing and adding the setting back if possible.

Platform Implementation Specifics

- Android
- iOS
- UWP

The Android KeyStore is used to store the cipher key used to encrypt the value before it is saved into a Shared Preferences with a filename of **[YOUR-APP-PACKAGE-ID].xamarinessentials**. The key (not a cryptographic key, the *key* to the *value*) used in the shared preferences file is a *MD5 Hash* of the key passed into the `SecureStorage` APIs.

API Level 23 and Higher

On newer API levels, an **AES** key is obtained from the Android KeyStore and used with an **AES/GCM/NoPadding** cipher to encrypt the value before it is stored in the shared preferences file.

API Level 22 and Lower

On older API levels, the Android KeyStore only supports storing **RSA** keys, which is used with an **RSA/ECB/PKCS1Padding** cipher to encrypt an **AES** key (randomly generated at runtime) and stored in the shared preferences file under the key *SecureStorageKey*, if one has not already been generated.

SecureStorage uses the Preferences API and follows the same data persistence outlined in the Preferences documentation. If a device upgrades from API level 22 or lower to API level 23 and higher, this type of encryption will continue to be used unless the app is uninstalled or **RemoveAll** is called.

Limitations

This API is intended to store small amounts of text. Performance may be slow if you try to use it to store large amounts of text.

API

- SecureStorage source code
- SecureStorage API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Share

7/15/2021 • 3 minutes to read • Edit Online

The **Share** class enables an application to share data such as text and web links to other applications on the device.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

- Android
- iOS
- UWP

No additional setup required.

Using Share

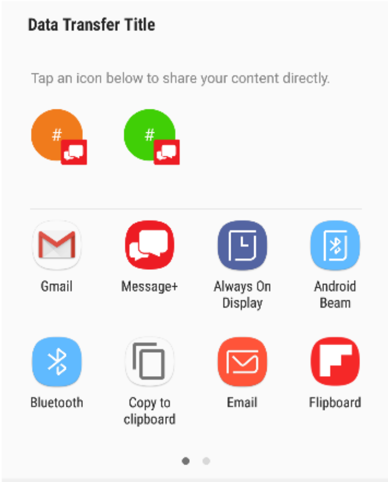
Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

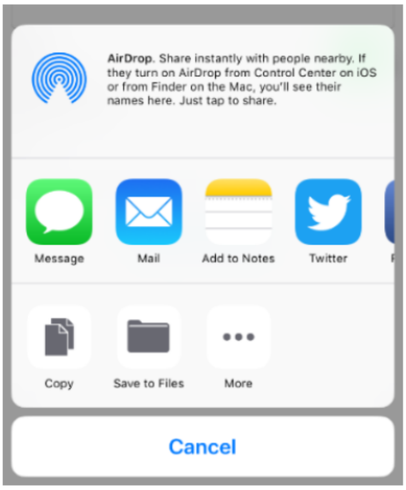
The Share functionality works by calling the `RequestAsync` method with a data request payload that includes information to share to other applications. Text and Uri can be mixed and each platform will handle filtering based on content.

```
public class ShareTest
{
    public async Task ShareText(string text)
    {
        await Share.RequestAsync(new ShareTextRequest
            {
                Text = text,
                Title = "Share Text"
            });
    }

    public async Task ShareUri(string uri)
    {
        await Share.RequestAsync(new ShareTextRequest
            {
                Uri = uri,
                Title = "Share Web Link"
            });
    }
}
```

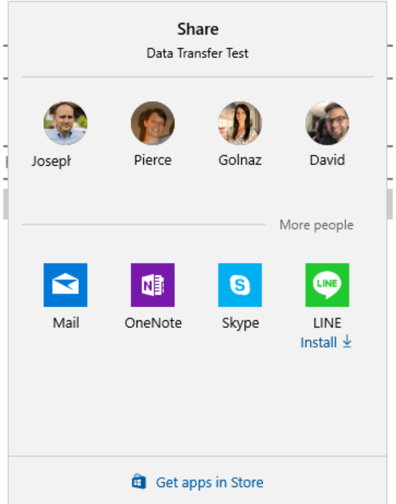
User interface to share to external application that appears when request is made:



Android



iOS



UWP

File

This features enables an app to share files to other applications on the device. Xamarin.Essentials will automatically detect the file type (MIME) and request a share. Each platform may only support specific file extensions.

Here is a sample of writing text to disk and sharing it to other apps:

```
var fn =  "Attachment.txt";
var file = Path.Combine(FileSystem.CacheDirectory, fn);
File.WriteAllText(file, "Hello World");

await Share.RequestAsync(new ShareFileRequest
{
    Title = Title,
    File = new ShareFile(file)
});
```

Multiple Files

The usage of share multiple files differs from the single file only in the ability of sending several files at once:

```
var file1 = Path.Combine(FileSystem.CacheDirectory, "Attachment1.txt");
File.WriteAllText(file, "Content 1");
var file2 = Path.Combine(FileSystem.CacheDirectory, "Attachment2.txt");
File.WriteAllText(file, "Content 2");

await Share.RequestAsync(new ShareMultipleFilesRequest
{
    Title = ShareFilesTitle,
    Files = new List<ShareFile> { new ShareFile(file1), new ShareFile(file2) }
});
```

Presentation Location

When requesting a share or opening launcher on iPadOS you have the ability to present in a pop over control. This specifies where the pop over will appear and point an arrow directly to. This location is often the control that launched the action. You can specify the location using the `PresentationSourceBounds` property:

```
await Share.RequestAsync(new ShareFileRequest
{
    Title = Title,
    File = new ShareFile(file),
    PresentationSourceBounds = DeviceInfo.Platform== DevicePlatform.iOS && DeviceInfo.Idiom ==
DeviceIdiom.Tablet
                            ? new System.Drawing.Rectangle(0, 20, 0, 0)
                            : System.Drawing.Rectangle.Empty
});
```

```
await Launcher.OpenAsync(new OpenFileRequest
{
    File = new ReadOnlyFile(file),
    PresentationSourceBounds = DeviceInfo.Platform== DevicePlatform.iOS && DeviceInfo.Idiom ==
DeviceIdiom.Tablet
                            ? new System.Drawing.Rectangle(0, 20, 0, 0)
                            : System.Drawing.Rectangle.Empty
});
```

Everything described here works equally for `Share` and `Launcher`.

If you are using Xamarin.Forms you are able to pass in a `View` and calculate the bounds:

```
public static class ViewHelpers
{
    public static Rectangle GetAbsoluteBounds(this Xamarin.Forms.View element)
    {
        Element looper = element;

        var absoluteX = element.X + element.Margin.Top;
        var absoluteY = element.Y + element.Margin.Left;

        // Add logic to handle titles, headers, or other non-view bars

        while (looper.Parent != null)
        {
            looper = looper.Parent;
            if (looper is Xamarin.Forms.View v)
            {
                absoluteX += v.X + v.Margin.Top;
                absoluteY += v.Y + v.Margin.Left;
            }
        }

        return new Rectangle(absoluteX, absoluteY, element.Width, element.Height);
    }

    public static System.Drawing.Rectangle ToSystemRectangle(this Rectangle rect) =>
        new System.Drawing.Rectangle((int)rect.X, (int)rect.Y, (int)rect.Width, (int)rect.Height);
}
```

This can then be used when calling `RequestAsync`:

```
public Command<Xamarin.Forms.View> ShareCommand { get; } = new Command<Xamarin.Forms.View>(Share);
async void Share(Xamarin.Forms.View element)
{
    try
    {
        Analytics.TrackEvent("ShareWithFriends");
        var bounds = element.GetAbsoluteBounds();

        await Share.RequestAsync(new ShareTextRequest
        {
            PresentationSourceBounds = bounds.ToSystemRectangle(),
            Title = "Title",
            Text = "Text"
        });
    }
    catch (Exception)
    {
        // Handle exception that share failed
    }
}
```

You can pass in the calling element when the `Command` is triggered:

```
<Button Text="Share"
        Command="{Binding ShareWithFriendsCommand}"
        CommandParameter="{Binding Source={RelativeSource Self}}"/>
```

Platform Differences

- Android
- iOS
- UWP

- `Subject` property is used for desired subject of a message.

API

- Share source code
- Share API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: SMS

11/2/2020 • 2 minutes to read • Edit Online

The **Sms** class enables an application to open the default SMS application with a specified message to send to a recipient.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **Sms** functionality the following platform specific setup is required.

- Android
- iOS
- UWP

If your project's Target Android version is set to **Android 11 (R API 30)** you must update your Android Manifest with queries that are used with the new package visibility requirements.

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node:

```
<queries>
  <intent>
    <action android:name="android.intent.action.VIEW" />
    <data android:scheme="smsto"/>
  </intent>
</queries>
```

Using Sms

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The SMS functionality works by calling the `ComposeAsync` method an `SmsMessage` that contains the message's recipient and the body of the message, both of which are optional.

```
public class SmsTest
{
    public async Task SendSms(string messageText, string recipient)
    {
        try
        {
            var message = new SmsMessage(messageText, new []{ recipient });
            await Sms.ComposeAsync(message);
        }
        catch (FeatureNotSupportedException ex)
        {
            // Sms is not supported on this device.
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}
```

Additionally, you can pass in multiple receipients to a `SmsMessage`:

```
public class SmsTest
{
    public async Task SendSms(string messageText, string[] recipients)
    {
        try
        {
            var message = new SmsMessage(messageText, recipients);
            await Sms.ComposeAsync(message);
        }
        catch (FeatureNotSupportedException ex)
        {
            // Sms is not supported on this device.
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}
```

API

- Sms source code
- Sms API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Text-to-Speech

11/2/2020 • 2 minutes to read • Edit Online

The **TextToSpeech** class enables an application to utilize the built-in text-to-speech engines to speak back text from the device and also to query available languages that the engine can support.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using Text-to-Speech

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

Text-to-Speech works by calling the `SpeakAsync` method with text and optional parameters, and returns after the utterance has finished.

```
public async Task SpeakNowDefaultSettings()
{
    await TextToSpeech.SpeakAsync("Hello World");

    // This method will block until utterance finishes.
}

public void SpeakNowDefaultSettings2()
{
    TextToSpeech.SpeakAsync("Hello World").ContinueWith((t) =>
    {
        // Logic that will run after utterance finishes.

    }, TaskScheduler.FromCurrentSynchronizationContext());
}
```

This method takes in an optional `CancellationToken` to stop the utterance once it starts.

```
CancellationTokenSource cts;
public async Task SpeakNowDefaultSettings()
{
    cts = new CancellationTokenSource();
    await TextToSpeech.SpeakAsync("Hello World", cancelToken: cts.Token);

    // This method will block until utterance finishes.
}

// Cancel speech if a cancellation token exists & hasn't been already requested.
public void CancelSpeech()
{
    if (cts?.IsCancellationRequested ?? true)
        return;

    cts.Cancel();
}
```

Text-to-Speech will automatically queue speech requests from the same thread.

```
bool isBusy = false;
public void SpeakMultiple()
{
    isBusy = true;
    Task.Run(async () =>
    {
        await TextToSpeech.SpeakAsync("Hello World 1");
        await TextToSpeech.SpeakAsync("Hello World 2");
        await TextToSpeech.SpeakAsync("Hello World 3");
        isBusy = false;
    });

    // or you can query multiple without a Task:
    Task.WhenAll(
        TextToSpeech.SpeakAsync("Hello World 1"),
        TextToSpeech.SpeakAsync("Hello World 2"),
        TextToSpeech.SpeakAsync("Hello World 3"))
        .ContinueWith((t) => { isBusy = false; }, TaskScheduler.FromCurrentSynchronizationContext());
}
```

Speech Settings

For more control over how the audio is spoken back with `SpeechOptions` that allows setting the volume, pitch, and locale.

```
public async Task SpeakNow()
{
    var settings = new SpeechOptions()
        {
            Volume = .75f,
            Pitch = 1.0f
        };

    await TextToSpeech.SpeakAsync("Hello World", settings);
}
```

The following are supported values for these parameters:

PARAMETER	MINIMUM	MAXIMUM
Pitch	0	2.0

PARAMETER	MINIMUM	MAXIMUM
Volume	0	1.0

Speech Locales

Each platform supports different locales, to speak back text in different languages and accents. Platforms have different codes and ways of specifying the locale, which is why Xamarin.Essentials provides a cross-platform `Locale` class and a way to query them with `GetLocalesAsync`.

```
public async Task SpeakNow()
{
    var locales = await TextToSpeech.GetLocalesAsync();

    // Grab the first locale
    var locale = locales.FirstOrDefault();

    var settings = new SpeechOptions()
        {
            Volume = .75f,
            Pitch = 1.0f,
            Locale = locale
        };

    await TextToSpeech.SpeakAsync("Hello World", settings);
}
```

Limitations

- Utterance queue is not guaranteed if called across multiple threads.
- Background audio playback is not officially supported.

API

- TextToSpeech source code
- TextToSpeech API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Unit Converters

11/2/2020 • 2 minutes to read • Edit Online

The **UnitConverters** class provides several unit converters to help developers when using Xamarin.Essentials.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using Unit Converters

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

All unit converters are available by using the static `UnitConverters` class in Xamarin.Essentials. For instance you can easily convert Fahrenheit to Celsius.

```
var celsius = UnitConverters.FahrenheitToCelsius(32.0);
```

Here is a list of available conversions:

- FahrenheitToCelsius
- CelsiusToFahrenheit
- CelsiusToKelvin
- KelvinToCelsius
- MilesToMeters
- MilesToKilometers
- KilometersToMiles
- MetersToInternationalFeet
- InternationalFeetToMeters
- DegreesToRadians
- RadiansToDegrees
- DegreesPerSecondToRadiansPerSecond
- RadiansPerSecondToDegreesPerSecond
- DegreesPerSecondToHertz
- RadiansPerSecondToHertz
- HertzToDegreesPerSecond
- HertzToRadiansPerSecond
- KilopascalsToHectopascals
- HectopascalsToKilopascals
- KilopascalsToPascals
- HectopascalsToPascals
- AtmospheresToPascals
- PascalsToAtmospheres

- CoordinatesToMiles
- CoordinatesToKilometers
- KilogramsToPounds
- PoundsToKilograms
- StonesToPounds
- PoundsToStones

API

- Unit Converters source code
- Unit Converters API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Version Tracking

11/2/2020 • 2 minutes to read • Edit Online

The **VersionTracking** class lets you check the applications version and build numbers along with seeing additional information such as if it is the first time the application launched ever or for the current version, get the previous build information, and more.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

Using Version Tracking

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The first time you use the **VersionTracking** class it will start tracking the current version. You must call `Track` early only in your application each time it is loaded to ensure the current version information is tracked:

```
VersionTracking.Track();
```

After the initial `Track` is called version information can be read:

```
// First time ever launched application
var firstLaunch = VersionTracking.IsFirstLaunchEver;

// First time launching current version
var firstLaunchCurrent = VersionTracking.IsFirstLaunchForCurrentVersion;

// First time launching current build
var firstLaunchBuild = VersionTracking.IsFirstLaunchForCurrentBuild;

// Current app version (2.0.0)
var currentVersion = VersionTracking.CurrentVersion;

// Current build (2)
var currentBuild = VersionTracking.CurrentBuild;

// Previous app version (1.0.0)
var previousVersion = VersionTracking.PreviousVersion;

// Previous app build (1)
var previousBuild = VersionTracking.PreviousBuild;

// First version of app installed (1.0.0)
var firstVersion = VersionTracking.FirstInstalledVersion;

// First build of app installed (1)
var firstBuild = VersionTracking.FirstInstalledBuild;

// List of versions installed (1.0.0, 2.0.0)
var versionHistory = VersionTracking.VersionHistory;

// List of builds installed (1, 2)
var buildHistory = VersionTracking.BuildHistory;
```

Platform Implementation Specifics

All version information is stored using the Preferences API in Xamarin.Essentials and is stored with a filename of **[YOUR-APP-PACKAGE-ID].xamarinessentials.versiontracking** and follows the same data persistence outlined in the Preferences documentation.

API

- Version Tracking source code
- Version Tracking API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Vibration

11/2/2020 • 2 minutes to read • Edit Online

The **Vibration** class lets you start and stop the vibrate functionality for a desired amount of time.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **Vibration** functionality the following platform specific setup is required.

- Android
- iOS
- UWP

The Vibrate permission is required and must be configured in the Android project. This can be added in the following ways:

Open the **AssemblyInfo.cs** file under the **Properties** folder and add:

```
[assembly: UsesPermission(Android.Manifest.Permission.Vibrate)]
```

OR Update Android Manifest:

Open the **AndroidManifest.xml** file under the **Properties** folder and add the following inside of the **manifest** node.

```
<uses-permission android:name="android.permission.VIBRATE" />
```

Or right click on the Android project and open the project's properties. Under **Android Manifest** find the **Required permissions:** area and check the **VIBRATE** permission. This will automatically update the **AndroidManifest.xml** file.

Using Vibration

Add a reference to Xamarin.Essentials in your class:

```
using Xamarin.Essentials;
```

The Vibration functionality can be requested for a set amount of time or the default of 500 milliseconds.

```
try
{
    // Use default vibration length
    Vibration.Vibrate();

    // Or use specified time
    var duration = TimeSpan.FromSeconds(1);
    Vibration.Vibrate(duration);
}
catch (FeatureNotSupportedException ex)
{
    // Feature not supported on device
}
catch (Exception ex)
{
    // Other error has occurred.
}
```

Cancellation of device vibration can be requested with the `Cancel` method:

```
try
{
    Vibration.Cancel();
}
catch (FeatureNotSupportedException ex)
{
    // Feature not supported on device
}
catch (Exception ex)
{
    // Other error has occurred.
}
```

Platform Differences

- Android
- iOS
- UWP

No platform differences.

API

- Vibration source code
- Vibration API documentation

Related Video

Find more Xamarin videos on Channel 9 and YouTube.

Xamarin.Essentials: Web Authenticator

7/15/2021 • 7 minutes to read • Edit Online

The **WebAuthenticator** class lets you initiate browser based flows which listen for a callback to a specific URL registered to the app.

Overview

Many apps require adding user authentication, and this often means enabling your users to sign in their existing Microsoft, Facebook, Google, and now Apple Sign In accounts.

Microsoft Authentication Library (MSAL) provides an excellent turn-key solution to adding authentication to your app. There's even support for Xamarin apps in their client NuGet package.

If you're interested in using your own web service for authentication, it's possible to use **WebAuthenticator** to implement the client side functionality.

Why use a server back end?

Many authentication providers have moved to only offering explicit or two-legged authentication flows to ensure better security. This means you'll need a *'client secret'* from the provider to complete the authentication flow. Unfortunately, mobile apps are not a great place to store secrets and anything stored in a mobile app's code, binaries, or otherwise is generally considered to be insecure.

The best practice here is to use a web backend as a middle layer between your mobile app and the authentication provider.

> **IMPORTANT**
>
> We strongly recommend against using older mobile-only authentication libraries and patterns which do not leverage a web backend in the authentication flow due to their inherent lack of security for storing client secrets.

Get started

To start using this API, read the getting started guide for Xamarin.Essentials to ensure the library is properly installed and set up in your projects.

To access the **WebAuthenticator** functionality the following platform specific setup is required.

- Android
- iOS
- UWP

Android requires an Intent Filter setup to handle your callback URI. This is easily accomplished by subclassing the `WebAuthenticatorCallbackActivity` class:

```
const string CALLBACK_SCHEME = "myapp";

[Activity(NoHistory = true, LaunchMode = LaunchMode.SingleTop)]
[IntentFilter(new[] { Android.Content.Intent.ActionView },
    Categories = new[] { Android.Content.Intent.CategoryDefault, Android.Content.Intent.CategoryBrowsable },
    DataScheme = CALLBACK_SCHEME)]
public class WebAuthenticationCallbackActivity : Xamarin.Essentials.WebAuthenticatorCallbackActivity
{
}
```

If your project's Target Android version is set to **Android 11 (R API 30)** you must update your Android Manifest with queries that are used with the new package visibility requirements.

Open the **AndroidManifest.xml** file under the Properties folder and add the following inside of the manifest node:

```
<queries>
    <intent>
        <action android:name="android.support.customtabs.action.CustomTabsService" />
    </intent>
</queries>
```

Using WebAuthenticator

Add a reference to Xamarin.Essentials in your class:

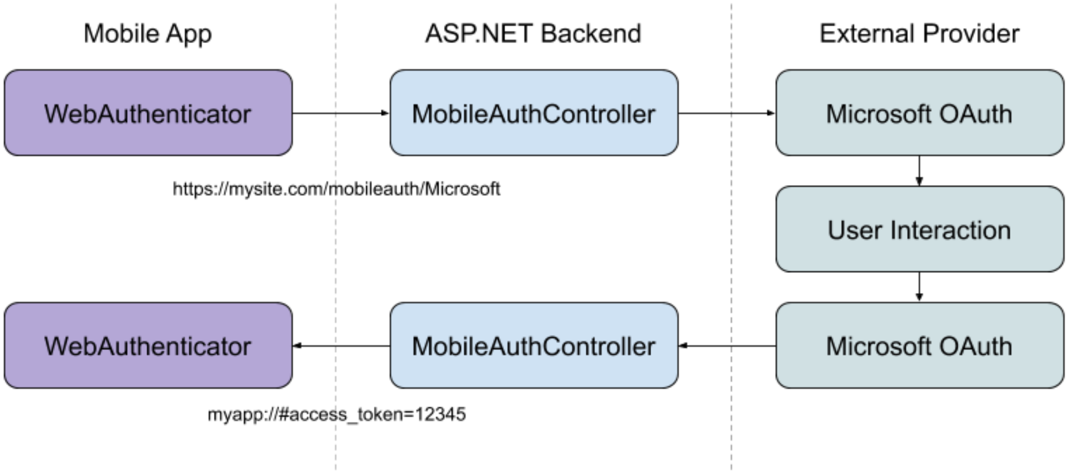
```
using Xamarin.Essentials;
```

The API consists mainly of a single method `AuthenticateAsync` which takes two parameters: The url which should be used to start the web browser flow, and the Uri which you expect the flow to ultimately call back to and which your app is registered to be able to handle.

The result is a `WebAuthenticatorResult` which includes any query parameters parsed from the callback URI:

```
var authResult = await WebAuthenticator.AuthenticateAsync(
        new Uri("https://mysite.com/mobileauth/Microsoft"),
        new Uri("myapp://"));

var accessToken = authResult?.AccessToken;
```

The `WebAuthenticator` API takes care of launching the url in the browser and waiting until the callback is received:



If the user cancels the flow at any point, a `TaskCanceledException` is thrown.

Private authentication session

iOS 13 introduced an ephemeral web browser API for developers to launch the authentication session as private. This enables developers to request that no shared cookies or browsing data is available between authentication sessions and will be a fresh login session each time. This is available through the new `WebAuthenticatorOptions` that was introduced in Xamarin.Essentials 1.7 for iOS.

```
var url = new Uri("https://mysite.com/mobileauth/Microsoft");
var callbackUrl = new Uri("myapp://")
var authResult = await WebAuthenticator.AuthenticateAsync(new WebAuthenticatorOptions
    {
        Url = url,
        CallbackUrl = callbackUrl,
        PrefersEphemeralWebBrowserSession = true
    });
```

Platform differences

- Android
- iOS
- UWP

Custom Tabs are used whenever available, otherwise an Intent is started for the URL.

Apple Sign In

According to Apple's review guidelines, if your app uses any social login service to authenticate, it must also offer Apple Sign In as an option.

To add Apple Sign In to your apps, first you'll need to configure your app to use Apple Sign In.

For iOS 13 and higher you'll want to call the `AppleSignInAuthenticator.AuthenticateAsync()` method. This will use the native Apple Sign in API's under the hood so your users get the best experience possible on these devices. You can write your shared code to use the right API at runtime like this:

```
var scheme = "..."; // Apple, Microsoft, Google, Facebook, etc.
WebAuthenticatorResult r = null;

if (scheme.Equals("Apple")
    && DeviceInfo.Platform == DevicePlatform.iOS
    && DeviceInfo.Version.Major >= 13)
{
    // Use Native Apple Sign In API's
    r = await AppleSignInAuthenticator.AuthenticateAsync();
}
else
{
    // Web Authentication flow
    var authUrl = new Uri(authenticationUrl + scheme);
    var callbackUrl = new Uri("xamarinessentials://");

    r = await WebAuthenticator.AuthenticateAsync(authUrl, callbackUrl);
}

var authToken = string.Empty;
if (r.Properties.TryGetValue("name", out var name) && !string.IsNullOrEmpty(name))
    authToken += $"Name: {name}{Environment.NewLine}";
if (r.Properties.TryGetValue("email", out var email) && !string.IsNullOrEmpty(email))
    authToken += $"Email: {email}{Environment.NewLine}";

// Note that Apple Sign In has an IdToken and not an AccessToken
authToken += r?.AccessToken ?? r?.IdToken;
```

TIP

For non-iOS 13 devices this will start the web authentication flow, which can also be used to enable Apple Sign In on your Android and UWP devices. You can sign into your iCloud account on your iOS simulator to test Apple Sign In.

ASP.NET core server back end

It's possible to use the `WebAuthenticator` API with any web back end service. To use it with an ASP.NET core app, first you need to configure the web app with the following steps:

1. Setup your desired external social authentication providers in an ASP.NET Core web app.
2. Set the Default Authentication Scheme to `CookieAuthenticationDefaults.AuthenticationScheme` in your `.AddAuthentication()` call.
3. Use `.AddCookie()` in your Startup.cs `.AddAuthentication()` call.
4. All providers must be configured with `.SaveTokens = true;`.

```
services.AddAuthentication(o =>
    {
        o.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    })
    .AddCookie()
    .AddFacebook(fb =>
    {
        fb.AppId = Configuration["FacebookAppId"];
        fb.AppSecret = Configuration["FacebookAppSecret"];
        fb.SaveTokens = true;
    });
```

TIP

If you'd like to include Apple Sign In, you can use the `AspNet.Security.OAuth.Apple` NuGet package. You can view the full Startup.cs sample in the Essentials GitHub repository.

Add a custom mobile auth controller

With a mobile authentication flow it is usually desirable to initiate the flow directly to a provider that the user has chosen (e.g. by clicking a "Microsoft" button on the sign in screen of the app). It is also important to be able to return relevant information to your app at a specific callback URI to end the authentication flow.

To achieve this, use a custom API Controller:

```
[Route("mobileauth")]
[ApiController]
public class AuthController : ControllerBase
{
    const string callbackScheme = "myapp";

    [HttpGet("{scheme}")] // eg: Microsoft, Facebook, Apple, etc
    public async Task Get([FromRoute]string scheme)
    {
        // 1. Initiate authentication flow with the scheme (provider)
        // 2. When the provider calls back to this URL
        //    a. Parse out the result
        //    b. Build the app callback URL
        //    c. Redirect back to the app
    }
}
```

The purpose of this controller is to infer the scheme (provider) that the app is requesting, and initiate the authentication flow with the social provider. When the provider calls back to the web backend, the controller parses out the result and redirects to the app's callback URI with parameters.

Sometimes you may want to return data such as the provider's `access_token` back to the app which you can do via the callback URI's query parameters. Or, you may want to instead create your own identity on your server and pass back your own token to the app. What and how you do this part is up to you!

Check out the full controller sample in the Essentials repository.

NOTE

The above sample demonstrates how to return the Access Token from the 3rd party authentication (ie: OAuth) provider. To obtain a token you can use to authorize web requests to the web backend itself, you should create your own token in your web app, and return that instead. The Overview of ASP.NET Core authentication has more information about advanced authentication scenarios in ASP.NET Core.

API

- WebAuthenticator source code
- WebAuthenticator API documentation
- ASP.NET Core Server Sample

Xamarin.Essentials: Troubleshooting

11/2/2020 • 2 minutes to read • Edit Online

Error: Version conflict detected for Xamarin.Android.Support.Compat

The following error may occur when updating NuGet packages (or adding a new package) with a Xamarin.Forms project that uses Xamarin.Essentials:

```
NU1107: Version conflict detected for Xamarin.Android.Support.Compat. Reference the package directly from
the project to resolve this issue.
 MyApp -> Xamarin.Essentials 1.3.1 -> Xamarin.Android.Support.CustomTabs 28.0.0.3 ->
Xamarin.Android.Support.Compat (= 28.0.0.3)
 MyApp -> Xamarin.Forms 3.1.0.583944 -> Xamarin.Android.Support.v4 25.4.0.2 ->
Xamarin.Android.Support.Compat (= 25.4.0.2).
```

The problem is mismatched dependencies for the two NuGets. This can be resolved by manually adding a specific version of the dependency (in this case **Xamarin.Android.Support.Compat**) that can support both.

To do this, add the NuGet that is the source of the conflict manually, and use the **Version** list to select a specific version. Currently version 28.0.0.3 of the Xamarin.Android.Support.Compat & Xamarin.Android.Support.Core.Util NuGet will resolve this error.

Refer to this blog post for more information and a video on how to resolve the issue.

If run into any issues or find a bug please report it on the Xamarin.Essentials GitHub repository.

Data and Cloud Services

10/28/2019 • 2 minutes to read • Edit Online

Data and Cloud Services

Xamarin.Android applications often need access to data (from either a local database or from the cloud), and many of these apps consume web services implemented using a wide variety of technologies. The guides in this section examine how to access data and make use of cloud services.

Data Access

This section discusses data access in Xamarin.Android using SQLite as the database engine.

Google Messaging

Google provides both Firebase Cloud Messaging and legacy Google Cloud Messaging services for facilitating messaging between mobile apps and server applications. This section provides overviews for each service provided by step-by-step explanation of how to use these services to implement remote notifications (also called push notifications) in Xamarin.Android applications.

Microsoft Azure Active Directory

7/12/2021 • 2 minutes to read • Edit Online

Azure Active Directory allows developers to secure resources such as files, links, and Web APIs, Office 365, and more using the same organizational account that employees use to sign in to their systems or check their emails.

Getting Started

Follow the getting started instructions to configure the Azure portal and add Active Directory authentication to your Xamarin apps.

1. Registering with Azure Active Directory on the *windowsazure.com* portal, then
2. Configure services.
3. Hook up one of the following:

Office 365

Once you have added Active Directory authentication to an app, you can also use the credentials to interact with Office 365.

Graph API

Learn how to access the Graph API using Xamarin (also covered in our blog).

Azure Active Directory

7/12/2021 • 2 minutes to read • Edit Online

Register an app to use Azure Active Directory

Azure Active Directory allows developers to secure resources such as files, links, and Web APIs using the same organizational account that employees use to sign in to their systems or check their emails.

Developing mobile applications which can authenticate with Azure Active Directory involves three steps. The first two steps are generally the same, regardless of what services you plan to use. The third step is different for each service-type:

1. Registering with Azure Active Directory on the *windowsazure.com* portal, then
2. Configure services.
3. Develop mobile apps using the services.

Examples of different services you can access include:

- Graph API
- Web API
- Office365

Conclusion

Using the steps above you can authenticate your mobile apps against Azure Active Directory. The Active Directory Authentication Library (ADAL) makes it much easier with fewer lines of code, while keeping most of the code the same and thus making it shareable across platforms.

Related Links

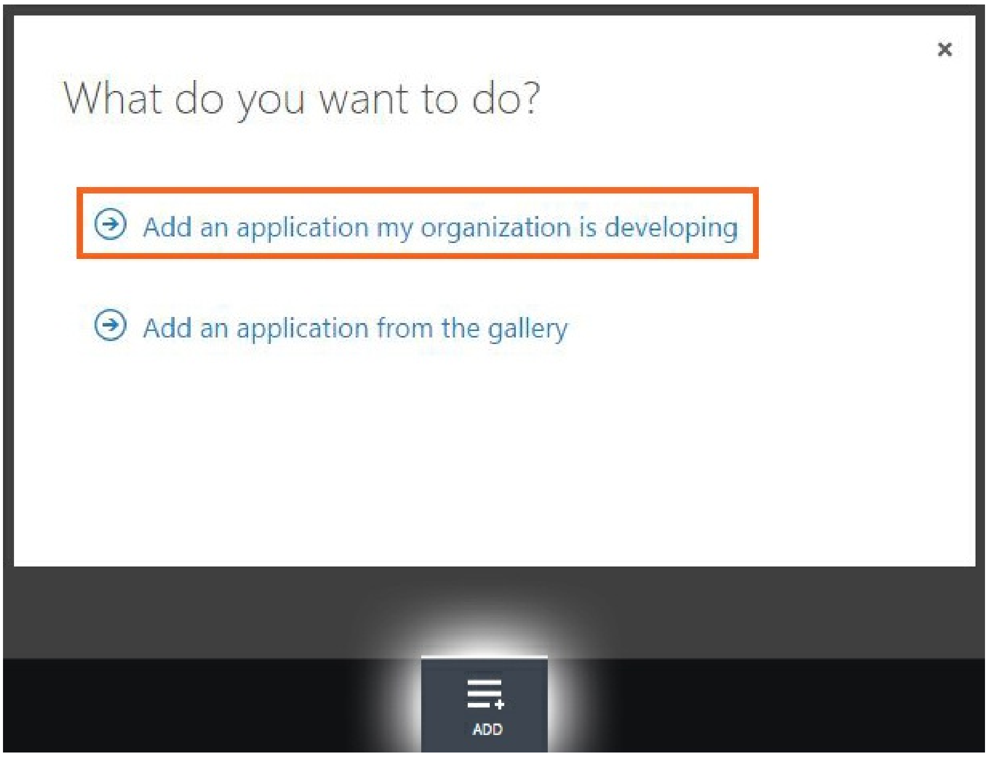
- Microsoft NativeClient sample

Step 1. Register an app to use Azure Active Directory

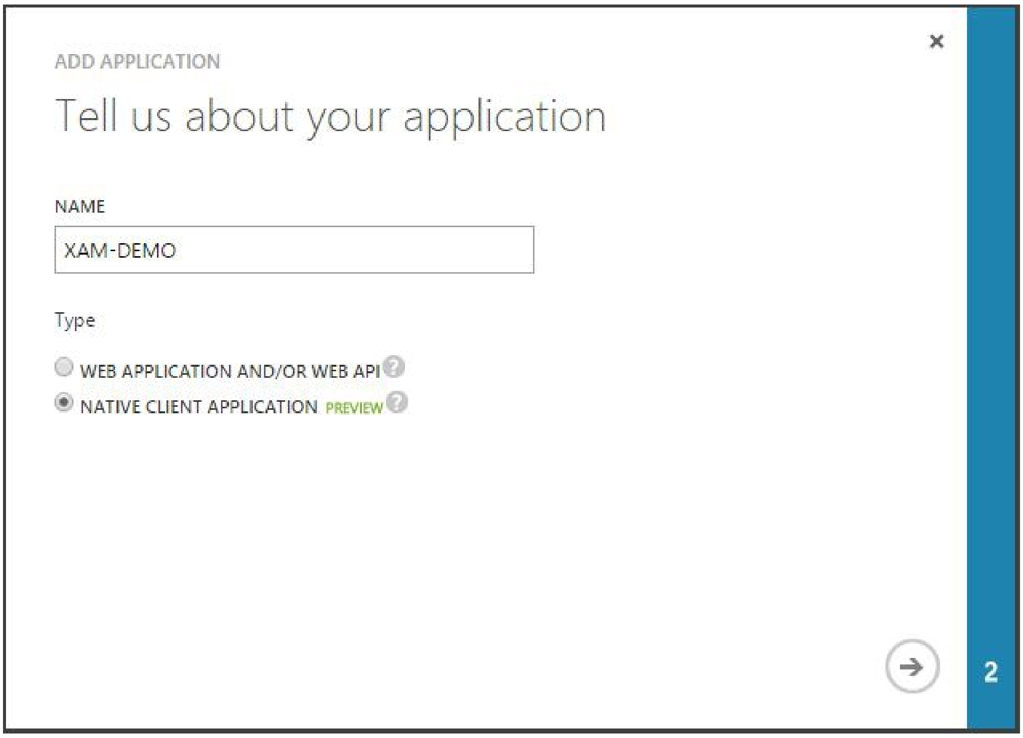
7/12/2021 • 2 minutes to read • Edit Online

1. Navigate to windowsazure.com and log in with your Microsoft Account or Organization Account in the Azure Portal. If you don't have an Azure subscription, you can get a trial from azure.com
2. After signing in, go to the **Active Directory** (1) section and choose the directory where you want to register the application (2)

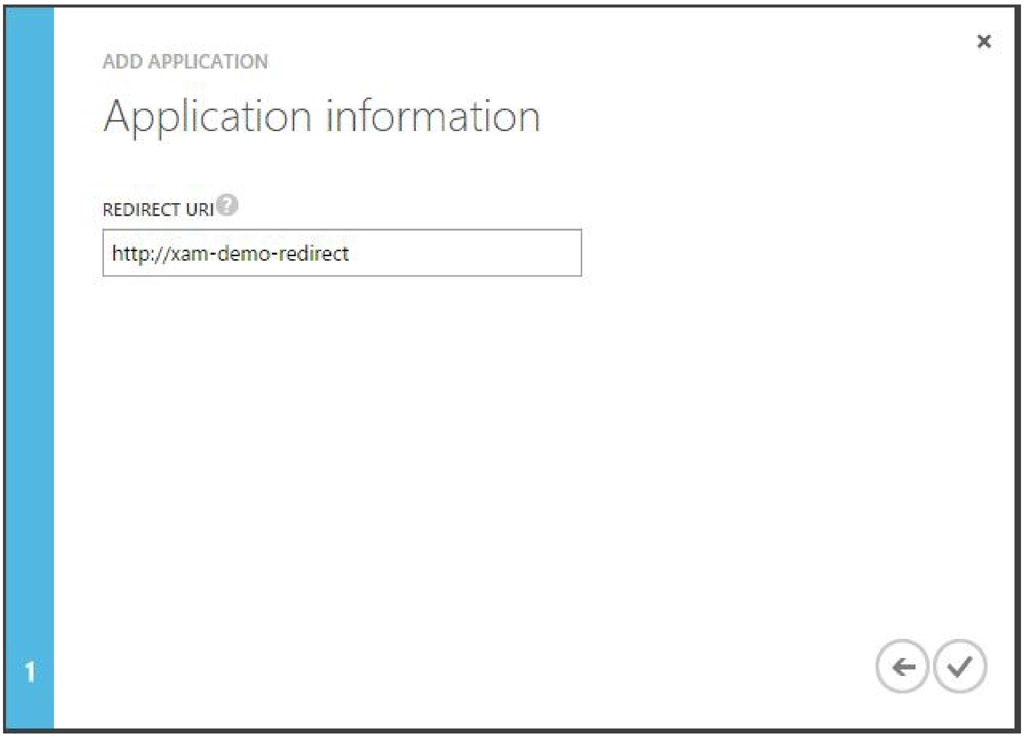
3. Click **Add** to create new application, then select **Add an application my organization is developing**



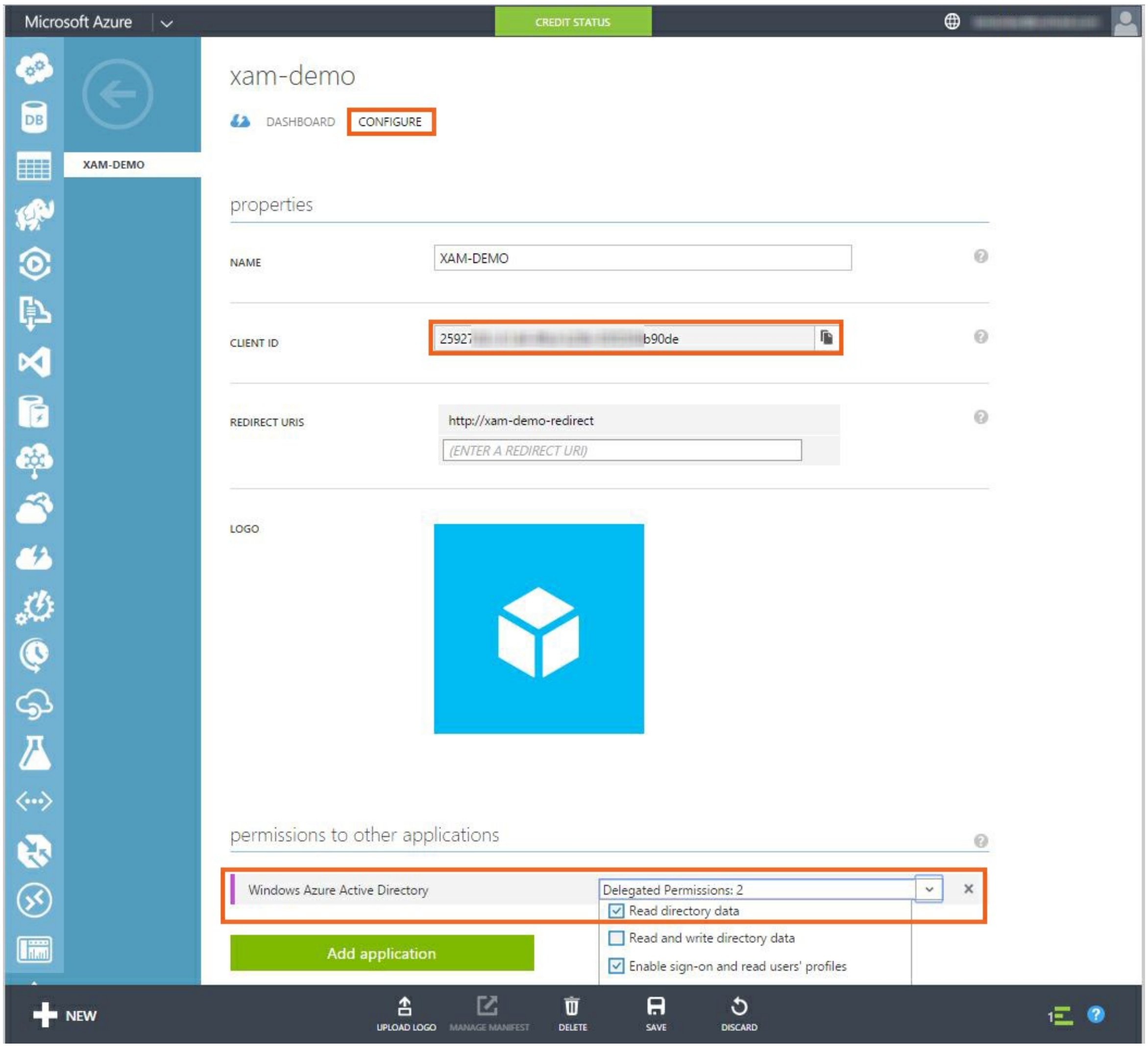
4. On the next screen, give your app a name (eg. XAM-DEMO). Make sure you select **Native Client Application** as the type of application.



5. On the final screen, provide a **Redirect URI* that is unique to your application as it will return to this URI when authentication is complete.



6. Once the app is created, navigate to the **Configure** tab. Write down the **Client ID** which we'll use in our application later. Also, on this screen you can give your mobile application access to Active Directory or add another application like Web API or Office365, which can be used by mobile application once authentication is complete.



Related Links

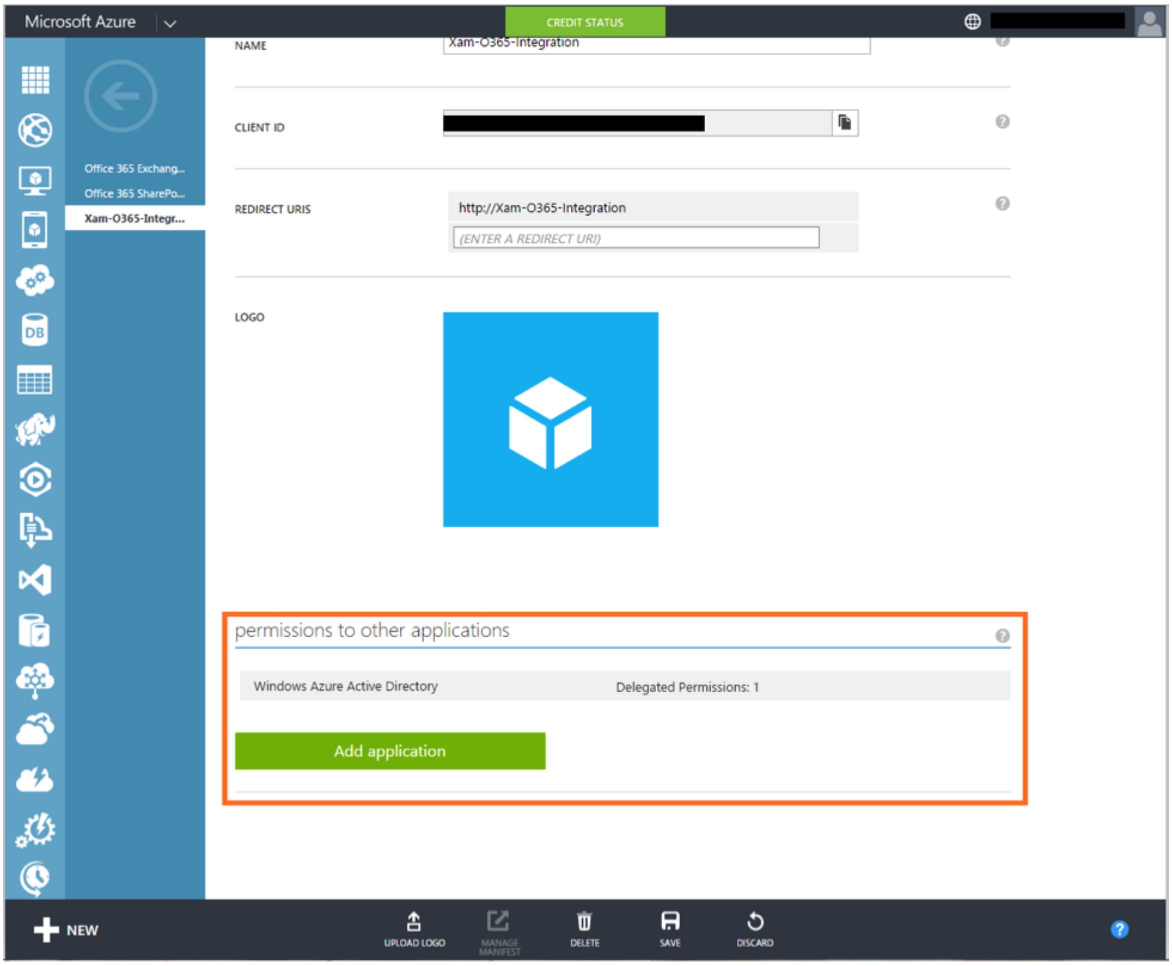
- Microsoft NativeClient sample

Step 2. Configure Service Access for Mobile Application

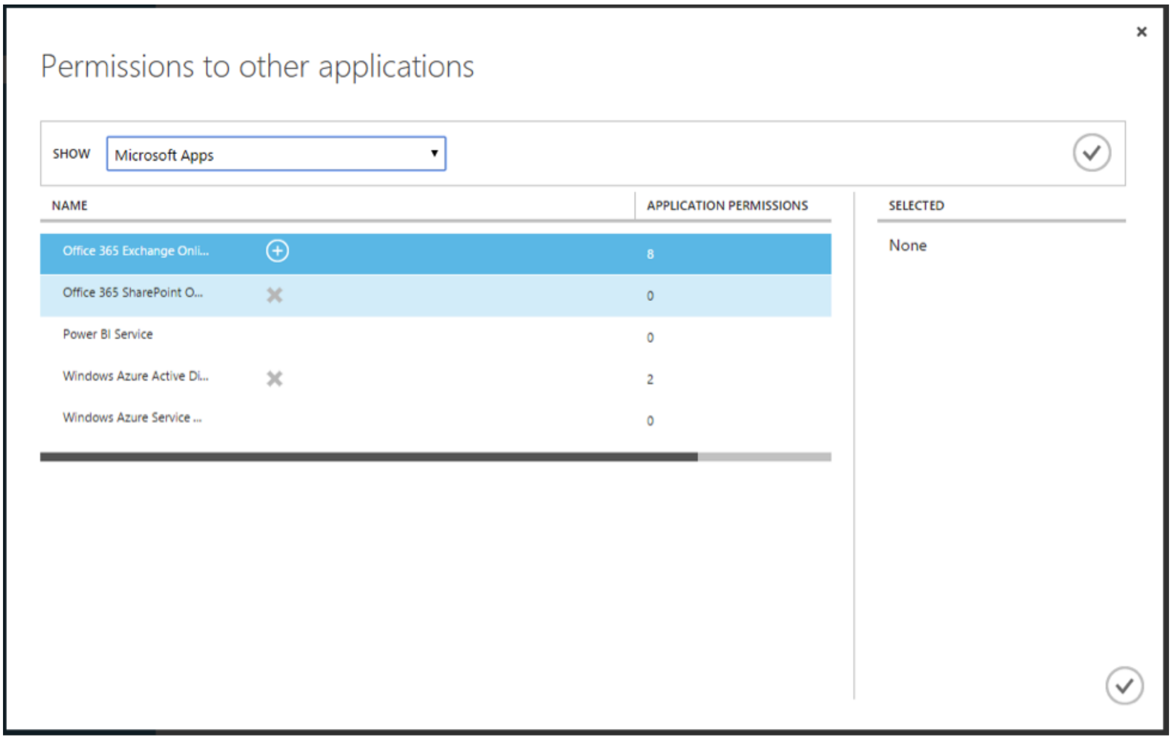
7/12/2021 • 2 minutes to read • Edit Online

Whenever any resource e.g. web application, web service, etc. needs to be secured by Azure Active Directory, it needs to be registered. All the secure applications or services can be seen under **Applications** tab. Here you can select the application which needs to be accessed from mobile application and give access to it.

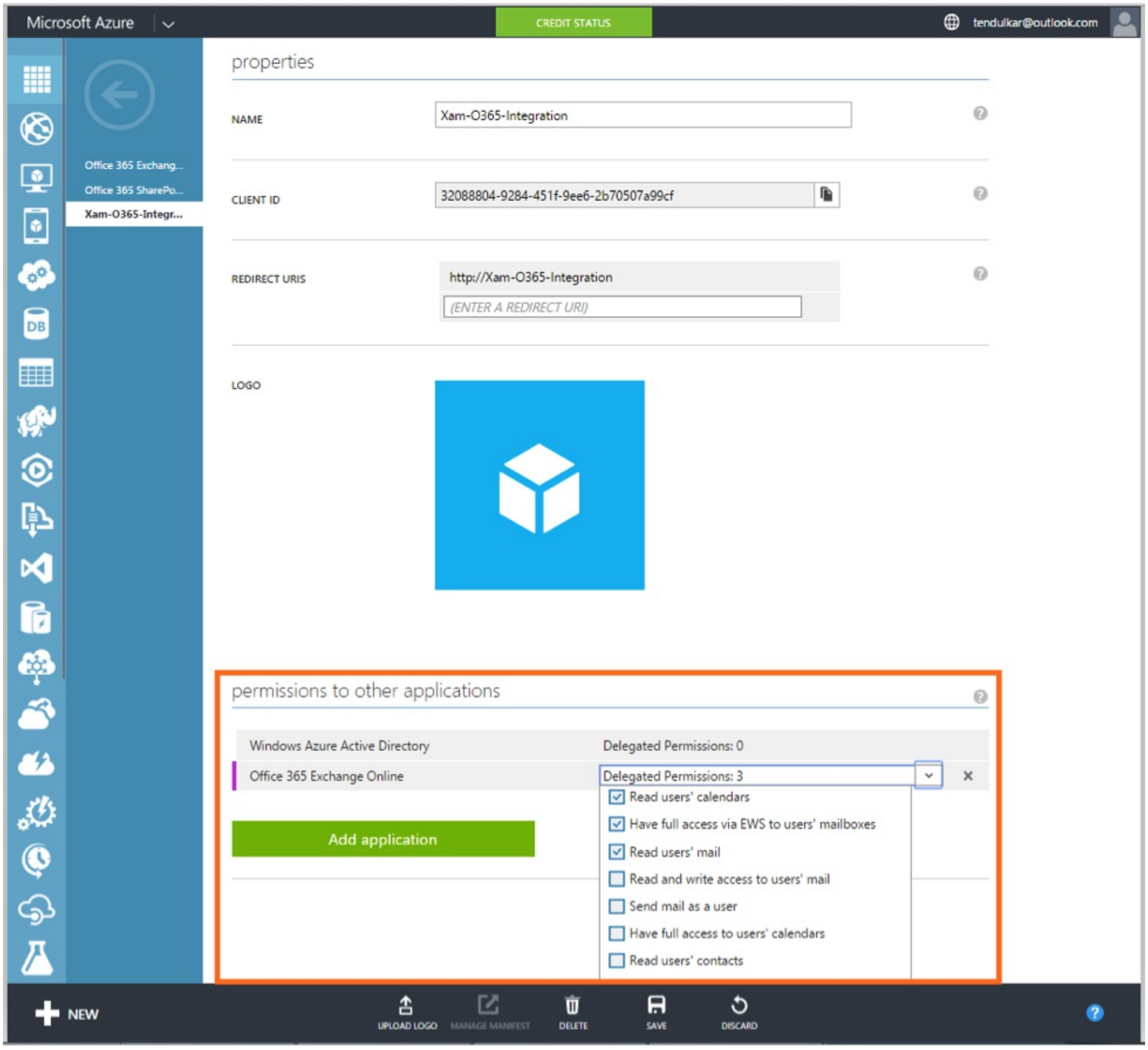
1. On the **Configure** tab, locate **permissions to other applications** section:



2. Click on **Add application** button. On the next screen pop-up you should see list of all the applications which are secured by Azure Active Directory. Select the applications that needs to be accessed from the mobile application.



3. After selecting the application, once again select the newly-added application in **permissions to other applications** section and give appropriate rights.



4. Finally, **Save** the configuration. These services should now be available in mobile applications!

Related Links

- Microsoft NativeClient sample

Accessing the Graph API

7/12/2021 • 3 minutes to read • Edit Online

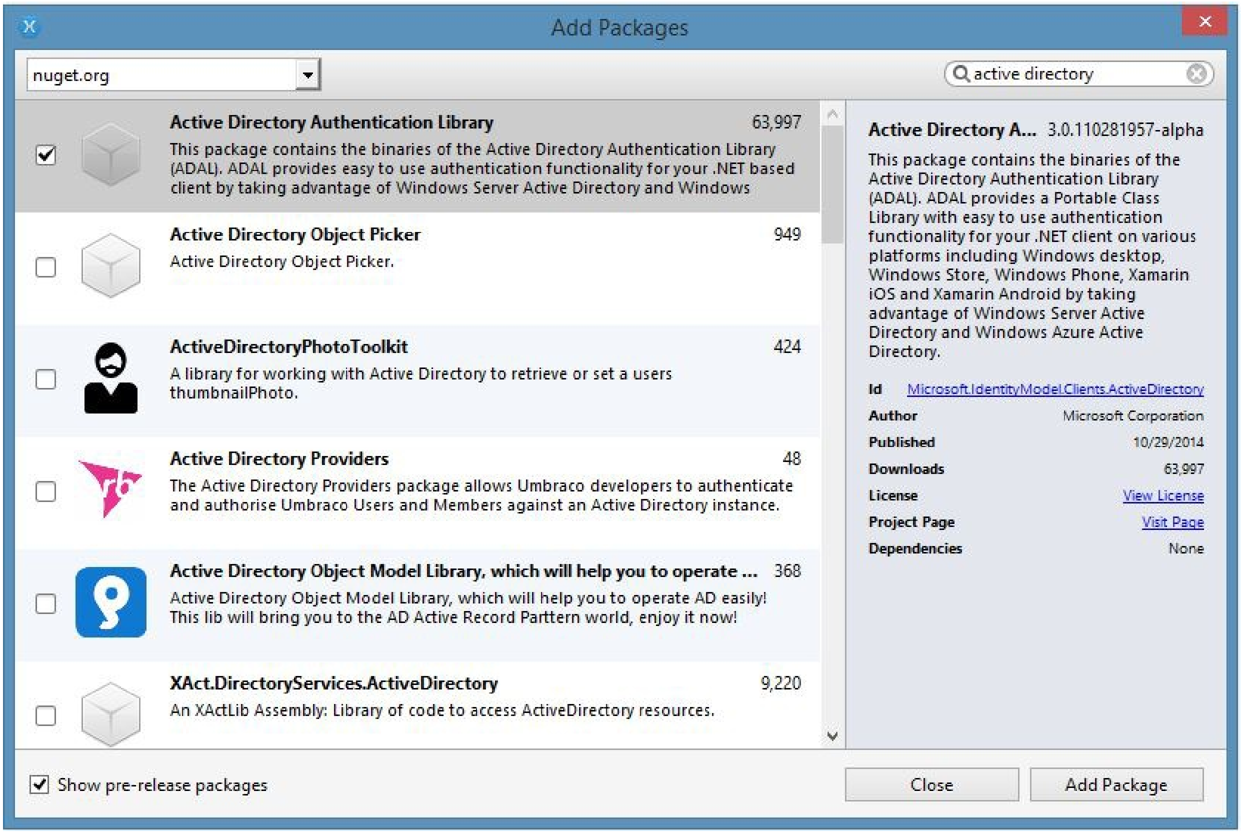
Follow these steps to use the Graph API from within a Xamarin application:

1. Registering with Azure Active Directory on the *windowsazure.com* portal, then
2. Configure services.

Step 3. Adding Active Directory authentication to an app

In your application, add a reference to **Azure Active Directory Authentication Library (Azure ADAL)** using the NuGet Package Manager in Visual Studio or Visual Studio for Mac. Make sure you select **Show pre-release packages** to include this package, as it is still in preview.

> **IMPORTANT**
>
> Note: Azure ADAL 3.0 is currently a preview and there may be breaking changes before the final version is released.



In your application, you will now need to add the following class level variables that are required for the authentication flow.

```
//Client ID
public static string clientId = "25927d3c-.....-63f2304b90de";
public static string commonAuthority = "https://login.windows.net/common"
//Redirect URI
public static Uri returnUri = new Uri("http://xam-demo-redirect");
//Graph URI if you've given permission to Azure Active Directory
const string graphResourceUri = "https://graph.windows.net";
public static string graphApiVersion = "2013-11-08";
//AuthenticationResult will hold the result after authentication completes
AuthenticationResult authResult = null;
```

One thing to note here is `commonAuthority`. When the authentication endpoint is `common`, your app becomes **multi-tenant**, which means any user can use login with their Active Directory credentials. After authentication, that user will work on the context of their own Active Directory – i.e. they will see details related to his Active Directory.

Write method to acquire Access Token

The following code (for Android) will start the authentication and upon completion assign the result in `authResult`. The iOS and Windows Phone implementations differ slightly: the second parameter (`Activity`) is different on iOS and absent on Windows Phone.

```
public static async Task<AuthenticationResult> GetAccessToken
            (string serviceResourceId, Activity activity)
{
    authContext = new AuthenticationContext(Authority);
    if (authContext.TokenCache.ReadItems().Count() > 0)
        authContext = new AuthenticationContext(authContext.TokenCache.ReadItems().First().Authority);
    var authResult = await authContext.AcquireTokenAsync(serviceResourceId, clientId, returnUri, new
AuthorizationParameters(activity));
    return authResult;
}
```

In the above code, the `AuthenticationContext` is responsible for the authentication with commonAuthority. It has an `AcquireTokenAsync` method, which take parameters as a resource which needs to be accessed, in this case `graphResourceUri`, `clientId`, and `returnUri`. The app will return to the `returnUri` when authentication completes. This code will remain the same for all platforms, however, the last parameter, `AuthorizationParameters`, will be different on different platforms and is responsible for governing the authentication flow.

In the case of Android or iOS, we pass `this` parameter to `AuthorizationParameters(this)` to share the context, whereas in Windows it is passed without any parameter as new `AuthorizationParameters()`.

Handle continuation for Android

After authentication is complete, the flow should return to the app. In the case of Android it is handled by following code, which should be added to **MainActivity.cs**:

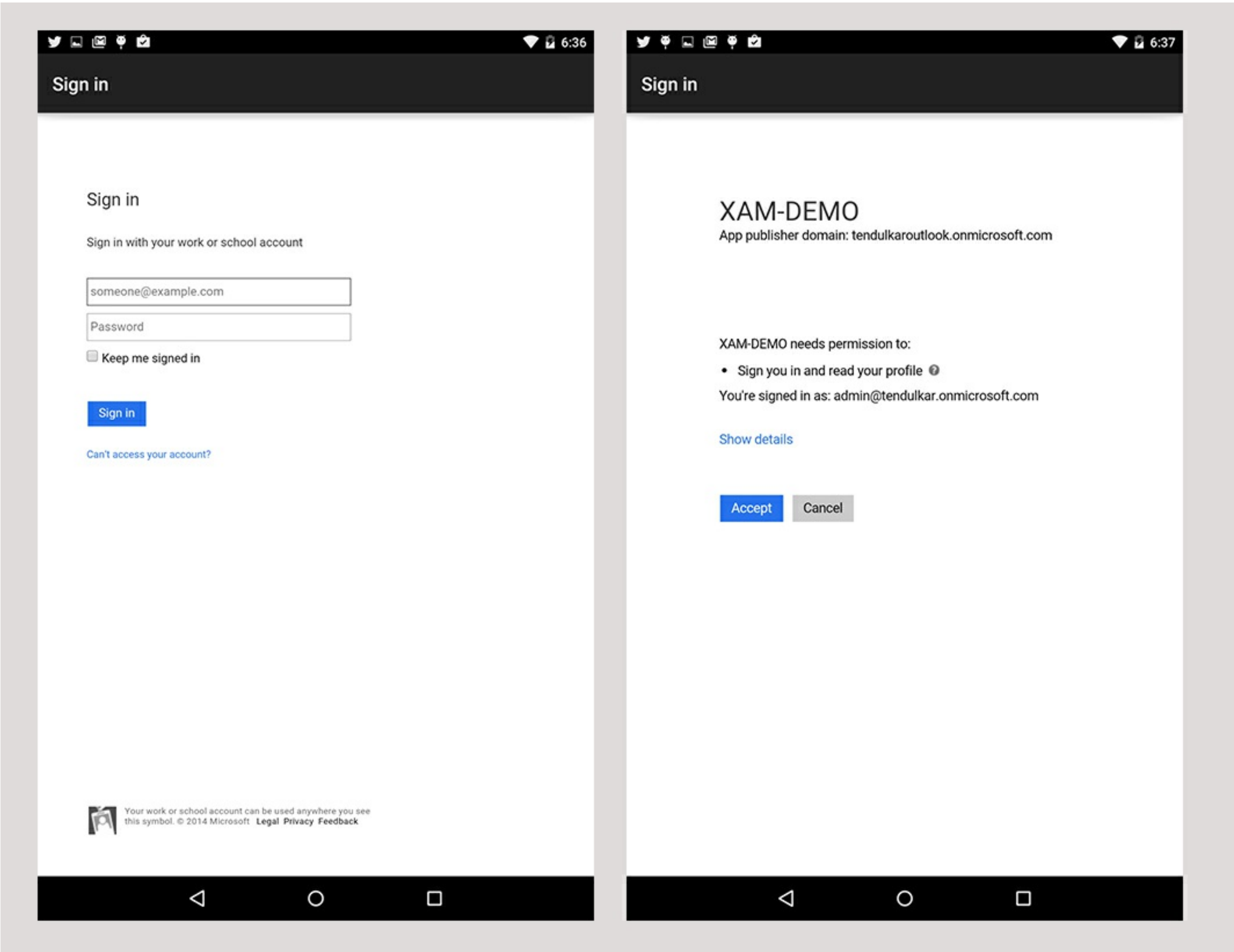
```
protected override void OnActivityResult(int requestCode, Result resultCode, Intent data)
{
  base.OnActivityResult(requestCode, resultCode, data);
  AuthenticationAgentContinuationHelper.SetAuthenticationAgentContinuationEventArgs(requestCode, resultCode,
data);
}
```

Handle continuation for Windows Phone

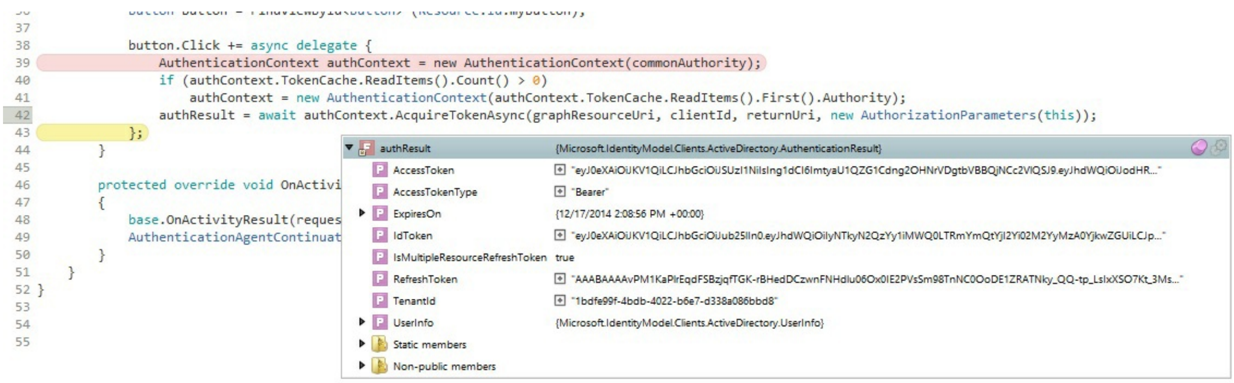
For Windows Phone modify the `OnActivated` method in the **App.xaml.cs** file with the below code:

```
protected override void OnActivated(IActivatedEventArgs args)
{
#if WINDOWS_PHONE_APP
  if (args is IWebAuthenticationBrokerContinuationEventArgs)
  {
     WebAuthenticationBrokerContinuationHelper.SetWebAuthenticationBrokerContinuationEventArgs(args as
IWebAuthenticationBrokerContinuationEventArgs);
  }
#endif
  base.OnActivated(args);
}
```

Now if you run the application, you should see an authentication dialog. Upon successful authentication, it will ask your permissions to access the resources (in our case Graph API):



If authentication is successful and you've authorized the app to access the resources, you should get an `AccessToken` and `RefreshToken` combo in `authResult`. These tokens are required for further API calls and for authorization with Azure Active Directory behind the scenes.



For example, the code below allows you to get a user list from Active Directory. You can replace the Web API URL with your Web API which is protected by Azure AD.

```
var client = new HttpClient();
var request = new HttpRequestMessage(HttpMethod.Get,
    "https://graph.windows.net/tendulkar.onmicrosoft.com/users?api-version=2013-04-05");
request.Headers.Authorization =
  new AuthenticationHeaderValue("Bearer", authResult.AccessToken);
var response = await client.SendAsync(request);
var content = await response.Content.ReadAsStringAsync();
```

Xamarin mobile apps with Azure and App Center

7/12/2021 • 2 minutes to read • Edit Online

Xamarin developers can take advantage of a wide variety of cloud services, from continuous integration build with App Center to machine learning with Azure. Download this poster or follow the links below to learn more.

App Center

Visual Studio App Center supports end to end and integrated services central to mobile app development. Developers can use **Build**, **Test** and **Distribute** services to set up a Continuous Integration and Delivery pipeline. Once the app is deployed, developers can monitor the status and usage of their app using the **Analytics** and **Diagnostics** services, and engage with users using the **Push** service. Developers can also leverage **Auth** to authenticate their users and the **Data** service to persist and sync app data in the cloud.

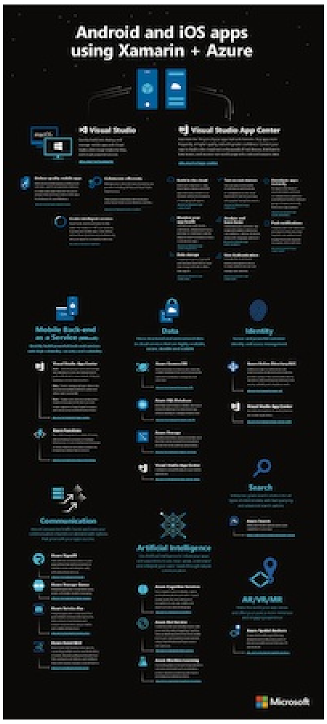
If you are looking to integrate cloud services in your Xamarin apps, visit the docs and sign up with App Center today.

Azure

Learn about mobile app development with cloud services, including SignalR, cognitive services, machine learning, spatial anchors, search, and more.

Download the poster

Download this PDF (180kb) reference of the most popular Azure and App Center services available for mobile app development with Xamarin:



Xamarin.Android Data Access

10/28/2019 • 2 minutes to read • Edit Online

Most applications have some requirement to save data on the device locally. Unless the amount of data is trivially small, this usually requires a database and a data layer in the application to manage database access. Android has the SQLite database engine 'built in' and access to store and retrieve data is simplified by Xamarin's platform. This document shows how to access an SQLite database in a cross-platform way.

Data Access Overview

Most applications have some requirement to save data on the device locally. Unless the amount of data is trivially small, this usually requires a database and a data layer in the application to manage database access. Android both has the SQLite database engine "built in" and access to the data is simplified by Xamarin's platform which comes with the SQLite Data Provider.

Xamarin.Android support database access APIs such as:

- ADO.NET framework.
- SQLite-NET 3rd party library.

The majority of the code in this section is completely cross-platform and will run on iOS or Android without modification. There are two sample apps discussed:

- DataAccess_Basic – Simple data operations writes the results to a text display control;
- DataAccess_Advanced – Integrates data operations into a small working application that lists and edits a simple data structure.

Both sample solutions contain iOS and Android sample application projects.

For Xamarin.Forms applications, read working with databases which explains how to work with SQLite in a PCL library with Xamarin.Forms.

The topics in this section discuss data access in Xamarin.Android using SQLite as the database engine. The database can be accessed "directly" by using ADO.NET syntax or you can include the SQLite.NET ORM and perform data operations in C#.

Two samples are reviewed: one that contains very simple data access code that outputs to a text field, and a simple application that includes create, read, update and delete functionality. Threading and how to seed your application with a pre-populated SQLite database is also discussed.

For additional examples of cross-platform data access see our Tasky Pro case study.

Related Links

- DataAccess Basic (sample)
- DataAccess Advanced (sample)
- Android Data Recipes
- Xamarin.Forms data access

Introduction

10/29/2019 • 3 minutes to read • Edit Online

When to use a Database

While the storage and processing capabilities of mobile devices are increasing, phones and tablets still lag behind their desktop and laptop counterparts. For this reason it is worth taking some time to plan the data storage architecture for your app rather than just assuming a database is the right answer all the time. There are a number of different options that suit different requirements, such as:

- **Preferences** – Android offers a built-in mechanism for storing simple key-value pairs of data. If you are storing simple user settings or small pieces of data (such as personalization information) then use the platform's native features for storing this type of information.
- **Text Files** – User input or caches of downloaded content (eg. HTML) can be stored directly on the file-system. Use an appropriate file-naming convention to help you organize the files and find data.
- **Serialized Data Files** – Objects can be persisted as XML or JSON on the file-system. The .NET framework includes libraries that make serializing and de-serializing objects easy. Use appropriate names to organize data files.
- **Database** – The SQLite database engine is available on the Android platform, and is useful for storing structured data that you need to query, sort or otherwise manipulate. Database storage is suited to lists of data with many properties.
- **Image files** – Although it's possible to store binary data in the database on a mobile device, it is recommended that you store them directly in the file-system. If necessary you can store the filenames in a database to associate the image with other data. When dealing with large images, or lots of images, it is good practice to plan a caching strategy that deletes files you no longer need to avoid consuming all the user's storage space.

If a database is the right storage mechanism for your app, the remainder of this document discusses how to use SQLite on the Xamarin platform.

Advantages of using a Database

There are a number of advantages to using an SQL database in your mobile app:

- SQL databases allow efficient storage of structured data.
- Specific data can be extracted with complex queries.
- Query results can be sorted.
- Query results can be aggregated.
- Developers with existing database skills can utilize their knowledge to design the database and data access code.
- The data model from the server component of a connected application may be re-used (in whole or in part) in the mobile application.

SQLite Database Engine

SQLite is an open-source database engine that has been adopted by Google for their mobile platform. The SQLite database engine is built-in to both operating systems so there is no additional work for developers to take advantage of it. SQLite is well suited to cross-platform mobile development because:

- The database engine is small, fast and easily portable.

- A database is stored in a single file, which is easy to manage on mobile devices.
- The file format is easy to use across platforms: whether 32- or 64-bit, and big- or little-endian systems.
- It implements most of the SQL92 standard.

Because SQLite is designed to be small and fast, there are some caveats on its use:

- Some OUTER join syntax is not supported.
- Only table RENAME and ADDCOLUMN are supported. You cannot perform other modifications to your schema.
- Views are read-only.

You can learn more about SQLite on the website - SQLite.org - however all the information you need to use SQLite with Xamarin is contained in this document and associated samples. The SQLite database engine has been supported in Android since Android 2. Although not covered in this chapter, SQLite is also available for use on Windows Phone and Windows applications.

Windows and Windows Phone

SQLite can also be used on Windows platforms, although those platforms are not covered in this document. Read more in the Tasky and Tasky Pro case studies, and review Tim Heuer's blog.

Related Links

- DataAccess Basic (sample)
- DataAccess Advanced (sample)
- Android Data Recipes
- Xamarin.Forms data access

Configuration

10/28/2019 • 2 minutes to read • Edit Online

To use SQLite in your Xamarin.Android application you will need to determine the correct file location for your database file.

Database File Path

Regardless of which data access method you use, you must create a database file before data can be stored with SQLite. Depending on what platform you are targeting the file location will be different. For Android you can use Environment class to construct a valid path, as shown in the following code snippet:

```
string dbPath = Path.Combine (
        Environment.GetFolderPath (Environment.SpecialFolder.Personal),
        "database.db3");
// dbPath contains a valid file path for the database file to be stored
```

There are other things to take into consideration when deciding where to store the database file. For example, on Android you can choose whether to use internal or external storage.

If you wish to use a different location on each platform in your cross platform application you can use a compiler directive as shown to generate a different path for each platform:

```
var sqliteFilename = "MyDatabase.db3";
#if __ANDROID__
// Just use whatever directory SpecialFolder.Personal returns
string libraryPath = Environment.GetFolderPath(Environment.SpecialFolder.Personal); ;
#else
// we need to put in /Library/ on iOS5.1 to meet Apple's iCloud terms
// (they don't want non-user-generated data in Documents)
string documentsPath = Environment.GetFolderPath (Environment.SpecialFolder.Personal); // Documents folder
string libraryPath = Path.Combine (documentsPath, "..", "Library"); // Library folder instead
#endif
var path = Path.Combine (libraryPath, sqliteFilename);
```

For hints on using the file system in Android, refer to the Browse Files recipe. See the Building Cross Platform Applications document for more information on using compiler directives to write code specific to each platform.

Threading

You should not use the same SQLite database connection across multiple threads. Be careful to open, use and then close any connections you create on the same thread.

To ensure that your code is not attempting to access the SQLite database from multiple threads at the same time, manually take a lock whenever you are going to access the database, like this:

```
object locker = new object(); // class level private field
// rest of class code
lock (locker){
    // Do your query or insert here
}
```

All database access (reads, writes, updates, etc.) should be wrapped with the same lock. Care must be taken to avoid a deadlock situation by ensuring that the work inside the lock clause is kept simple and does not call out to other methods that may also take a lock!

Related Links

- DataAccess Basic (sample)
- DataAccess Advanced (sample)
- Android Data Recipes
- Xamarin.Forms data access

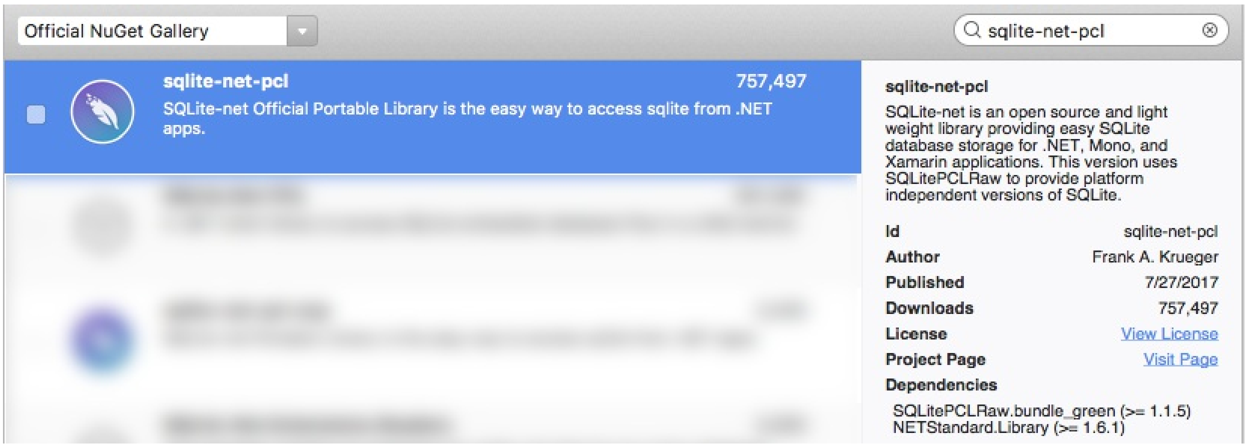
Using SQLite.NET with Android

7/8/2021 • 6 minutes to read • Edit Online

The SQLite.NET library that Xamarin recommends is a very basic ORM that lets you easily store and retrieve objects in the local SQLite database on an Android device. ORM stands for Object Relational Mapping – an API that lets you save and retrieve "objects" from a database without writing SQL statements.

To include the SQLite.NET library in a Xamarin app, add the following NuGet package to your project:

- **Package Name:** sqlite-net-pcl
- **Author:** Frank A. Krueger
- **Id:** sqlite-net-pcl
- **Url:** nuget.org/packages/sqlite-net-pcl



> **TIP**
> There are a number of different SQLite packages available – be sure to choose the correct one (it might not be the top result in search).

Once you have the SQLite.NET library available, follow these three steps to use it to access a database:

1. **Add a using statement** – Add the following statement to the C# files where data access is required:

```
using SQLite;
```

2. **Create a Blank Database** – A database reference can be created by passing the file path the SQLiteConnection class constructor. You do not need to check if the file already exists – it will automatically be created if required, otherwise the existing database file will be opened. The `dbPath` variable should be determined according the rules discussed earlier in this document:

```
var db = new SQLiteConnection (dbPath);
```

3. **Save Data** – Once you have created a SQLiteConnection object, database commands are executed by calling its methods, such as CreateTable and Insert like this:

```
db.CreateTable<Stock> ();
db.Insert (newStock); // after creating the newStock object
```

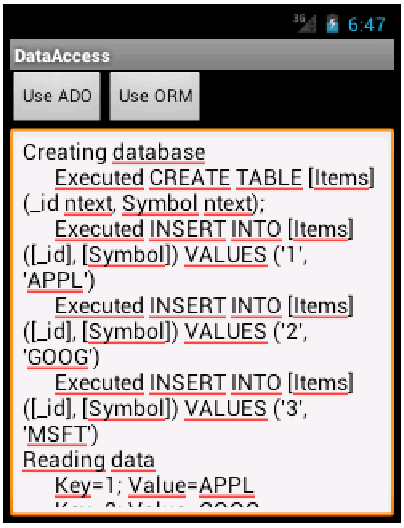
4. **Retrieve Data** – To retrieve an object (or a list of objects) use the following syntax:

```
var stock = db.Get<Stock>(5); // primary key id of 5
var stockList = db.Table<Stock>();
```

Basic Data Access Sample

The *DataAccess_Basic* sample code for this document looks like this when running on Android. The code illustrates how to perform simple SQLite.NET operations and shows the results in as text in the application's main window.

Android



The following code sample shows an entire database interaction using the SQLite.NET library to encapsulate the underlying database access. It shows:

1. Creating the database file
2. Inserting some data by creating objects and then saving them
3. Querying the data

You'll need to include these namespaces:

```
using SQLite; // from the github SQLite.cs class
```

The last one requires that you have added SQLite to your project. Note that the SQLite database table is defined by adding attributes to a class (the `Stock` class) rather than a CREATE TABLE command.

```
[Table("Items")]
public class Stock {
    [PrimaryKey, AutoIncrement, Column("_id")]
    public int Id { get; set; }
    [MaxLength(8)]
    public string Symbol { get; set; }
}
public static void DoSomeDataAccess () {
       Console.WriteLine ("Creating database, if it doesn't already exist");
   string dbPath = Path.Combine (
        Environment.GetFolderPath (Environment.SpecialFolder.Personal),
        "ormdemo.db3");
   var db = new SQLiteConnection (dbPath);
   db.CreateTable<Stock> ();
   if (db.Table<Stock> ().Count() == 0) {
        // only insert the data if it doesn't already exist
        var newStock = new Stock ();
        newStock.Symbol = "AAPL";
        db.Insert (newStock);
        newStock = new Stock ();
        newStock.Symbol = "GOOG";
        db.Insert (newStock);
        newStock = new Stock ();
        newStock.Symbol = "MSFT";
        db.Insert (newStock);
    }
    Console.WriteLine("Reading data");
    var table = db.Table<Stock> ();
    foreach (var s in table) {
        Console.WriteLine (s.Id + " " + s.Symbol);
    }
}
```

Using the `[Table]` attribute without specifying a table name parameter will cause the underlying database table to have the same name as the class (in this case, "Stock"). The actual table name is important if you write SQL queries directly against the database rather than use the ORM data access methods. Similarly the `[Column("_id")]` attribute is optional, and if absent a column will be added to the table with the same name as the property in the class.

SQLite Attributes

Common attributes that you can apply to your classes to control how they are stored in the underlying database include:

- **[PrimaryKey]** – This attribute can be applied to an integer property to force it to be the underlying table's primary key. Composite primary keys are not supported.
- **[AutoIncrement]** – This attribute will cause an integer property's value to be auto-increment for each new object inserted into the database
- **[Column(name)]** – The `name` parameter sets the underlying database column's name.
- **[Table(name)]** – Marks the class as being able to be stored in an underlying SQLite table with the name specified.
- **[MaxLength(value)]** – Restrict the length of a text property, when a database insert is attempted. Consuming code should validate this prior to inserting the object as this attribute is only 'checked' when a database insert or update operation is attempted.
- **[Ignore]** – Causes SQLite.NET to ignore this property. This is particularly useful for properties that have a type that cannot be stored in the database, or properties that model collections that cannot be resolved

automatically by SQLite.

- **[Unique]** – Ensures that the values in the underlying database column are unique.

Most of these attributes are optional. You should always specify an integer primary key so that selection and deletion queries can be performed efficiently on your data.

More Complex Queries

The following methods on `SQLiteConnection` can be used to perform other data operations:

- **Insert** – Adds a new object to the database.
- **Get<T>** – Attempts to retrieve an object using the primary key.
- **Table<T>** – Returns all the objects in the table.
- **Delete** – Deletes an object using its primary key.
- **Query<T>** – Perform an SQL query that returns a number of rows (as objects).
- **Execute** – Use this method (and not `Query`) when you don't expect rows back from the SQL (such as INSERT, UPDATE and DELETE instructions).

Getting an object by the primary key

SQLite.Net provides the Get method to retrieve a single object based on its primary key.

```
var existingItem = db.Get<Stock>(3);
```

Selecting an object using Linq

Methods that return collections support `IEnumerable<T>` so you can use Linq to query or sort the contents of a table. The following code shows an example using Linq to filter out all entries that begin with the letter "A":

```
var apple = from s in db.Table<Stock>()
    where s.Symbol.StartsWith ("A")
    select s;
Console.WriteLine ("-> " + apple.FirstOrDefault ().Symbol);
```

Selecting an object using SQL

Even though SQLite.Net can provide object-based access to your data, sometimes you might need to do a more complex query than Linq allows (or you may need faster performance). You can use SQL commands with the Query method, as shown here:

```
var stocksStartingWithA = db.Query<Stock>("SELECT * FROM Items WHERE Symbol = ?", "A");
foreach (var s in stocksStartingWithA) {
    Console.WriteLine ("a " + s.Symbol);
}
```

> **NOTE**
>
> When writing SQL statements directly you create a dependency on the names of tables and columns in your database, which have been generated from your classes and their attributes. If you change those names in your code you must remember to update any manually written SQL statements.

Deleting an object

The primary key is used to delete the row, as shown here:

```
var rowcount = db.Delete<Stock>(someStock.Id); // Id is the primary key
```

You can check the `rowcount` to confirm how many rows were affected (deleted in this case).

Using SQLite.NET with Multiple Threads

SQLite supports three different threading modes: *Single-thread*, *Multi-thread*, and *Serialized*. If you want to access the database from multiple threads without any restrictions, you can configure SQLite to use the **Serialized** threading mode. It's important to set this mode early in your application (for example, at the beginning of the `OnCreate` method).

To change the threading mode, call `SqliteConnection.SetConfig`. For example, this line of code configures SQLite for **Serialized** mode:

```
using using Mono.Data.Sqlite;
...
SqliteConnection.SetConfig(SQLiteConfig.Serialized);
```

The Android version of SQLite has a limitation that requires a few more steps. If the call to `SqliteConnection.SetConfig` produces a SQLite exception such as `library used incorrectly`, then you must use the following workaround:

1. Link to the native **libsqlite.so** library so that the `sqlite3_shutdown` and `sqlite3_initialize` APIs are made available to the app:

```
[DllImport("libsqlite.so")]
internal static extern int sqlite3_shutdown();

[DllImport("libsqlite.so")]
internal static extern int sqlite3_initialize();
```

2. At the very beginning of the `OnCreate` method, add this code to shutdown SQLite, configure it for **Serialized** mode, and reinitialize SQLite:

```
using using Mono.Data.Sqlite;
...
sqlite3_shutdown();
SqliteConnection.SetConfig(SQLiteConfig.Serialized);
sqlite3_initialize();
```

This workaround also works for the `Mono.Data.Sqlite` library. For more information about SQLite and multi-threading, see SQLite and Multiple Threads.

Related Links

- DataAccess Basic (sample)
- DataAccess Advanced (sample)
- Xamarin.Forms data access

Using ADO.NET with Android

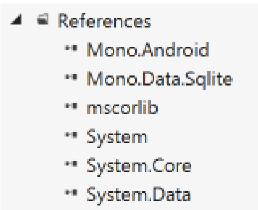
7/8/2021 • 5 minutes to read • Edit Online

Xamarin has built-in support for the SQLite database that is available on Android and can be exposed using familiar ADO.NET-like syntax. Using these APIs requires you to write SQL statements that are processed by SQLite, such as `CREATE TABLE`, `INSERT` and `SELECT` statements.

Assembly References

To use access SQLite via ADO.NET you must add `System.Data` and `Mono.Data.Sqlite` references to your Android project, as shown here:

- Visual Studio
- Visual Studio for Mac



Right-click **References > Edit References...** then click to select the required assemblies.

About Mono.Data.Sqlite

We will use the `Mono.Data.Sqlite.SqliteConnection` class to create a blank database file and then to instantiate `SqliteCommand` objects that we can use to execute SQL instructions against the database.

Creating a Blank Database – Call the `CreateFile` method with a valid (i.e. writeable) file path. You should check whether the file already exists before calling this method, otherwise a new (blank) database will be created over the top of the old one, and the data in the old file will be lost. `Mono.Data.Sqlite.SqliteConnection.CreateFile (dbPath);` The `dbPath` variable should be determined according the rules discussed earlier in this document.

Creating a Database Connection – After the SQLite database file has been created you can create a connection object to access the data. The connection is constructed with a connection string which takes the form of `Data Source=file_path`, as shown here:

```
var connection = new SqliteConnection ("Data Source=" + dbPath);
connection.Open();
// do stuff
connection.Close();
```

As mentioned earlier, a connection should never be re-used across different threads. If in doubt, create the connection as required and close it when you're done; but be mindful of doing this more often than required too.

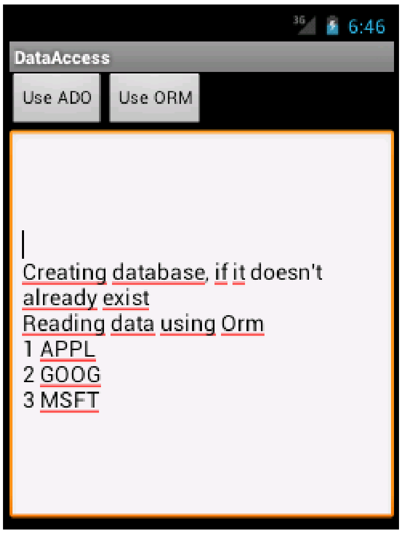
Creating and Executing a Database Command – Once we have a connection we can execute arbitrary SQL commands against it. The code below shows a `CREATE TABLE` statement being executed.

```
using (var command = connection.CreateCommand ()) {
    command.CommandText = "CREATE TABLE [Items] ([_id] int, [Symbol] ntext, [Name] ntext);";
    var rowcount = command.ExecuteNonQuery ();
}
```

When executing SQL directly against the database you should take the normal precautions not to make invalid requests, such as attempting to create a table that already exists. Keep track of the structure of your database so that you don't cause a `SqliteException` such as **SQLite error table [Items] already exists**.

Basic Data Access

The *DataAccess_Basic* sample code for this document looks like this when running on Android:



The code below illustrates how to perform simple SQLite operations and shows the results in as text in the application's main window.

You'll need to include these namespaces:

```
using System;
using System.IO;
using Mono.Data.Sqlite;
```

The following code sample shows an entire database interaction:

1. Creating the database file
2. Inserting some data
3. Querying the data

These operations would typically appear in multiple places throughout your code, for example you may create the database file and tables when your application first starts and perform data reads and writes in individual screens in your app. In the example below have been grouped into a single method for this example:

```
public static SqliteConnection connection;
public static string DoSomeDataAccess ()
{
    // determine the path for the database file
    string dbPath = Path.Combine (
        Environment.GetFolderPath (Environment.SpecialFolder.Personal),
        "adodemo.db3");

    bool exists = File.Exists (dbPath);

    if (!exists) {
        Console.WriteLine("Creating database");
        // Need to create the database before seeding it with some data
        Mono.Data.Sqlite.SqliteConnection.CreateFile (dbPath);
        connection = new SqliteConnection ("Data Source=" + dbPath);

        var commands = new[] {
            "CREATE TABLE [Items] (_id ntext, Symbol ntext);",
            "INSERT INTO [Items] ([_id], [Symbol]) VALUES ('1', 'AAPL')",
            "INSERT INTO [Items] ([_id], [Symbol]) VALUES ('2', 'GOOG')",
            "INSERT INTO [Items] ([_id], [Symbol]) VALUES ('3', 'MSFT')"
        };
        // Open the database connection and create table with data
        connection.Open ();
        foreach (var command in commands) {
            using (var c = connection.CreateCommand ()) {
                c.CommandText = command;
                var rowcount = c.ExecuteNonQuery ();
                Console.WriteLine("\tExecuted " + command);
            }
        }
    } else {
        Console.WriteLine("Database already exists");
        // Open connection to existing database file
        connection = new SqliteConnection ("Data Source=" + dbPath);
        connection.Open ();
    }

    // query the database to prove data was inserted!
    using (var contents = connection.CreateCommand ()) {
        contents.CommandText = "SELECT [_id], [Symbol] from [Items]";
        var r = contents.ExecuteReader ();
        Console.WriteLine("Reading data");
        while (r.Read ())
            Console.WriteLine("\tKey={0}; Value={1}",
                              r ["_id"].ToString (),
                              r ["Symbol"].ToString ());
    }
    connection.Close ();
}
```

More Complex Queries

Because SQLite allows arbitrary SQL commands to be run against the data, you can perform whatever `CREATE`, `INSERT`, `UPDATE`, `DELETE`, or `SELECT` statements you like. You can read about the SQL commands supported by SQLite at the SQLite website. The SQL statements are run using one of three methods on an `SqliteCommand` object:

- **ExecuteNonQuery** – Typically used for table creation or data insertion. The return value for some operations is the number of rows affected, otherwise it's -1.
- **ExecuteReader** – Used when a collection of rows should be returned as a `SqlDataReader`.

- **ExecuteScalar** – Retrieves a single value (for example an aggregate).

EXECUTENONQUERY

`INSERT`, `UPDATE`, and `DELETE` statements will return the number of rows affected. All other SQL statements will return -1.

```
using (var c = connection.CreateCommand ()) {
    c.CommandText = "INSERT INTO [Items] ([_id], [Symbol]) VALUES ('1', 'APPL')";
    var rowcount = c.ExecuteNonQuery (); // rowcount will be 1
}
```

EXECUTEREADER

The following method shows a `WHERE` clause in the `SELECT` statement. Because the code is crafting a complete SQL statement it must take care to escape reserved characters such as the quote (') around strings.

```
public static string MoreComplexQuery ()
{
    var output = "";
    output += "\nComplex query example: ";
    string dbPath = Path.Combine (
        Environment.GetFolderPath (Environment.SpecialFolder.Personal), "ormdemo.db3");

    connection = new SqliteConnection ("Data Source=" + dbPath);
    connection.Open ();
    using (var contents = connection.CreateCommand ()) {
        contents.CommandText = "SELECT * FROM [Items] WHERE Symbol = 'MSFT'";
        var r = contents.ExecuteReader ();
        output += "\nReading data";
        while (r.Read ())
            output += String.Format ("\n\tKey={0}; Value={1}",
                    r ["_id"].ToString (),
                    r ["Symbol"].ToString ());
    }
    connection.Close ();

    return output;
}
```

The `ExecuteReader` method returns a `SqliteDataReader` object. In addition to the `Read` method shown in the example, other useful properties include:

- **RowsAffected** – Count of the rows affected by the query.
- **HasRows** – Whether any rows were returned.

EXECUTESCALAR

Use this for `SELECT` statements that return a single value (such as an aggregate).

```
using (var contents = connection.CreateCommand ()) {
    contents.CommandText = "SELECT COUNT(*) FROM [Items] WHERE Symbol <> 'MSFT'";
    var i = contents.ExecuteScalar ();
}
```

The `ExecuteScalar` method's return type is `object` – you should cast the result depending on the database query. The result could be an integer from a `COUNT` query or a string from a single column `SELECT` query. Note that this is different to other `Execute` methods that return a reader object or a count of the number of rows affected.

Related Links

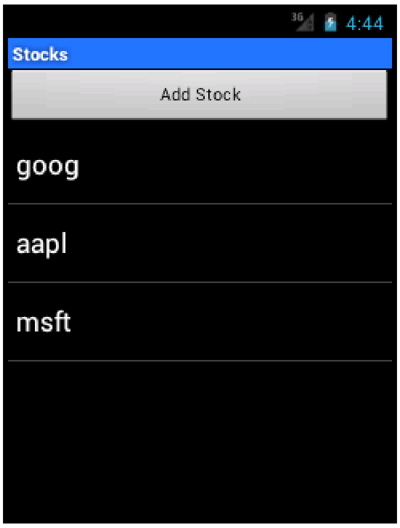
- DataAccess Basic (sample)
- DataAccess Advanced (sample)
- Android Data Recipes
- Xamarin.Forms data access

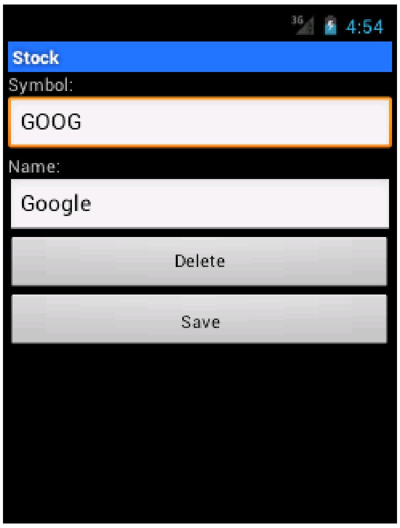
Using Data in an App

7/8/2021 • 3 minutes to read • Edit Online

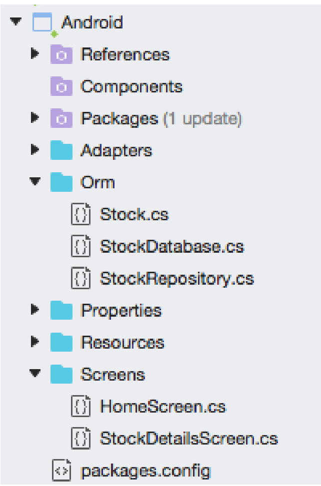
The **DataAccess_Adv** sample shows a working application that allows user-input and CRUD (Create, Read, Update and Delete) database functionality. The application consists of two screens: a list and a data entry form. All the data access code is re-usable in iOS and Android without modification.

After adding some data the application screens look like this on Android:





The Android Project is shown below – the code shown in this section is contained within the **Orm** directory:



The native UI code for the Activities in Android is out of scope for this document. Refer to the Android ListViews and Adapters guide for more information on the UI controls.

Read

There are a couple of read operations in the sample:

- Reading the list
- Reading individual records

The two methods in the `StockDatabase` class are:

```
public IEnumerable<Stock> GetStocks ()
{
    lock (locker) {
        return (from i in Table<Stock> () select i).ToList ();
    }
}
public Stock GetStock (int id)
{
    lock (locker) {
        return Table<Stock>().FirstOrDefault(x => x.Id == id);
    }
}
```

Android renders the data as a `ListView`.

Create and Update

To simplify the application code, a single save method is provided that does an Insert or Update depending on whether the PrimaryKey has been set. Because the `Id` property is marked with a `[PrimaryKey]` attribute you should not set it in your code. This method will detect whether the value has been previous saved (by checking the primary key property) and either insert or update the object accordingly:

```
public int SaveStock (Stock item)
{
    lock (locker) {
        if (item.Id != 0) {
            Update (item);
            return item.Id;
    } else {
            return Insert (item);
        }
    }
}
```

Real world applications will usually require some validation (such as required fields, minimum lengths or other business rules). Good cross-platform applications implement as much of the validation logical as possible in shared code, passing validation errors back up to the UI for display according to the platform's capabilities.

Delete

Unlike the `Insert` and `Update` methods, the `Delete<T>` method can accept just the primary key value rather than a complete `Stock` object. In this example a `Stock` object is passed into the method but only the Id property is passed on to the `Delete<T>` method.

```
public int DeleteStock(Stock stock)
{
    lock (locker) {
        return Delete<Stock> (stock.Id);
    }
}
```

Using a pre-populated SQLite database file

Some applications are shipped with a database already populated with data. You can easily accomplish this in your mobile application by shipping an existing SQLite database file with your app and copying it to a writable directory before accessing it. Because SQLite is a standard file format that is used on many platforms, there are a number of tools available to create an SQLite database file:

- **SQLite Manager Firefox Extension** – Works on Mac and Windows and produces files that are compatible with iOS and Android.
- **Command Line** – See www.sqlite.org/sqlite.html .

When creating a database file for distribution with your app, take care with the naming of tables and columns to ensure they match what your code expects, especially if you're using SQLite.NET which will expect the names to match your C# classes and properties (or the associated custom attributes).

To ensure that some code runs before anything else in your Android app, you can place it in the first activity to load or you can create an `Application` subclass that is loaded before any activities. The code below shows an `Application` subclass that copies an existing database file **data.sqlite** out of the **/Resources/Raw/** directory.

```
[Application]
public class YourAndroidApp : Application {
    public override void OnCreate ()
    {
        base.OnCreate ();
        var docFolder = Environment.GetFolderPath(Environment.SpecialFolder.Personal);
        Console.WriteLine ("Data path:" + Database.DatabaseFilePath);
        var dbFile = Path.Combine(docFolder, "data.sqlite"); // FILE NAME TO USE WHEN COPIED
        if (!System.IO.File.Exists(dbFile)) {
            var s = Resources.OpenRawResource(Resource.Raw.data);  // DATA FILE RESOURCE ID
            FileStream writeStream = new FileStream(dbFile, FileMode.OpenOrCreate, FileAccess.Write);
            ReadWriteStream(s, writeStream);
        }
    }
    // readStream is the stream you need to read
    // writeStream is the stream you want to write to
    private void ReadWriteStream(Stream readStream, Stream writeStream)
    {
        int Length = 256;
        Byte[] buffer = new Byte[Length];
        int bytesRead = readStream.Read(buffer, 0, Length);
        // write the required bytes
        while (bytesRead > 0)
        {
            writeStream.Write(buffer, 0, bytesRead);
            bytesRead = readStream.Read(buffer, 0, Length);
        }
        readStream.Close();
        writeStream.Close();
    }
}
```

Related Links

- DataAccess Basic (sample)
- DataAccess Advanced (sample)
- Android Data Recipes
- Xamarin.Forms data access

Google Messaging

10/28/2019 • 2 minutes to read • Edit Online

This section contains guides that describe how to implement Xamarin.Android apps using Google messaging services.

Firebase Cloud Messaging

Firebase Cloud Messaging (FCM) is a service that facilitates messaging between mobile apps and server applications. FCM is Google's successor to Google Cloud Messaging. This article provides an overview of how FCM works, and it provides a step-by-step procedure for acquiring credentials so that your app can use FCM services.

Remote Notifications with Firebase Cloud Messaging

This walkthrough provides a step-by-step explanation of how to use Firebase Cloud Messaging to implement remote notifications (also called push notifications) in a Xamarin.Android application. It illustrates how to implement the various classes that are needed for communications with Firebase Cloud Messaging (FCM), provides examples of how to configure the Android Manifest for access to FCM, and demonstrates downstream messaging using the Firebase Console.

Google Cloud Messaging

This section provides a high-level overview of how Google Cloud Messaging (GCM) routes messages between your app and an app server, and it provides a step-by-step procedure for acquiring credentials so that your app can use GCM services. (Note that GCM has been superceded by FCM.)

> **NOTE**
>
> GCM has been superceded by Firebase Cloud Messaging (FCM). GCM server and client APIs have been deprecated and will no longer be available as soon as April 11th, 2019.

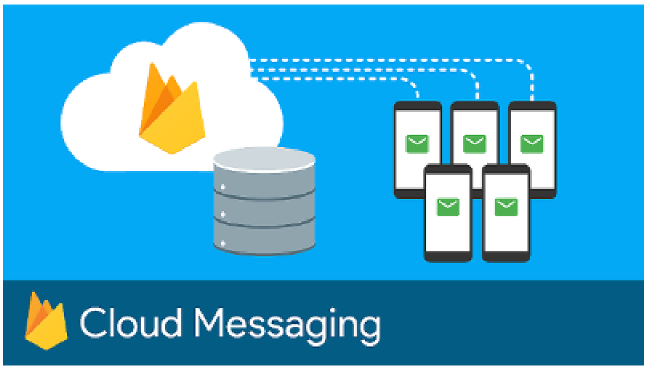
Remote Notifications with Google Cloud Messaging

This section provides a step-by-step explanation of how to implement remote notifications in Xamarin.Android using Google Cloud Messaging. It explains the various components that must be leveraged to enable Google Cloud Messaging in an Android application.

Firebase Cloud Messaging

7/8/2021 • 8 minutes to read • Edit Online

Firebase Cloud Messaging (FCM) is a service that facilitates messaging between mobile apps and server applications. This article provides an overview of how FCM works, and it explains how to configure Google Services so that your app can use FCM.

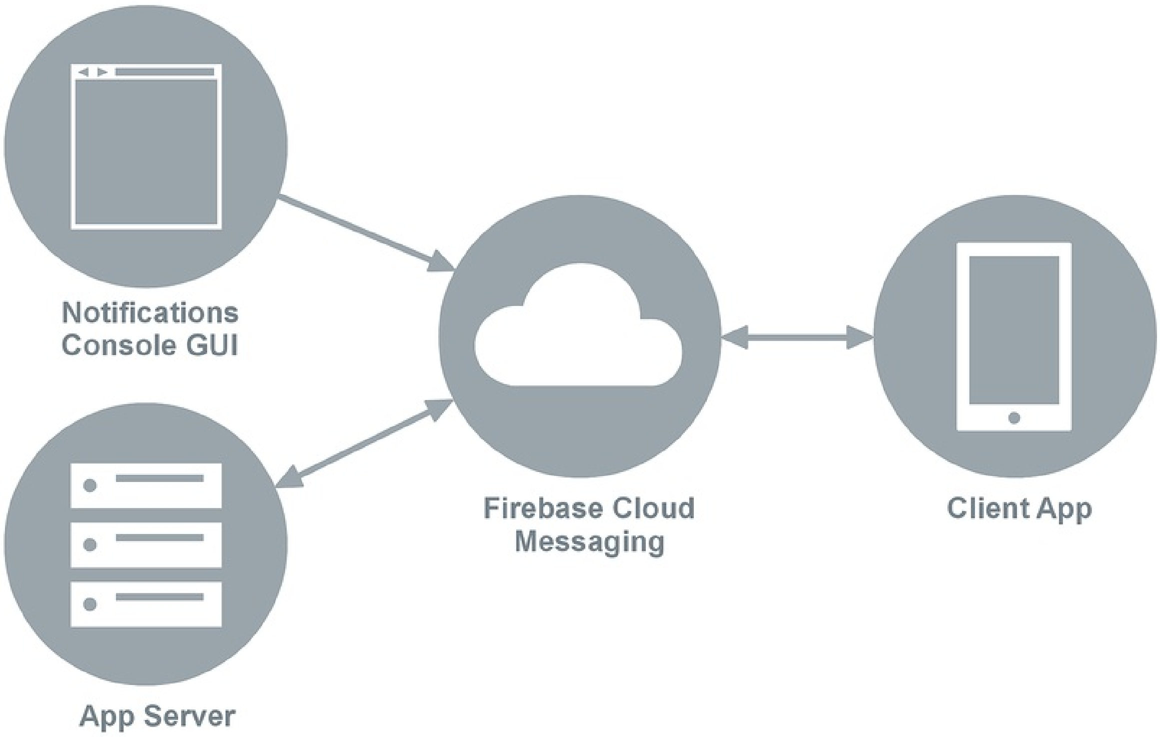


This topic provides a high-level overview of how Firebase Cloud Messaging routes messages between your Xamarin.Android app and an app server, and it provides a step-by-step procedure for acquiring credentials so that your app can use FCM services.

Overview

Firebase Cloud Messaging (FCM) is a cross-platform service that handles the sending, routing, and queueing of messages between server applications and mobile client apps. FCM is the successor to Google Cloud Messaging (GCM), and it is built on Google Play Services.

As illustrated in the following diagram, FCM acts as an intermediary between message senders and clients. A *client app* is an FCM-enabled app that runs on a device. The *app server* (provided by you or your company) is the FCM-enabled server that your client app communicates with through FCM. Unlike GCM, FCM makes it possible for you to send messages to client apps directly via the Firebase Console Notifications GUI:



Using FCM, app servers can send messages to a single device, to a group of devices, or to a number of devices that are subscribed to a topic. A client app can use FCM to subscribe to downstream messages from an app server (for example, to receive remote notifications). For more information about the different types of Firebase messages, see About FCM Messages.

Firebase Cloud Messaging in action

When a downstream message is sent to a client app from an app server, the app server sends the message to an *FCM connection server* provided by Google; the FCM connection server, in turn, forwards the message to a device that is running the client app. Messages can be sent over HTTP or XMPP (Extensible Messaging and Presence Protocol). Because client apps are not always connected or running, the FCM connection server enqueues and stores messages, sending them to client apps as they reconnect and become available. Similarly, FCM will enqueue upstream messages from the client app to the app server if the app server is unavailable. For more about FCM connection servers, see About Firebase Cloud Messaging Server.

FCM uses the following credentials to identify the app server and the client app, and it uses these credentials to authorize message transactions through FCM:

- **Sender ID** – The *Sender ID* is a unique numerical value that is assigned when you create your Firebase project. The sender ID is used to identify each app server that can send messages to the client app. The sender ID is also your project number; you obtain the sender ID from the Firebase Console when you register your project. An example of a Sender ID is `496915549731`.

- **API Key** – The *API key* gives the app server access to Firebase services; FCM uses this key to authenticate the app server. This credential is also referred to as the *Server Key* or the *Web API Key*. An example of an API Key is `AJzbSyCTcpfRT1YRqbz-jIwp1h06YdauvewGDzk`.

- **App ID** – The identity of your client app (independent of any given device) that registers to receive messages from FCM. An example of an App ID is `1:415712510732:android:0e1eb7a661af2460`.

- **Registration Token** – The *Registration Token* (also referred to as the *Instance ID*) is the FCM identity of your client app on a given device. The registration token is generated at run time – your app receives a registration token when it first registers with FCM while running on a device. The registration token authorizes an instance of your client app (running on that particular device) to receive messages from

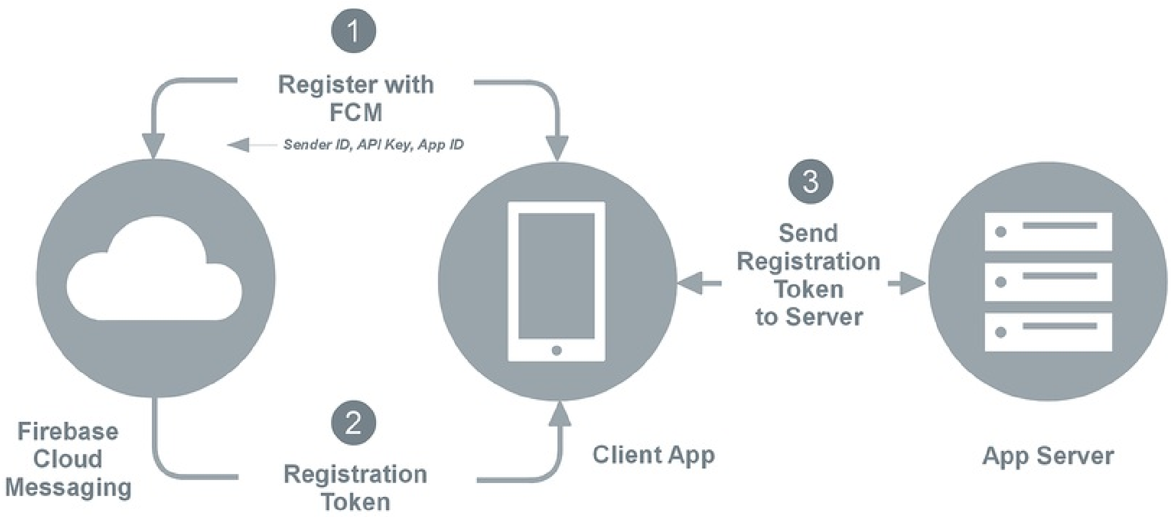
FCM. An example of a registration token is `fkBQTHxKKhs:AP91bHuEedxM4xFAUn0z ... JKZS` (a very long string).

Setting Up Firebase Cloud Messaging (later in this guide) provides detailed instructions for creating a project and generating these credentials. When you create a new project in the Firebase Console, a credentials file called **google-services.json** is created – add this file to your Xamarin.Android project as explained in Remote Notifications with FCM.

The following sections explain how these credentials are used when client apps communicate with app servers through FCM.

Registration with FCM

A client app must first register with FCM before messaging can take place. The client app must complete the registration steps shown in the following diagram:



1. The client app contacts FCM to obtain a registration token, passing the sender ID, API Key, and App ID to FCM.

2. FCM returns a registration token to the client app.

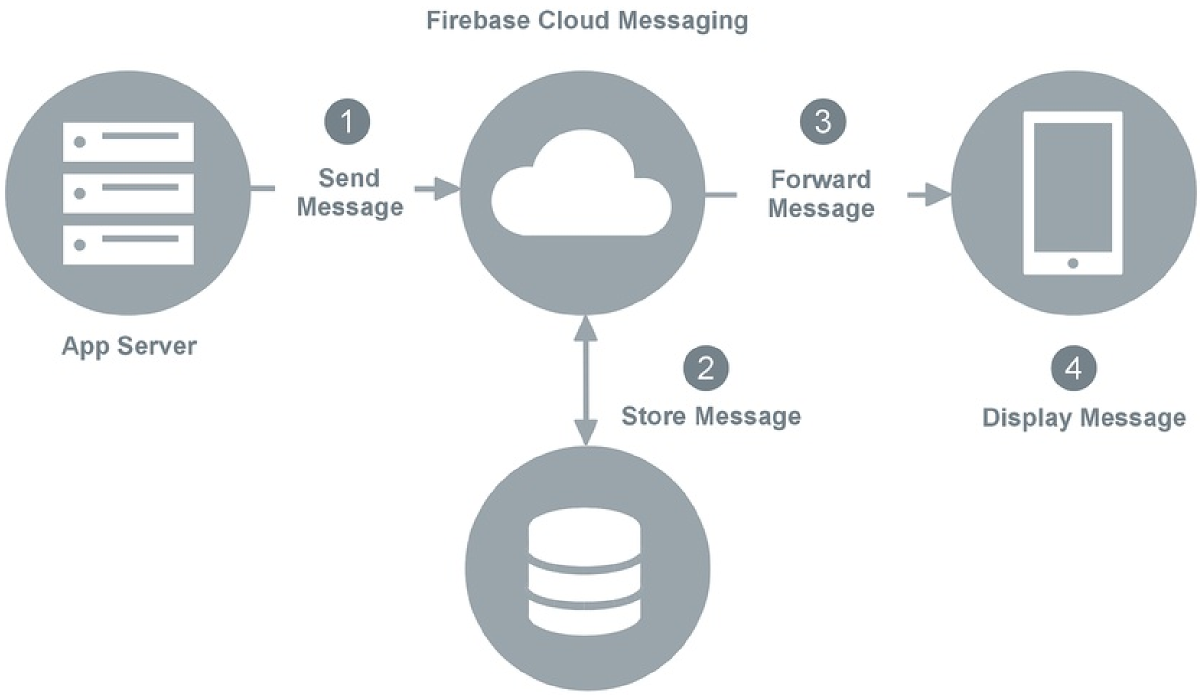
3. The client app (optionally) forwards the registration token to the app server.

The app server caches the registration token for subsequent communications with the client app. The app server can send an acknowledgement back to the client app to indicate that the registration token was received. After this handshake takes place, the client app can receive messages from (or send messages to) the app server. The client app may receive a new registration token if the old token is compromised (see Remote Notifications with FCM for an example of how an app receives registration token updates).

When the client app no longer wants to receive messages from the app server, it can send a request to the app server to delete the registration token. If the client app is uninstalled from a device, FCM detects this and automatically notifies the app server to delete the registration token.

Downstream messaging

The following diagram illustrates how Firebase Cloud Messaging stores and forwards downstream messages:



When the app server sends a downstream message to the client app, it uses the following steps as enumerated in the above diagram:

1. The app server sends the message to FCM.

2. If the client device is not available, the FCM server stores the message in a queue for later transmission. Messages are retained in FCM storage for a maximum of 4 weeks (for more information, see Setting the lifespan of a message).

3. When the client device is available, FCM forwards the message to the client app on that device.

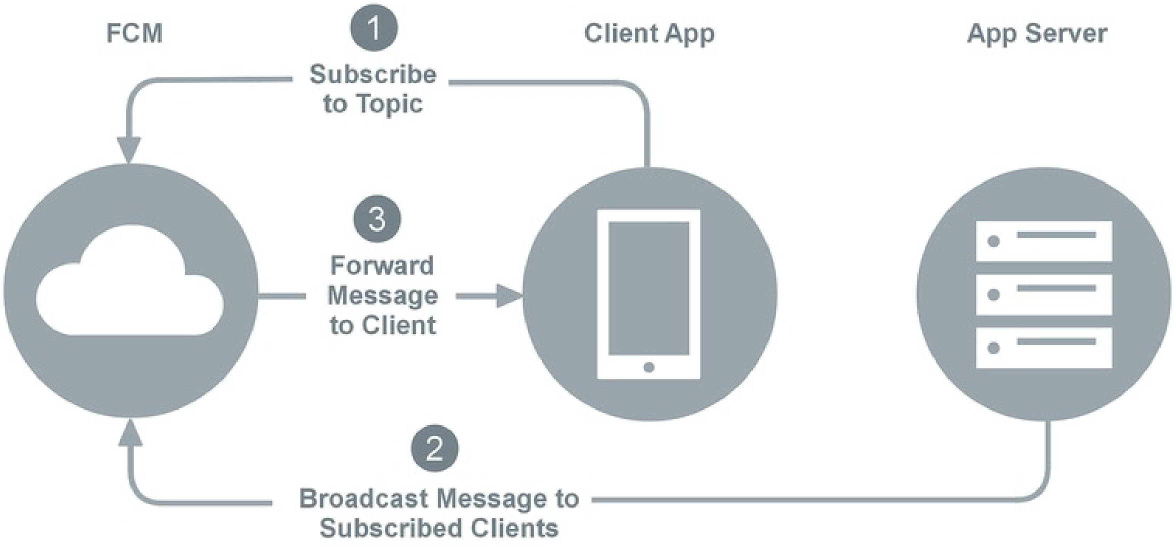
4. The client app receives the message from FCM, processes it, and displays it to the user. For example, if the message is a remote notification, it is presented to the user in the notification area.

In this messaging scenario (where the app server sends a message to a single client app), messages can be up to 4kB in length.

For detailed information about receiving downstream FCM messages on Android, see Remote Notifications with FCM.

Topic messaging

Topic Messaging makes it possible for an app server to send a message to multiple devices that have opted in to a particular topic. You can also compose and send topic messages via the Firebase Console Notifications GUI. FCM handles the routing and delivery of topic messages to subscribed clients. This feature can be used for messages such as weather alerts, stock quotes, and headline news.



The following steps are used in topic messaging (after the client app obtains a registration token as explained earlier):

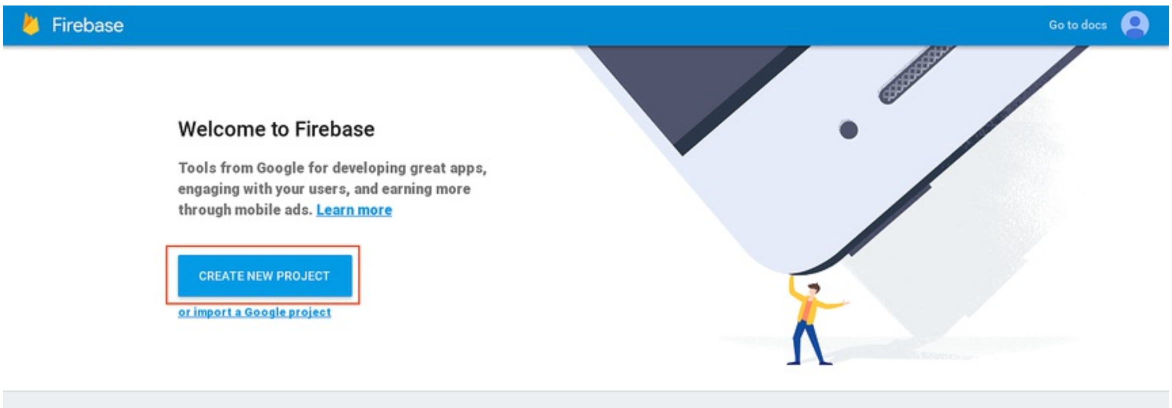
1. The client app subscribes to a topic by sending a subscribe message to FCM.
2. The app server sends topic messages to FCM for distribution.
3. FCM forwards topic messages to clients that have subscribed to that topic.

For more information about Firebase topic messaging, see Google's Topic Messaging on Android.

Setting up Firebase Cloud Messaging

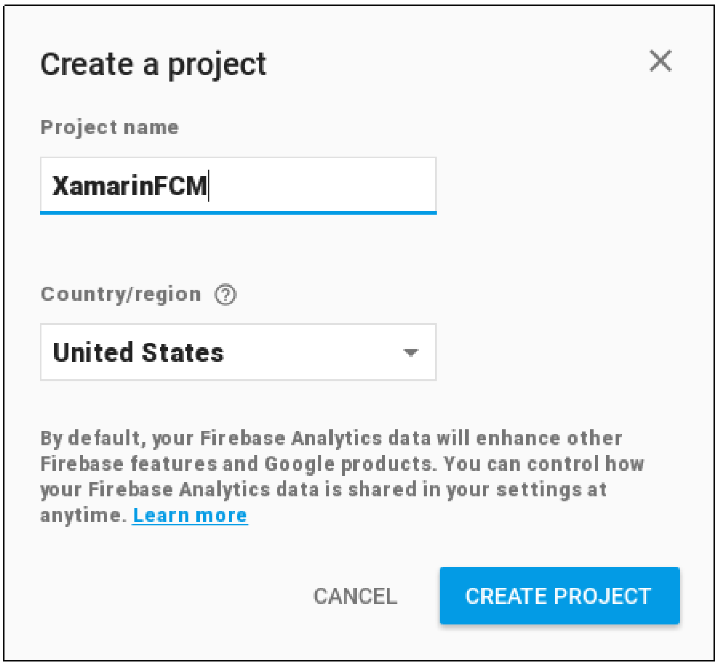
Before you can use FCM services in your app, you must create a new project (or import an existing project) via the Firebase Console. Use the following steps to create a Firebase Cloud Messaging project for your app:

1. Sign into the Firebase Console with your Google account (i.e., your Gmail address) and click **CREATE NEW PROJECT**:

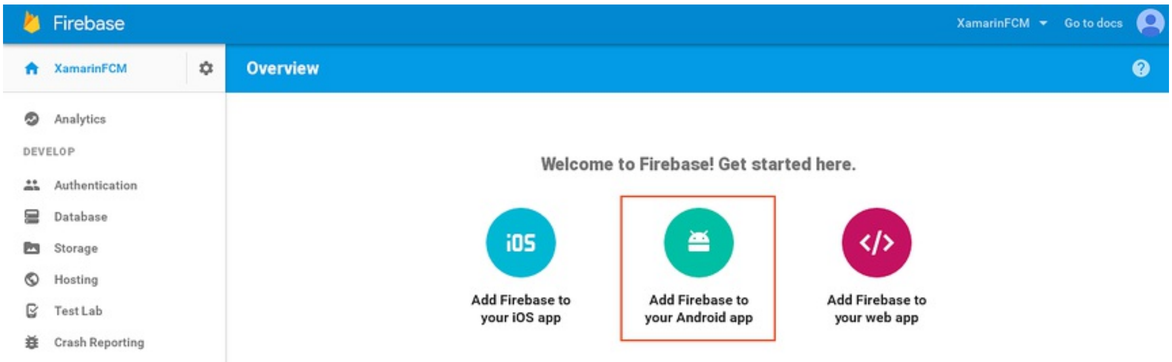
   

   If you have an existing project, click **import a Google project**.

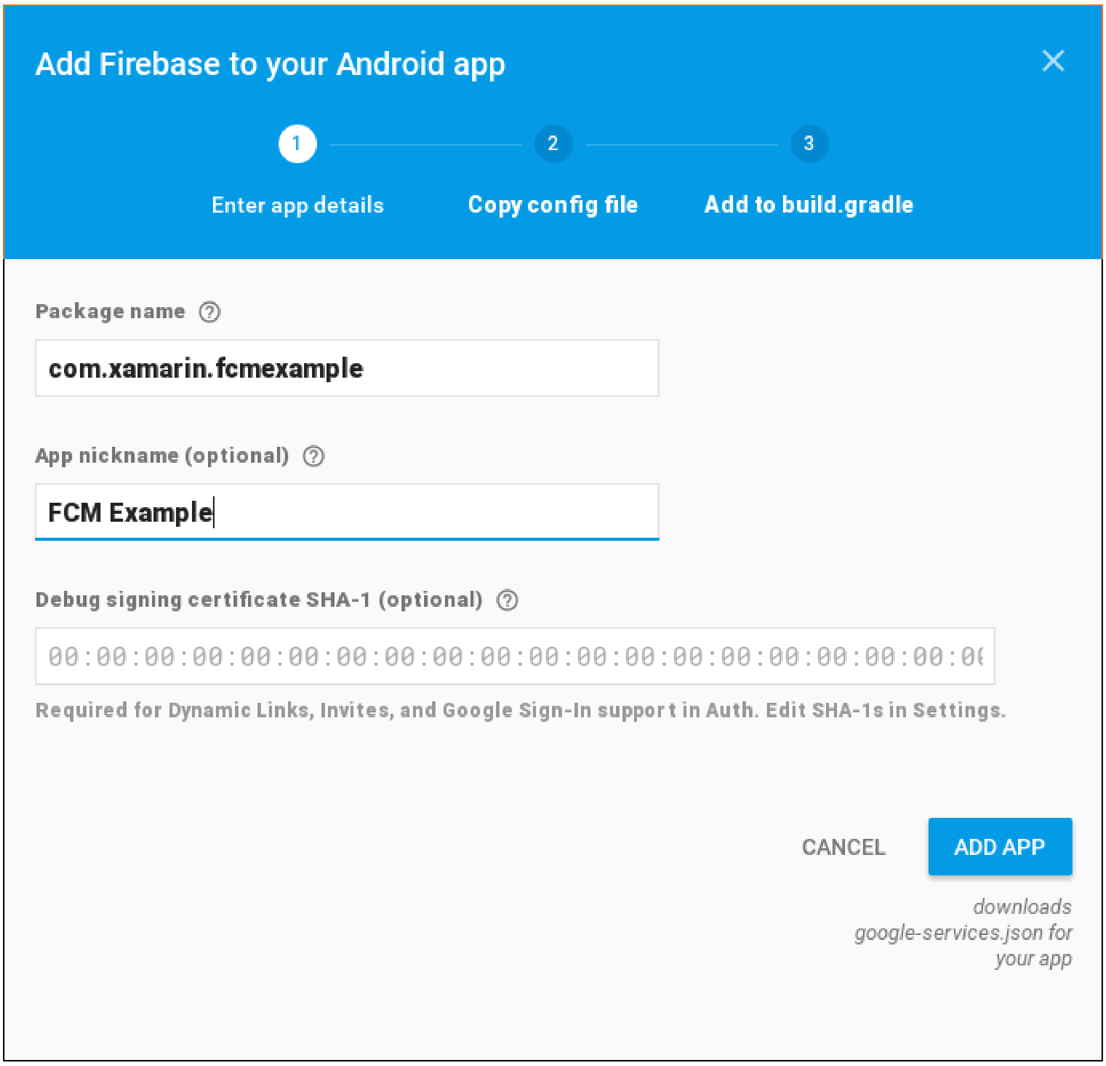
2. In the **Create a project** dialog, enter the name of your project and click **CREATE PROJECT**. In the following example, a new project called **XamarinFCM** is created:



3. In the Firebase Console **Overview**, click **Add Firebase to your Android app**:

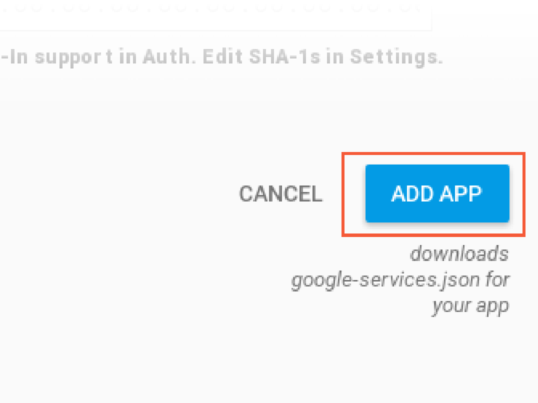


4. In the next screen, enter the package name of your app. In this example, the package name is **com.xamarin.fcmexample**. This value must match the package name of your Android app. An app nickname can also be entered in the **App nickname** field:



5. If your app uses Dynamic links, Invites, or Google Auth, you must also enter your debug signing certificate. For more information about locating your signing certificate, see Finding your Keystore's MD5 or SHA1 Signature. In this example, the signing certificate is left blank.

6. Click **ADD APP**:



A Server API key and a Client ID are automatically generated for the app. This information is packaged in a **google-services.json** file that is automatically downloaded when you click **ADD APP**. Be sure to save this file in a safe place.

For a detailed example of how to add **google-services.json** to an app project to receive FCM push notification messages on Android, see Remote Notifications with FCM.

For further reading

- Google's Firebase Cloud Messaging provides an overview of Firebase Cloud Messaging's key capabilities, an explanation of how it works, and setup instructions.
- Google's Build App Server Send Requests explains how to send messages with your app server.
- RFC 6120 and RFC 6121 explain and define the Extensible Messaging and Presence Protocol (XMPP).
- About FCM Messages describes the different types of messages that can be sent with Firebase Cloud Messaging.

Summary

This article provided an overview of Firebase Cloud Messaging (FCM). It explained the various credentials that are used to identify and authorize messaging between app servers and client apps. It illustrated the registration and downstream messaging scenarios, and it detailed the steps for registering your app with FCM to use FCM services.

Related Links

- Firebase Cloud Messaging

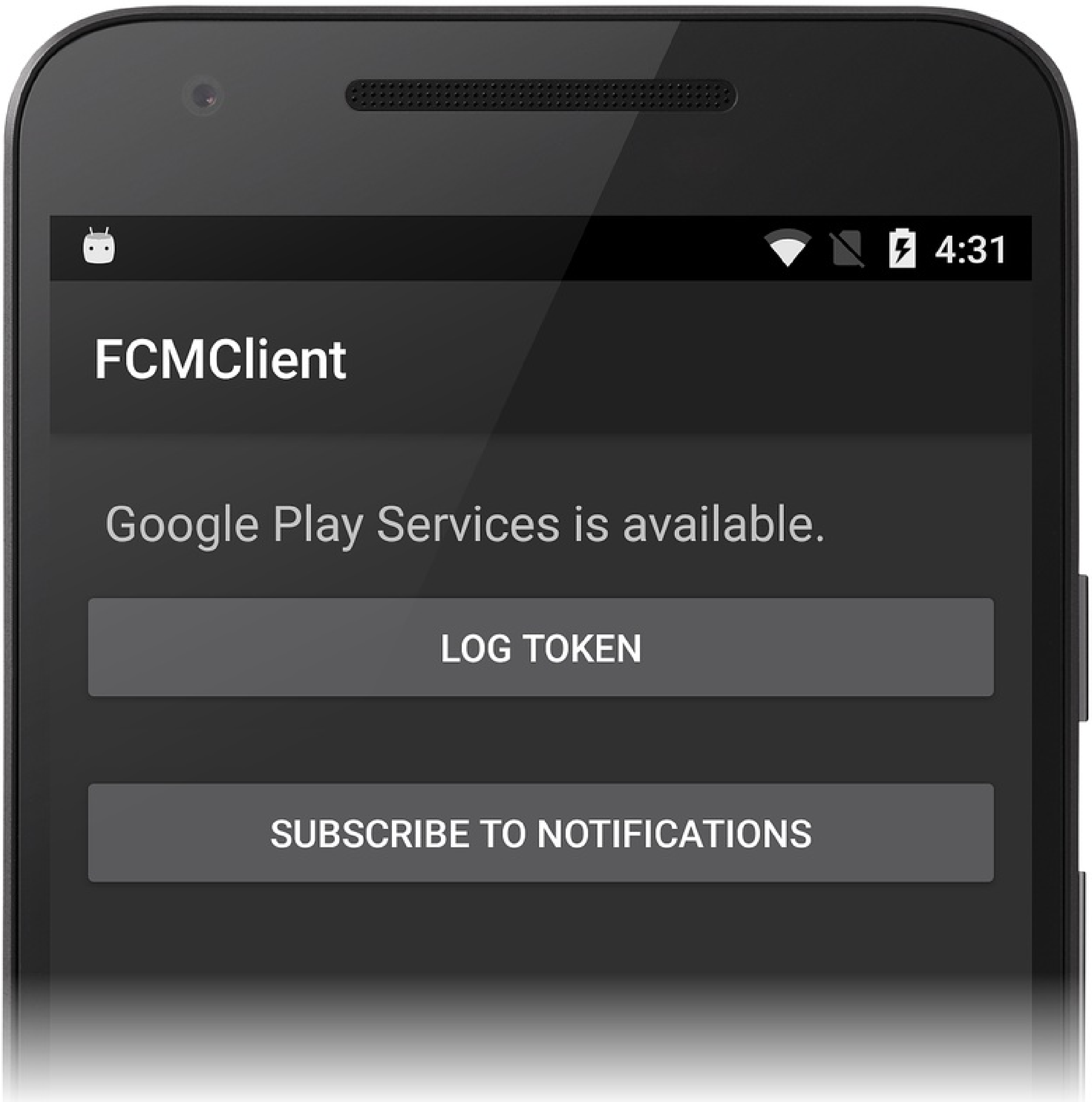
Remote Notifications with Firebase Cloud Messaging

7/8/2021 • 23 minutes to read • Edit Online

This walkthrough provides a step-by-step explanation of how to use Firebase Cloud Messaging to implement remote notifications (also called push notifications) in a Xamarin.Android application. It illustrates how to implement the various classes that are needed for communications with Firebase Cloud Messaging (FCM), provides examples of how to configure the Android Manifest for access to FCM, and demonstrates downstream messaging using the Firebase Console.

FCM notifications overview

In this walkthrough, a basic app called **FCMClient** will be created to illustrate the essentials of FCM messaging. **FCMClient** checks for the presence of Google Play Services, receives registration tokens from FCM, displays remote notifications that you send from the Firebase Console, and subscribes to topic messages:



The following topic areas will be explored:

1. Background Notifications

2. Topic Messages

3. Foreground Notifications

During this walkthrough, you will incrementally add functionality to **FCMClient** and run it on a device or emulator to understand how it interacts with FCM. You will use logging to witness live app transactions with FCM servers, and you will observe how notifications are generated from FCM messages that you enter into the Firebase Console Notifications GUI.

Requirements

It will be helpful to familiarize yourself with the different types of messages that can be sent by Firebase Cloud Messaging. The payload of the message will determine how a client app will receive and process the message.

Before you can proceed with this walkthrough, you must acquire the necessary credentials to use Google's FCM servers; this process is explained in Firebase Cloud Messaging. In particular, you must download the **google-services.json** file to use with the example code presented in this walkthrough. If you have not yet created a project in the Firebase Console (or if you have not yet downloaded the **google-services.json** file), see Firebase Cloud Messaging.

To run the example app, you will need an Android test device or emulator that is compatibile with Firebase. Firebase Cloud Messaging supports clients running on Android 4.0 or higher, and these devices must also have the Google Play Store app installed (Google Play Services 9.2.1 or later is required). If you do not yet have the Google Play Store app installed on your device, visit the Google Play web site to download and install it. Alternately, you can use the Android SDK emulator with Google Play Services installed instead of a test device (you do not have to install the Google Play Store if you are using the Android SDK emulator).

Start an app project

To begin, create a new empty Xamarin.Android project called **FCMClient**. If you are not familiar with creating Xamarin.Android projects, see Hello, Android. After the new app is created, the next step is to set the package name and install several NuGet packages that will be used for communication with FCM.

Set the package name

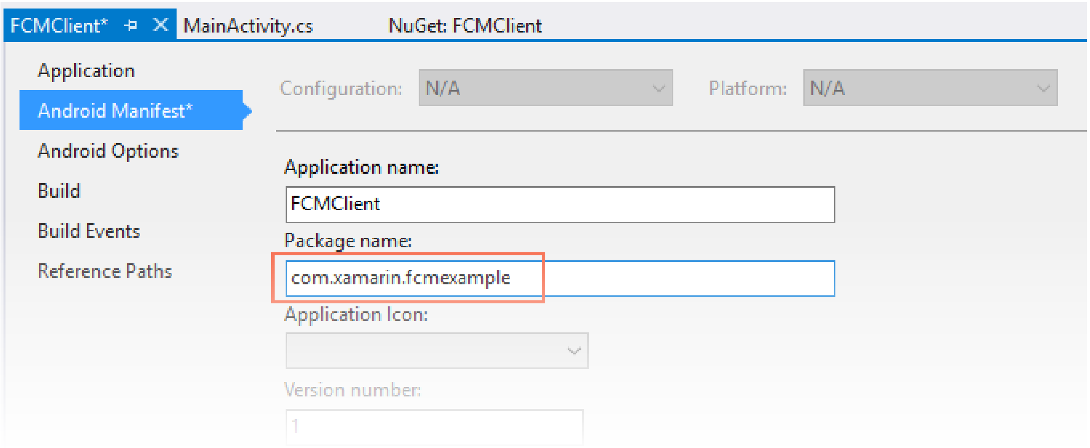
In Firebase Cloud Messaging, you specified a package name for the FCM-enabled app. This package name also serves as the *application ID* that is associated with the API key. Configure the app to use this package name:

- Visual Studio
- Visual Studio for Mac

1. Open the properties for the **FCMClient** project.

2. In the **Android Manifest** page, set the package name.

In the following example, the package name is set to `com.xamarin.fcmexample`:



While you are updating the **Android Manifest**, also check to be sure that the `Internet` permission is enabled.

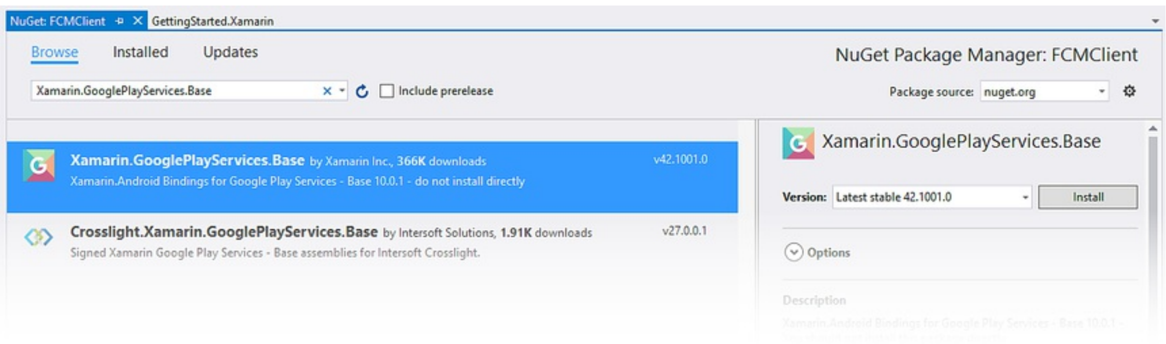
> **IMPORTANT**
>
> The client app will be unable to receive a registration token from FCM if this package name does not *exactly* match the package name that was entered into the Firebase Console.

Add the Xamarin Google Play Services Base package

Because Firebase Cloud Messaging depends on Google Play Services, the Xamarin Google Play Services - Base NuGet package must be added to the Xamarin.Android project. You will need version 29.0.0.2 or later.

- Visual Studio
- Visual Studio for Mac

1. In Visual Studio, right-click **References > Manage NuGet Packages ...**.
2. Click the **Browse** tab and search for **Xamarin.GooglePlayServices.Base**.
3. Install this package into the **FCMClient** project:



If you get an error during installation of the NuGet, close the **FCMClient** project, open it again, and retry the NuGet installation.

When you install **Xamarin.GooglePlayServices.Base**, all of the necessary dependencies are also installed. Edit **MainActivity.cs** and add the following `using` statement:

```
using Android.Gms.Common;
```

This statement makes the `GoogleApiAvailability` class in **Xamarin.GooglePlayServices.Base** available to **FCMClient** code. `GoogleApiAvailability` is used to check for the presence of Google Play Services.

Add the Xamarin Firebase Messaging package

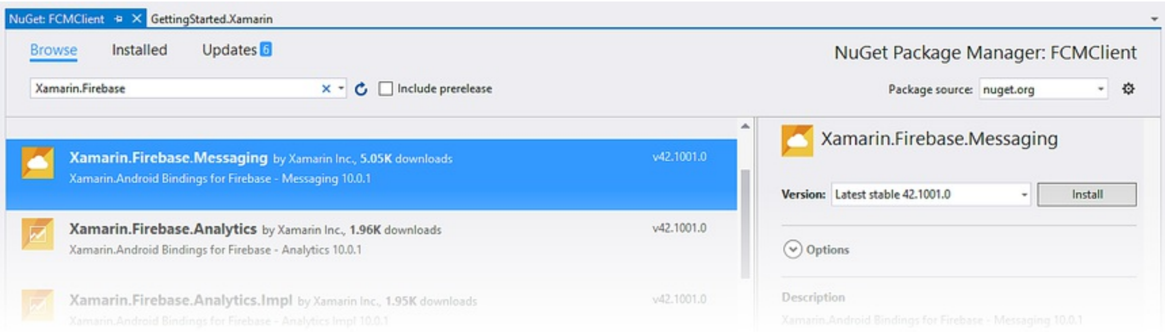
To receive messages from FCM, the Xamarin Firebase - Messaging NuGet package must be added to the app project. Without this package, an Android application cannot receive messages from FCM servers.

- Visual Studio
- Visual Studio for Mac

1. In Visual Studio, right-click **References > Manage NuGet Packages ...**.

2. Search for **Xamarin.Firebase.Messaging**.

3. Install this package into the **FCMClient** project:



When you install **Xamarin.Firebase.Messaging**, all of the necessary dependencies are also installed.

Next, edit **MainActivity.cs** and add the following `using` statements:

```
using Firebase.Messaging;
using Firebase.Iid;
using Android.Util;
```

The first two statements make types in the **Xamarin.Firebase.Messaging** NuGet package available to **FCMClient** code. **Android.Util** adds logging functionality that will be used to observe transactions with FMS.

Add the Google Services JSON file

The next step is to add the **google-services.json** file to the root directory of your project:

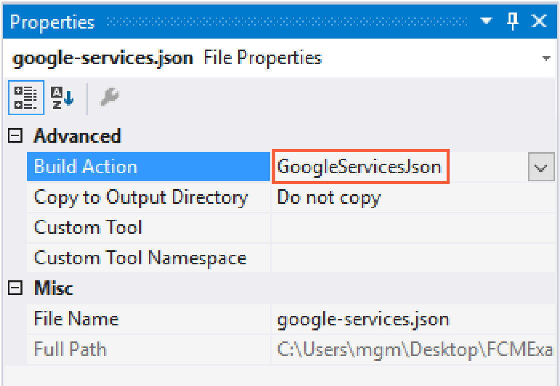
- Visual Studio
- Visual Studio for Mac

1. Copy **google-services.json** to the project folder.

2. Add **google-services.json** to the app project (click **Show All Files** in the **Solution Explorer**, right click **google-services.json**, then select **Include in Project**).

3. Select **google-services.json** in the **Solution Explorer** window.

4. In the **Properties** pane, set the **Build Action** to **GoogleServicesJson**:



NOTE

If the **GoogleServicesJson** build action is not shown, save and close the solution, then reopen it.

When **google-services.json** is added to the project (and the **GoogleServicesJson** build action is set), the build process extracts the client ID and API key and then adds these credentials to the merged/generated **AndroidManifest.xml** that resides at **obj/Debug/android/AndroidManifest.xml**. This merge process automatically adds any permissions and other FCM elements that are needed for connection to FCM servers.

Check for Google Play Services and create a notification channel

Google recommends that Android apps check for the presence of the Google Play Services APK before accessing Google Play Services features (for more information, see Check for Google Play services).

An initial layout for the app's UI will be created first. Edit **Resources/layout/Main.axml** and replace its contents with the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="10dp">
    <TextView
        android:text=" "
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/msgText"
        android:textAppearance="?android:attr/textAppearanceMedium"
        android:padding="10dp" />
</LinearLayout>
```

This `TextView` will be used to display messages that indicate whether Google Play Services is installed. Save the changes to **Main.axml**.

Edit **MainActivity.cs** and add the following instance variables to the `MainActivity` class:

```
public class MainActivity : AppCompatActivity
{
    static readonly string TAG = "MainActivity";

    internal static readonly string CHANNEL_ID = "my_notification_channel";
    internal static readonly int NOTIFICATION_ID = 100;

    TextView msgText;
```

The variables `CHANNEL_ID` and `NOTIFICATION_ID` will be used in the method `CreateNotificationChannel` that will be added to `MainActivity` later on in this walkthrough.

In the following example, the `OnCreate` method will verify that Google Play Services is available before the app attempts to use FCM services. Add the following method to the `MainActivity` class:

```
public bool IsPlayServicesAvailable ()
{
    int resultCode = GoogleApiAvailability.Instance.IsGooglePlayServicesAvailable (this);
    if (resultCode != ConnectionResult.Success)
    {
        if (GoogleApiAvailability.Instance.IsUserResolvableError (resultCode))
            msgText.Text = GoogleApiAvailability.Instance.GetErrorString (resultCode);
        else
        {
            msgText.Text = "This device is not supported";
            Finish ();
        }
        return false;
    }
    else
    {
        msgText.Text = "Google Play Services is available.";
        return true;
    }
}
```

This code checks the device to see if the Google Play Services APK is installed. If it is not installed, a message is displayed in the `TextBox` that instructs the user to download an APK from the Google Play Store (or to enable it in the device's system settings).

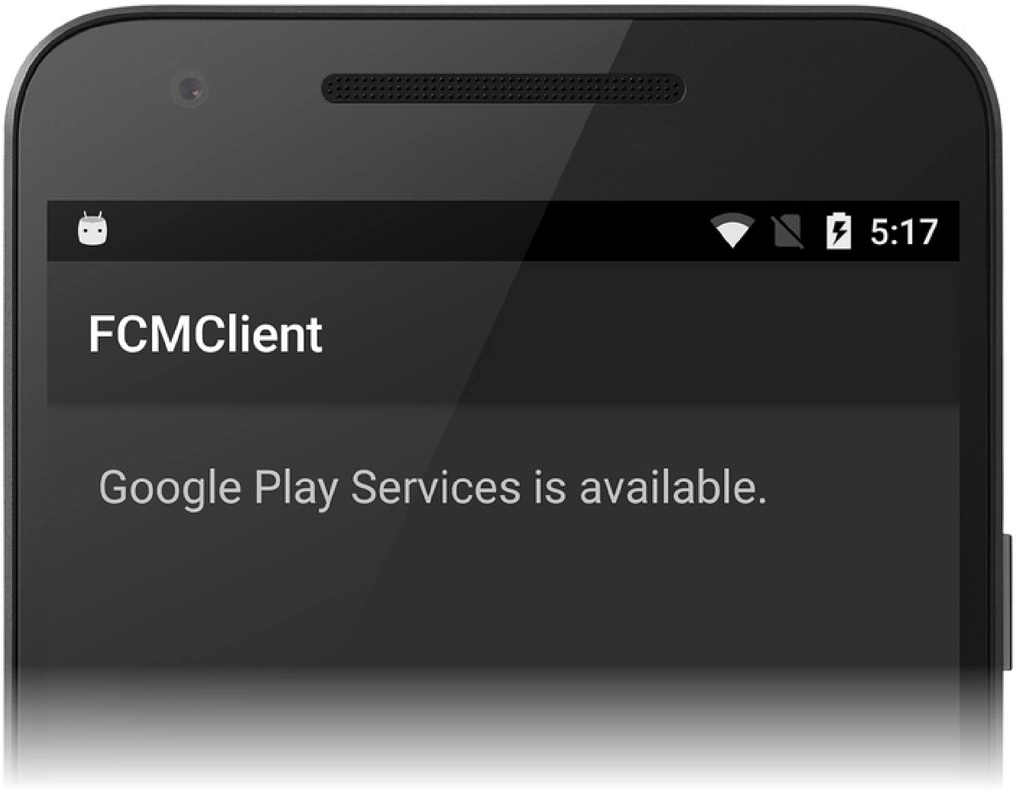
Apps that are running on Android 8.0 (API level 26) or higher must create a *notification channel* for publishing their notifications. Add the following method to the `MainActivity` class which will create the notification channel (if necessary):

```
void CreateNotificationChannel()
{
    if (Build.VERSION.SdkInt < BuildVersionCodes.O)
    {
        // Notification channels are new in API 26 (and not a part of the
        // support library). There is no need to create a notification
        // channel on older versions of Android.
        return;
    }

    var channel = new NotificationChannel(CHANNEL_ID,
                                          "FCM Notifications",
                                          NotificationImportance.Default)
                  {

                      Description = "Firebase Cloud Messages appear in this channel"
                  };

    var notificationManager =
(NotificationManager)GetSystemService(Android.Content.Context.NotificationService);
    notificationManager.CreateNotificationChannel(channel);
}
```

Replace the `OnCreate` method with the following code:

```
protected override void OnCreate (Bundle bundle)
{
    base.OnCreate (bundle);
    SetContentView (Resource.Layout.Main);
    msgText = FindViewById<TextView> (Resource.Id.msgText);

    IsPlayServicesAvailable ();

    CreateNotificationChannel();
}
```

`IsPlayServicesAvailable` is called at the end of `OnCreate` so that the Google Play Services check runs each time the app starts. The method `CreateNotificationChannel` is called to ensure that a notification channel exists for devices running Android 8 or higher. If your app has an `OnResume` method, it should call `IsPlayServicesAvailable` from `OnResume` as well. Completely rebuild and run the app. If all is configured properly, you should see a screen that looks like the following screenshot:



If you don't get this result, verify that the Google Play Services APK is installed on your device (for more information, see Setting Up Google Play Services). Also verify that you have added the **Xamarin.Google.Play.Services.Base** package to your **FCMClient** project as explained earlier.

Add the instance ID receiver

The next step is to add a service that extends `FirebaseInstanceIdService` to handle the creation, rotation, and updating of Firebase registration tokens. The `FirebaseInstanceIdService` service is required for FCM to be able to send messages to the device. When the `FirebaseInstanceIdService` service is added to the client app, the app will automatically receive FCM messages and display them as notifications whenever the app is backgrounded.

Declare the receiver in the Android Manifest

Edit **AndroidManifest.xml** and insert the following `<receiver>` elements into the `<application>` section:

```xml
<receiver
    android:name="com.google.firebase.iid.FirebaseInstanceIdInternalReceiver"
    android:exported="false" />
<receiver
    android:name="com.google.firebase.iid.FirebaseInstanceIdReceiver"
    android:exported="true"
    android:permission="com.google.android.c2dm.permission.SEND">
    <intent-filter>
        <action android:name="com.google.android.c2dm.intent.RECEIVE" />
        <action android:name="com.google.android.c2dm.intent.REGISTRATION" />
        <category android:name="${applicationId}" />
    </intent-filter>
</receiver>
```

This XML does the following:

- Declares a `FirebaseInstanceIdReceiver` implementation that provides a unique identifier for each app instance. This receiver also authenticates and authorizes actions.
- Declares an internal `FirebaseInstanceIdInternalReceiver` implementation that is used to start services securely.
- The app ID is stored in the **google-services.json** file that was added to the project. The Xamarin.Android Firebase bindings will replace the token `${applicationId}` with the app ID; no additional code is required by the client app to provide the app ID.

The `FirebaseInstanceIdReceiver` is a `WakefulBroadcastReceiver` that receives `FirebaseInstanceId` and `FirebaseMessaging` events and delivers them to the class that you derive from `FirebaseInstanceIdService`.

Implement the Firebase Instance ID Service

The work of registering the application with FCM is handled by the custom `FirebaseInstanceIdService` service that you provide. `FirebaseInstanceIdService` performs the following steps:

1. Uses the Instance ID API to generate security tokens that authorize the client app to access FCM and the app server. In return, the app gets back a registration token from FCM.
2. Forwards the registration token to the app server if the app server requires it.

Add a new file called **MyFirebaseIIDService.cs** and replace its template code with the following:

```
using System;
using Android.App;
using Firebase.Iid;
using Android.Util;

namespace FCMClient
{
    [Service]
    [IntentFilter(new[] { "com.google.firebase.INSTANCE_ID_EVENT" })]
    public class MyFirebaseIIDService : FirebaseInstanceIdService
    {
        const string TAG = "MyFirebaseIIDService";
        public override void OnTokenRefresh()
        {
            var refreshedToken = FirebaseInstanceId.Instance.Token;
            Log.Debug(TAG, "Refreshed token: " + refreshedToken);
            SendRegistrationToServer(refreshedToken);
        }
        void SendRegistrationToServer(string token)
        {
            // Add custom implementation, as needed.
        }
    }
}
```

This service implements an `OnTokenRefresh` method that is invoked when the registration token is initially created or changed. When `OnTokenRefresh` runs, it retrieves the latest token from the `FirebaseInstanceId.Instance.Token` property (which is updated asynchronously by FCM). In this example, the refreshed token is logged so that it can be viewed in the output window:

```
var refreshedToken = FirebaseInstanceId.Instance.Token;
Log.Debug(TAG, "Refreshed token: " + refreshedToken);
```

`OnTokenRefresh` is invoked infrequently: it is used to update the token under the following circumstances:

- When the app is installed or uninstalled.
- When the user deletes app data.
- When the app erases the Instance ID.
- When the security of the token has been compromised.

According to Google's Instance ID documentation, the FCM Instance ID service will request that the app refresh its token periodically (typically, every 6 months).

`OnTokenRefresh` also calls `SendRegistrationToAppServer` to associate the user's registration token with the server-side account (if any) that is maintained by the application:

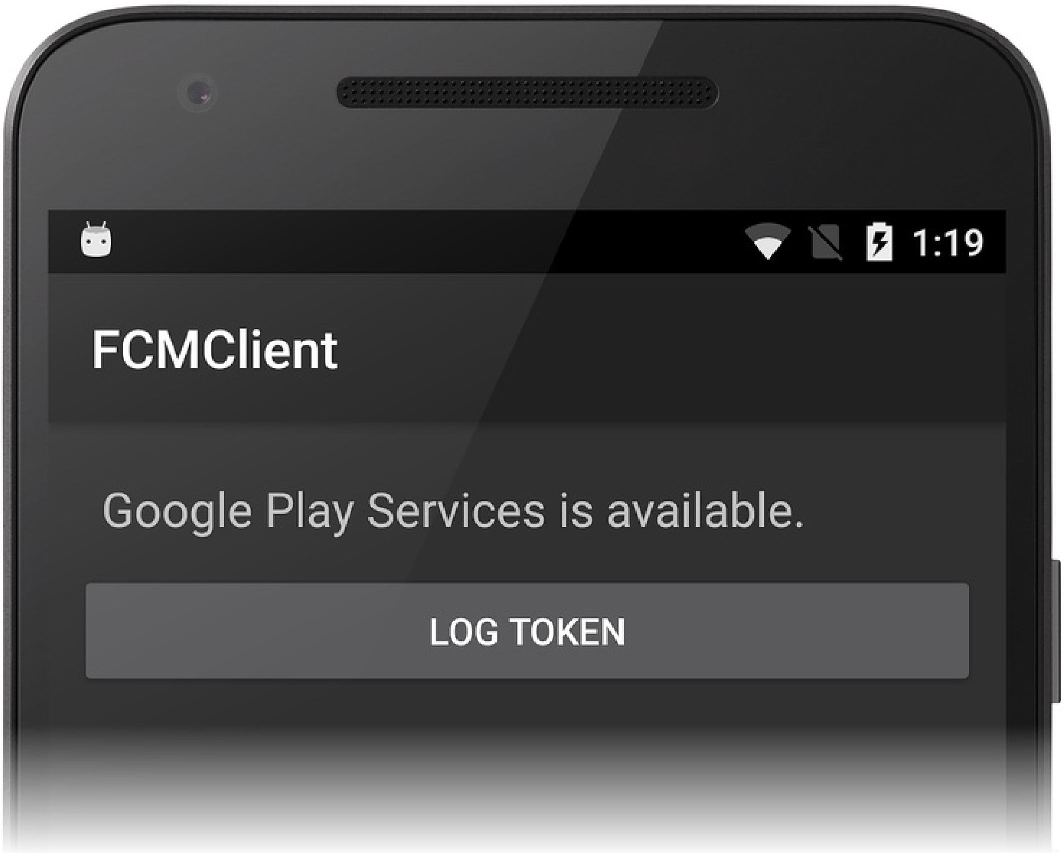
```
void SendRegistrationToAppServer (string token)
{
    // Add custom implementation here as needed.
}
```

Because this implementation depends on the design of the app server, an empty method body is provided in this example. If your app server requires FCM registration information, modify `SendRegistrationToAppServer` to associate the user's FCM instance ID token with any server-side account maintained by your app. (Note that the token is opaque to the client app.)

When a token is sent to the app server, `SendRegistrationToAppServer` should maintain a boolean to indicate whether the token has been sent to the server. If this boolean is false, `SendRegistrationToAppServer` sends the token to the app server – otherwise, the token was already sent to the app server in a previous call. In some cases (such as this `FCMClient` example), the app server does not need the token; therefore, this method is not required for this example.

Implement client app code

Now that the receiver services are in place, client app code can be written to take advantage of these services. In the following sections, a button is added to the UI to log the registration token (also called the *Instance ID token*), and more code is added to `MainActivity` to view `Intent` information when the app is launched from a notification:



Log tokens

The code added in this step is intended only for demonstration purposes – a production client app would have no need to log registration tokens. Edit **Resources/layout/Main.axml** and add the following `Button` declaration immediately after the `TextView` element:

```
<Button
  android:id="@+id/logTokenButton"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_gravity="center_horizontal"
  android:text="Log Token" />
```

Add the following code to the end of the `MainActivity.OnCreate` method:

```
var logTokenButton = FindViewById<Button>(Resource.Id.logTokenButton);
logTokenButton.Click += delegate {
    Log.Debug(TAG, "InstanceID token: " + FirebaseInstanceId.Instance.Token);
};
```

This code logs the current token to the output window when the **Log Token** button is tapped.

Handle notification intents

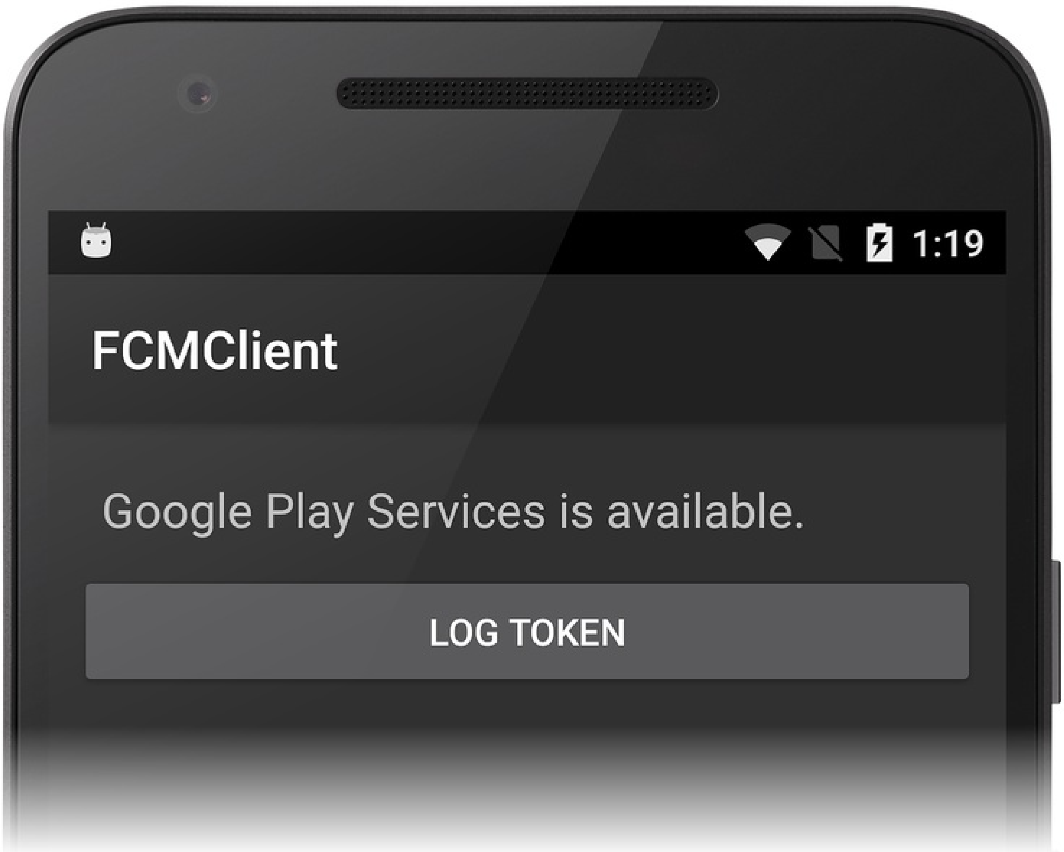
When the user taps a notification issued from **FCMClient**, any data accompanying that notification message is made available in `Intent` extras. Edit **MainActivity.cs** and add the following code to the top of the `OnCreate` method (before the call to `IsPlayServicesAvailable`):

```
if (Intent.Extras != null)
{
    foreach (var key in Intent.Extras.KeySet())
    {
        var value = Intent.Extras.GetString(key);
        Log.Debug(TAG, "Key: {0} Value: {1}", key, value);
    }
}
```

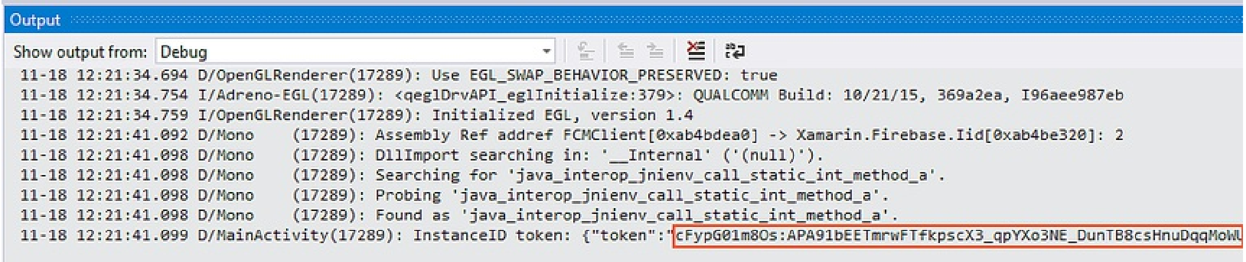
The app's launcher `Intent` is fired when the user taps its notification message, so this code will log any accompanying data in the `Intent` to the output window. If a different `Intent` must be fired, the `click_action` field of the notification message must be set to that `Intent` (the launcher `Intent` is used when no `click_action` is specified).

Background notifications

Build and run the **FCMClient** app. The **Log Token** button is displayed:



Tap the **Log Token** button. A message like the following should be displayed in the IDE output window:



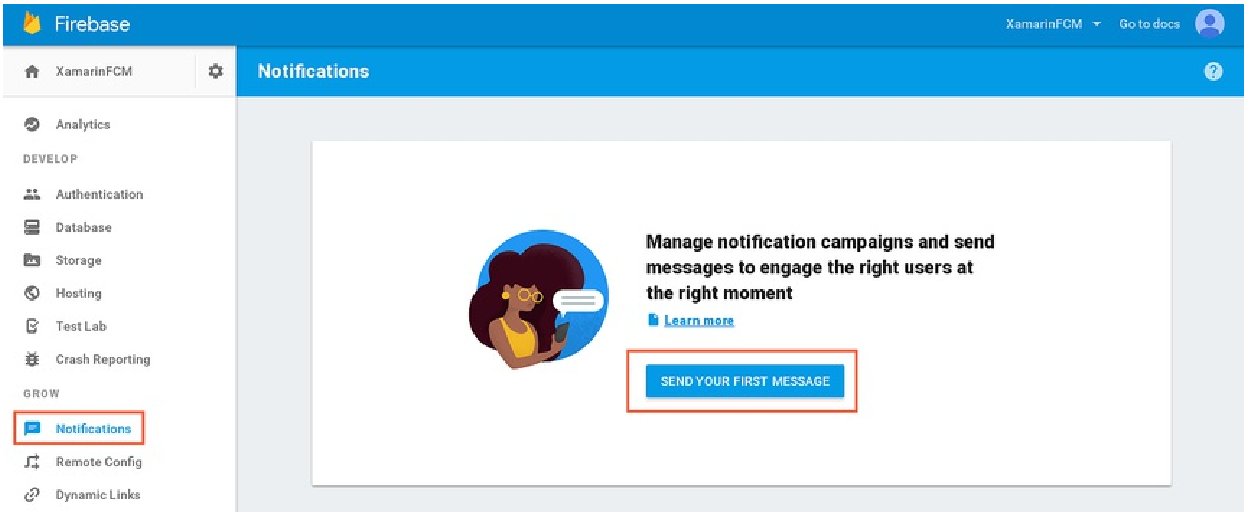
The long string labeled with **token** is the instance ID token that you will paste into the Firebase Console – select and copy this string to the clipboard. If you do not see an instance ID token, add the following line to the top of the `OnCreate` method to verify that **google-services.json** was parsed correctly:

```
Log.Debug(TAG, "google app id: " + GetString(Resource.String.google_app_id));
```

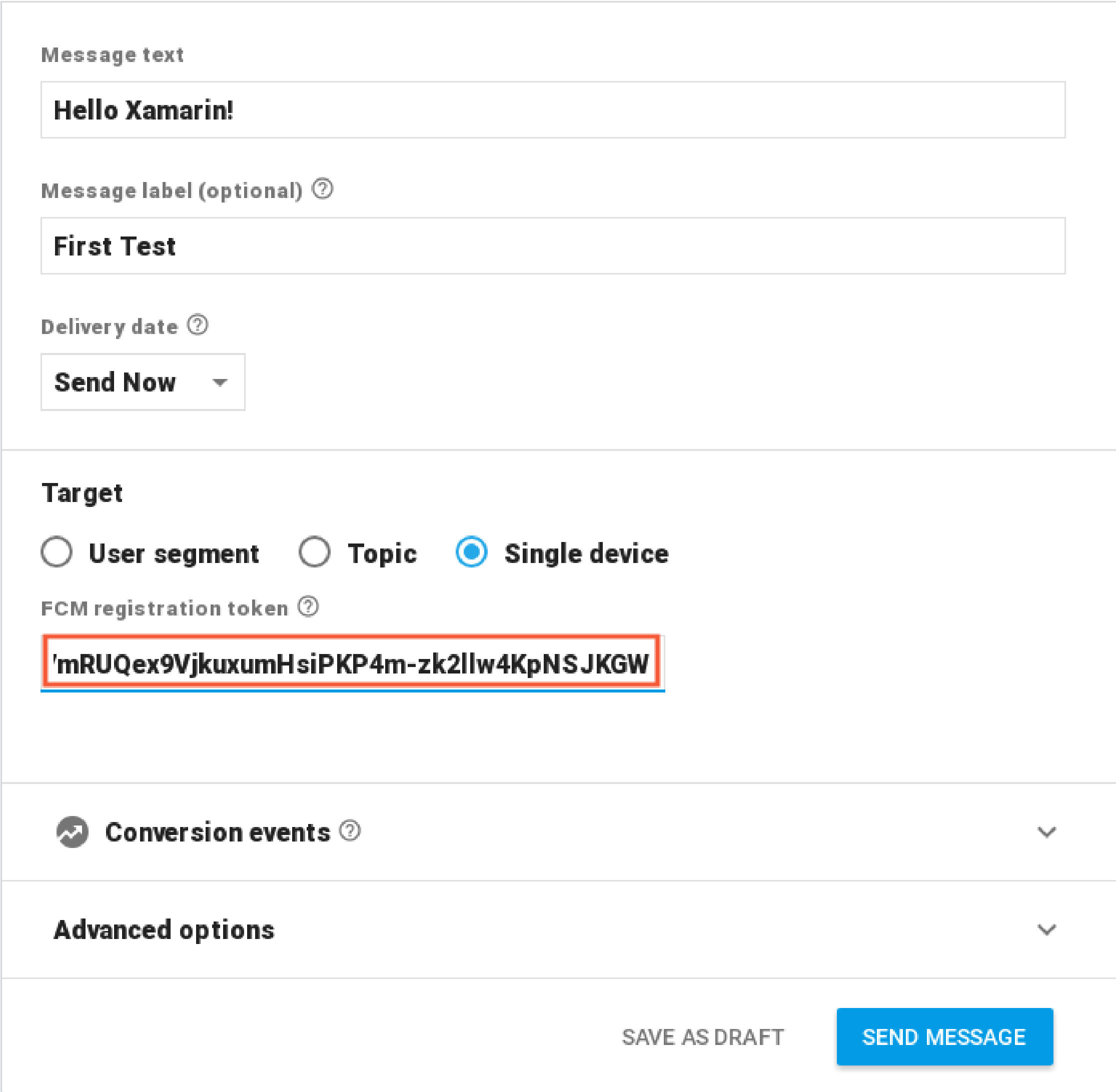
The `google_app_id` value logged to the output window should match the `mobilesdk_app_id` value recorded in **google-services.json**. The `Resource.String.google_app_id` is generated by msbuild when processing **google-services.json**.

Send a message

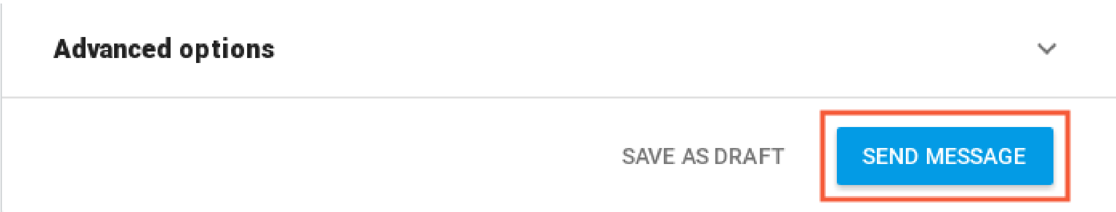
Sign into the Firebase Console, select your project, click **Notifications**, and click **SEND YOUR FIRST MESSAGE**:



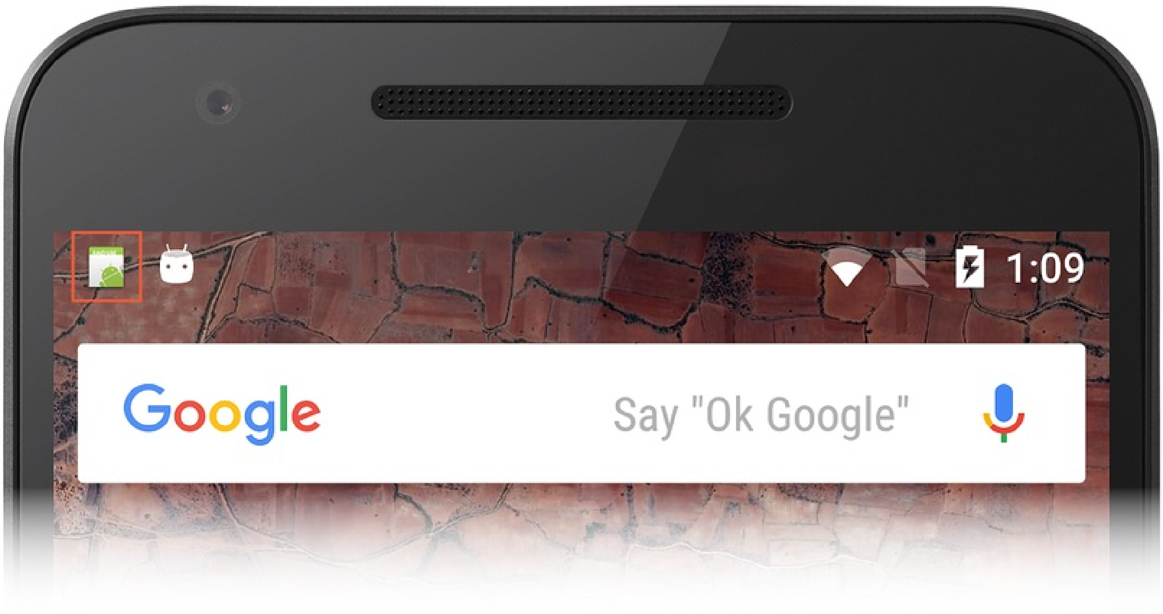
On the **Compose message** page, enter the message text and select **Single device**. Copy the instance ID token from the IDE output window and paste it into the **FCM registration token** field of the Firebase Console:



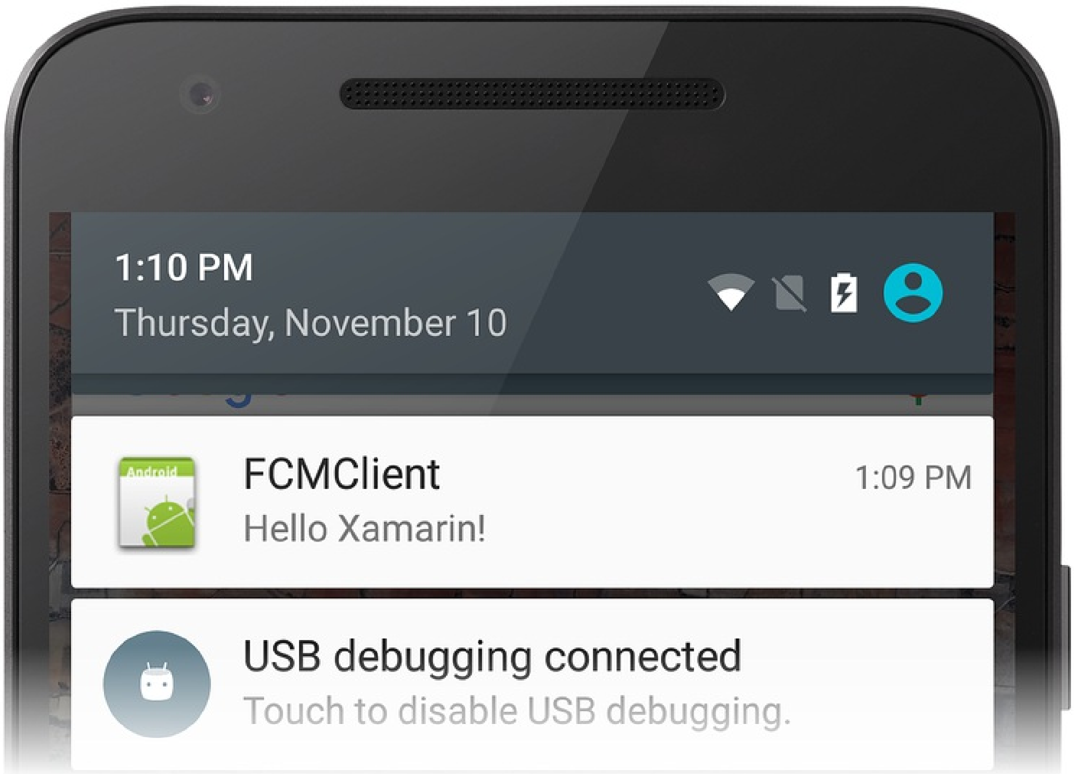
On the Android device (or emulator), background the app by tapping the Android **Overview** button and touching the home screen. When the device is ready, click **SEND MESSAGE** in the Firebase Console:



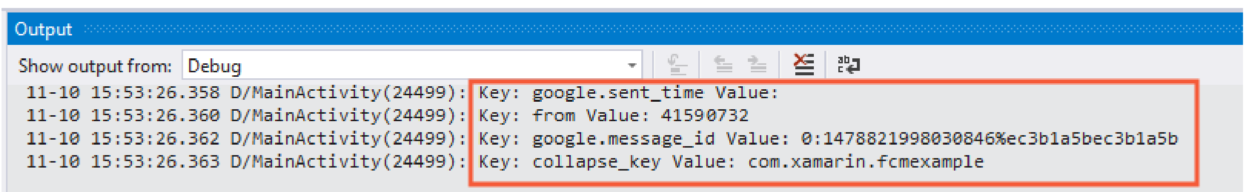
When the **Review message** dialog is displayed, click **SEND**. The notification icon should appear in the notification area of the device (or emulator):



Open the notification icon to view the message. The notification message should be exactly what was typed into the **Message text** field of the Firebase Console:



Tap the notification icon to launch the **FCMClient** app. The `Intent` extras sent to **FCMClient** are listed in the IDE output window:



In this example, the **from** key is set to the Firebase project number of the app (in this example, `41590732`), and the **collapse_key** is set to its package name (**com.xamarin.fcmexample**). If you do not receive a message, try deleting the **FCMClient** app on the device (or emulator) and repeat the above steps.

> **NOTE**
>
> If you force-close the app, FCM will stop delivering notifications. Android prevents background service broadcasts from inadvertently or unnecessarily launching components of stopped applications. (For more information about this behavior, see Launch controls on stopped applications.) For this reason, it is necessary to manually uninstall the app each time you run it and stop it from a debug session – this forces FCM to generate a new token so that messages will continue to be received.

Add a custom default notification icon

In the previous example, the notification icon is set to the application icon. The following XML configures a custom default icon for notifications. Android displays this custom default icon for all notification messages where the notification icon is not explicitly set.

To add a custom default notification icon, add your icon to the **Resources/drawable** directory, edit **AndroidManifest.xml**, and insert the following `<meta-data>` element into the `<application>` section:

```
<meta-data
    android:name="com.google.firebase.messaging.default_notification_icon"
    android:resource="@drawable/ic_stat_ic_notification" />
```

In this example, the notification icon that resides at **Resources/drawable/ic_stat_ic_notification.png** will be used as the custom default notification icon. If a custom default icon is not configured in **AndroidManifest.xml** and no icon is set in the notification payload, Android uses the application icon as the notification icon (as seen in the notification icon screenshot above).

Handle topic messages

The code written thus far handles registration tokens and adds remote notification functionality to the app. The next example adds code that listens for *topic messages* and forwards them to the user as remote notifications. Topic messages are FCM messages that are sent to one or more devices that subscribe to a particular topic. For more information about topic messages, see Topic Messaging.

Subscribe to a topic

Edit **Resources/layout/Main.axml** and add the following `Button` declaration immediately after the previous `Button` element:

```
<Button
  android:id="@+id/subscribeButton"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_gravity="center_horizontal"
  android:layout_marginTop="20dp"
  android:text="Subscribe to Notifications" />
```

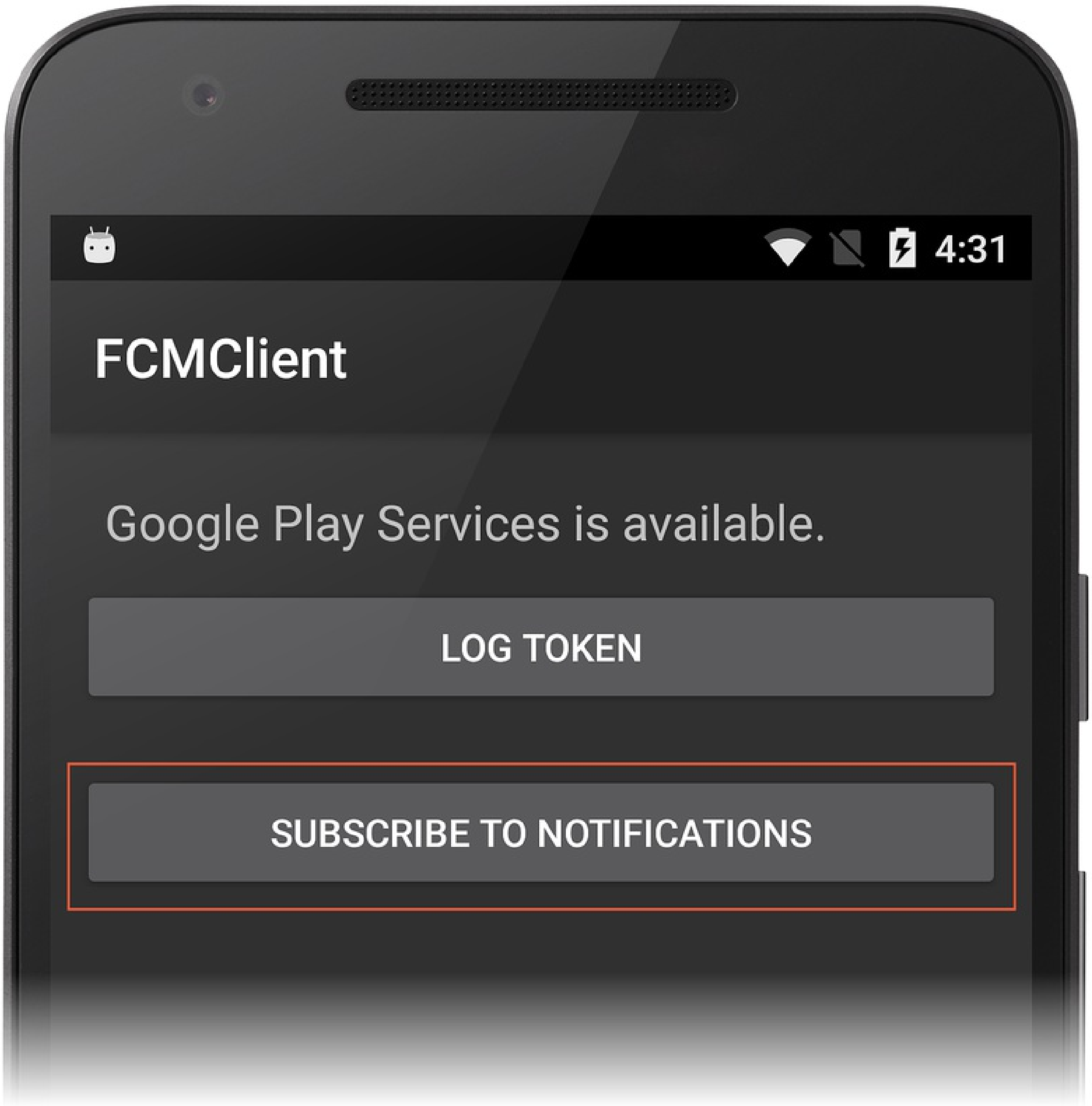
This XML adds a **Subscribe to Notification** button to the layout. Edit **MainActivity.cs** and add the following code to the end of the `OnCreate` method:

```
var subscribeButton = FindViewById<Button>(Resource.Id.subscribeButton);
subscribeButton.Click += delegate {
    FirebaseMessaging.Instance.SubscribeToTopic("news");
    Log.Debug(TAG, "Subscribed to remote notifications");
};
```

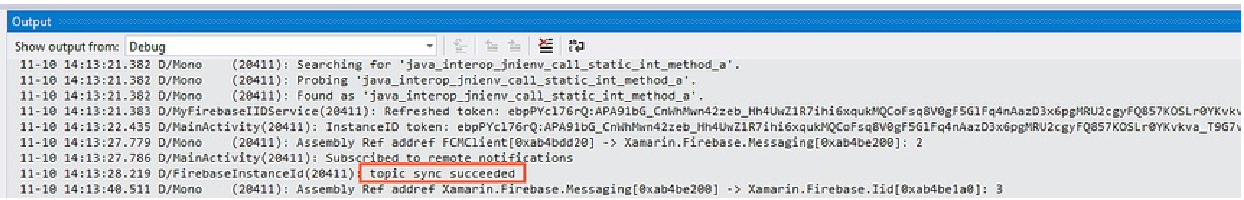
This code locates the **Subscribe to Notification** button in the layout and assigns its click handler to code that calls `FirebaseMessaging.Instance.SubscribeToTopic`, passing in the subscribed topic, *news*. When the user taps the **Subscribe** button, the app subscribes to the *news* topic. In the following section, a *news* topic message will be sent from the Firebase Console Notifications GUI.

Send a topic message

Uninstall the app, rebuild it, and run it again. Click the **Subscribe to Notifications** button:

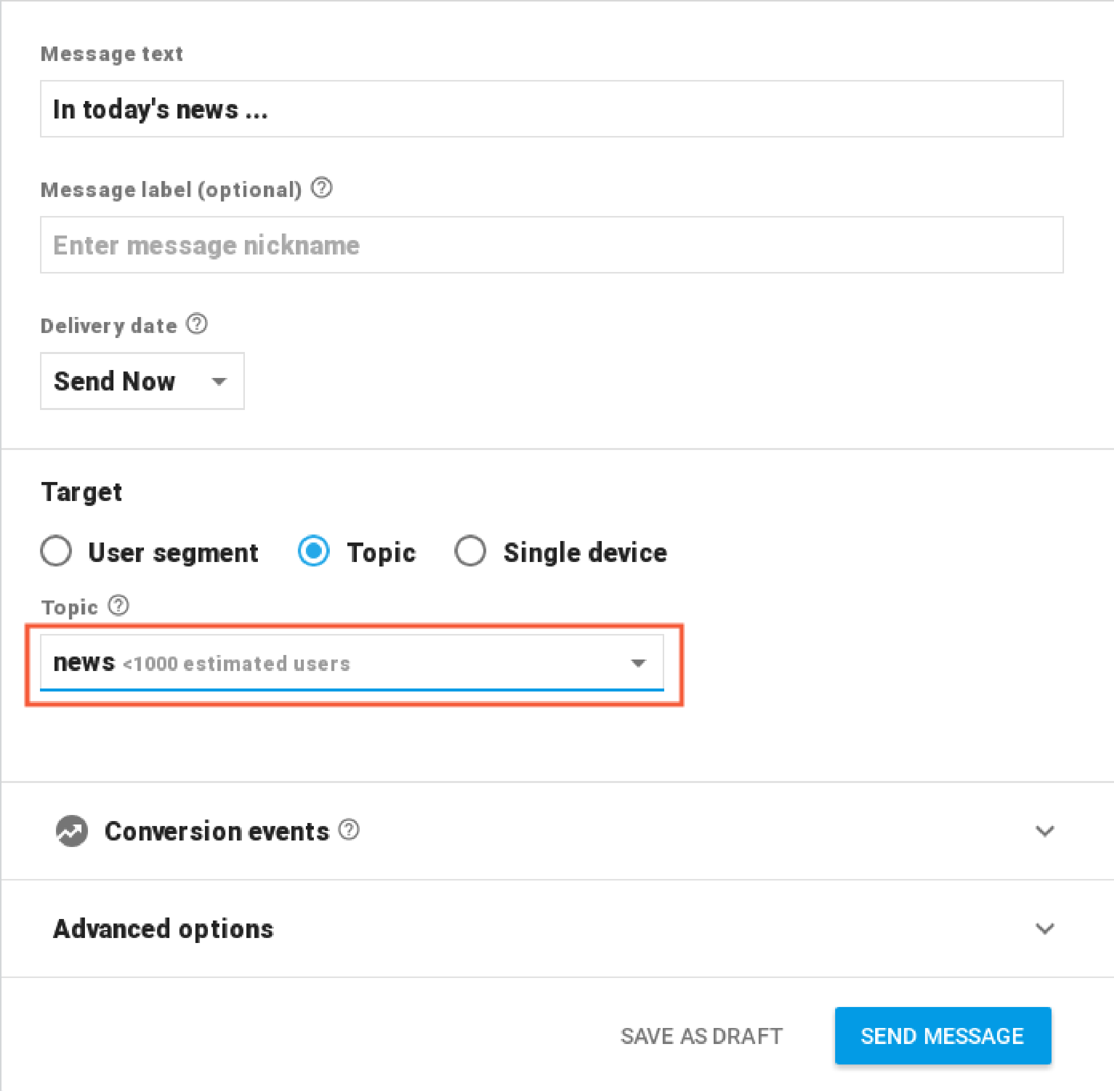


If the app has subscribed successfully, you should see **topic sync succeeded** in the IDE output window:



Use the following steps to send a topic message:

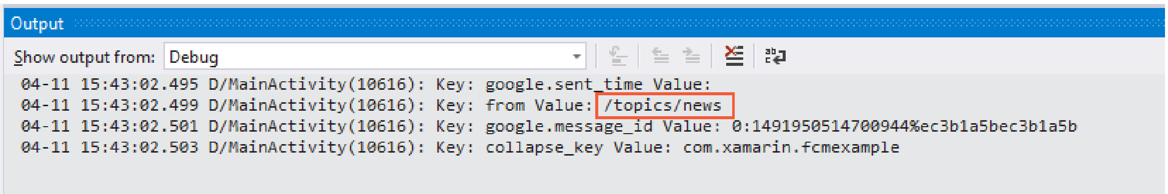
1. In the Firebase Console, click **NEW MESSAGE**.
2. On the **Compose message** page, enter the message text and select **Topic**.
3. In the **Topic** pull-down menu, select the built-in topic, **news**:



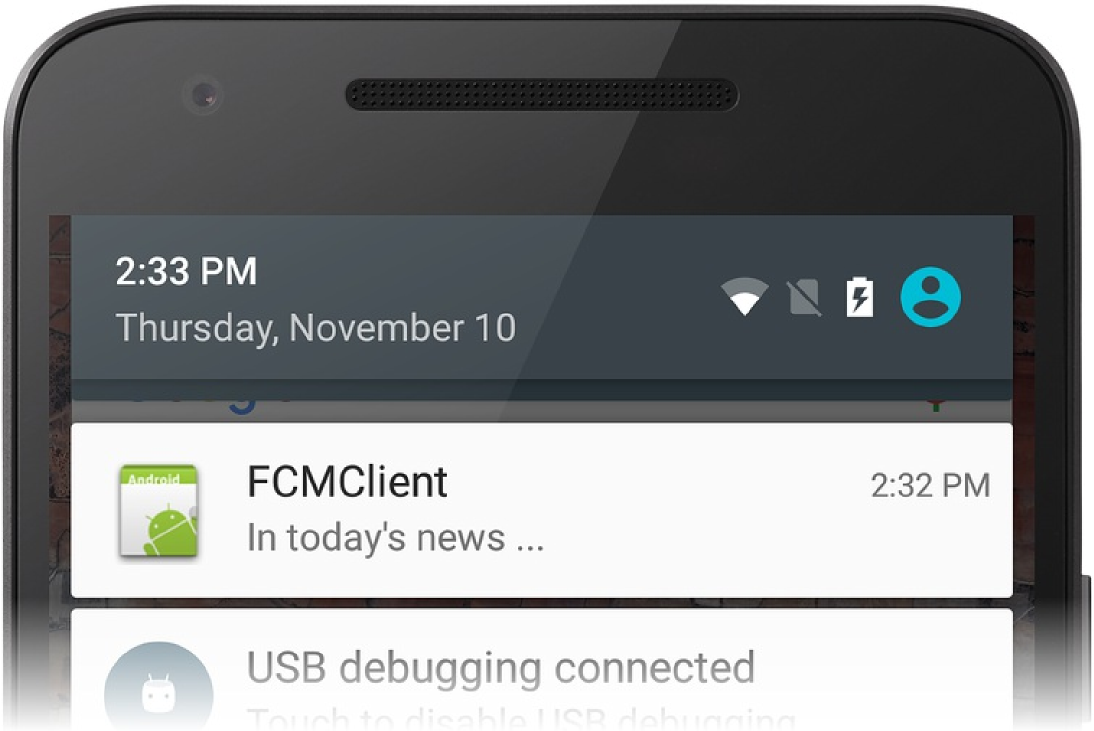
4. On the Android device (or emulator), background the app by tapping the Android **Overview** button and touching the home screen.

5. When the device is ready, click **SEND MESSAGE** in the Firebase Console.

6. Check the IDE output window to see **/topics/news** in the log output:



When this message is seen in the output window, the notification icon should also appear in the notification area on the Android device. Open the notification icon to view the topic message:



If you do not receive a message, try deleting the **FCMClient** app on the device (or emulator) and repeat the above steps.

Foreground notifications

To receive notifications in foregrounded apps, you must implement `FirebaseMessagingService`. This service is also required for receiving data payloads and for sending upstream messages. The following examples illustrate how to implement a service that extends `FirebaseMessagingService` – the resulting app will be able to handle remote notifications while it is running in the foreground.

Implement FirebaseMessagingService

The `FirebaseMessagingService` service is responsible for receiving and processing the messages from Firebase. Each app must subclass this type and override the `OnMessageReceived` to process an incoming message. When an app is in the foreground, the `OnMessageReceived` callback will always handle the message.

> **NOTE**
>
> Apps only have 10 seconds in which to handle an incoming Firebase Cloud Message. Any work that takes longer than this should be scheduled for background execution using a library such as the Android Job Scheduler or the Firebase Job Dispatcher.

Add a new file called **MyFirebaseMessagingService.cs** and replace its template code with the following:

```
using System;
using Android.App;
using Android.Content;
using Android.Media;
using Android.Util;
using Firebase.Messaging;

namespace FCMClient
{
    [Service]
    [IntentFilter(new[] { "com.google.firebase.MESSAGING_EVENT" })]
    public class MyFirebaseMessagingService : FirebaseMessagingService
    {
        const string TAG = "MyFirebaseMsgService";
        public override void OnMessageReceived(RemoteMessage message)
        {
            Log.Debug(TAG, "From: " + message.From);
            Log.Debug(TAG, "Notification Message Body: " + message.GetNotification().Body);
        }
    }
}
```

Note that the `MESSAGING_EVENT` intent filter must be declared so that new FCM messages are directed to `MyFirebaseMessagingService`:

```
[IntentFilter(new[] { "com.google.firebase.MESSAGING_EVENT" })]
```

When the client app receives a message from FCM, `OnMessageReceived` extracts the message content from the passed-in `RemoteMessage` object by calling its `GetNotification` method. Next, it logs the message content so that it can be viewed in the IDE output window:

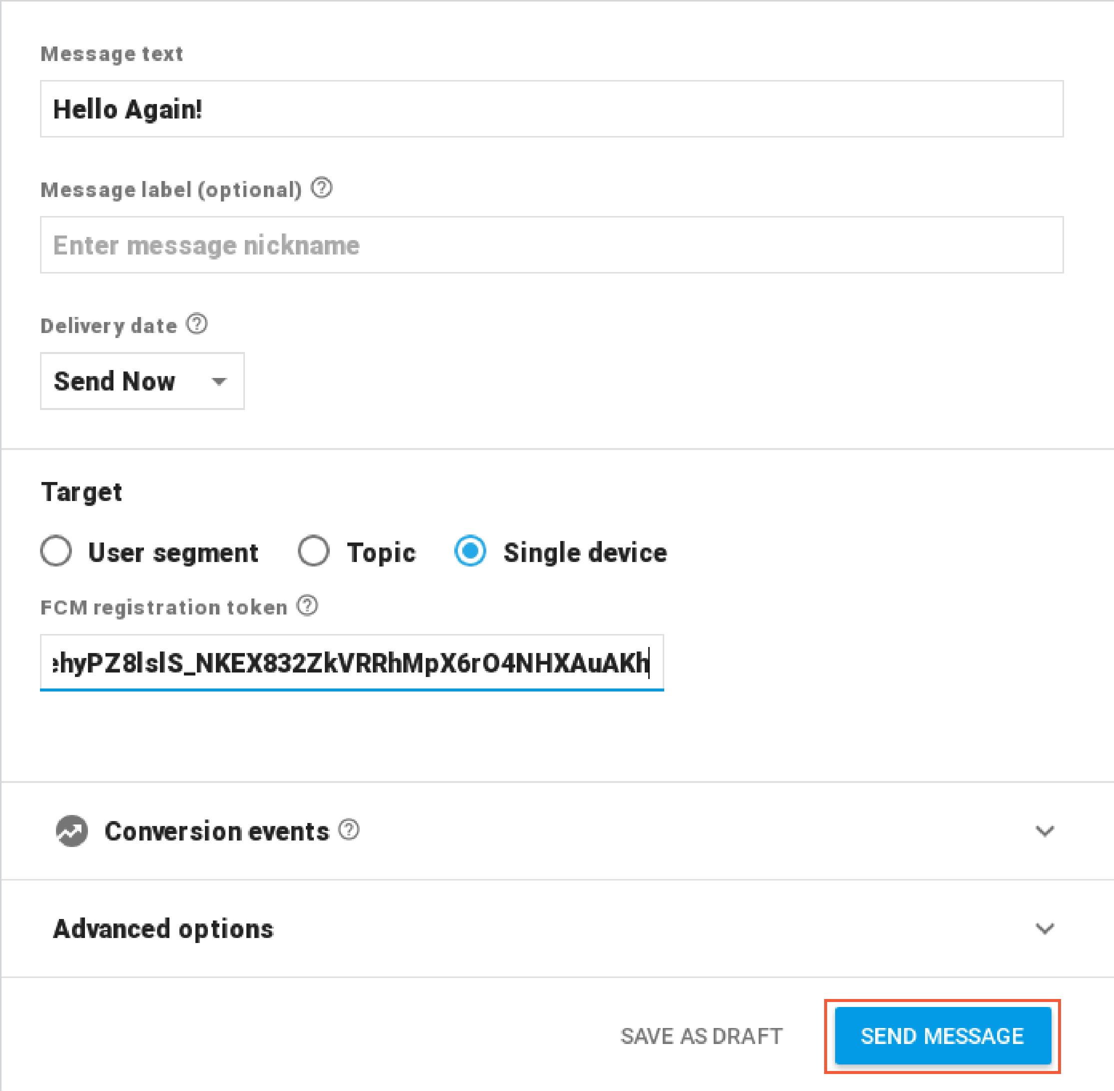
```
var body = message.GetNotification().Body;
Log.Debug(TAG, "Notification Message Body: " + body);
```

> **NOTE**
> If you set breakpoints in `FirebaseMessagingService`, your debugging session may or may not hit these breakpoints because of how FCM delivers messages.

Send another message

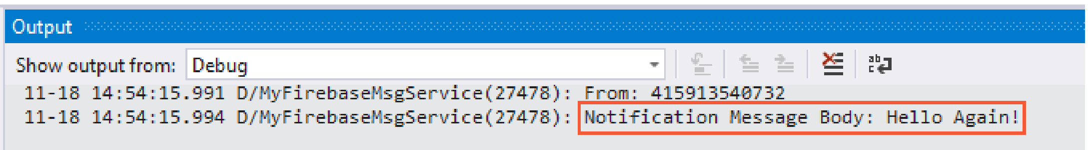
Uninstall the app, rebuild it, run it again, and follow these steps to send another message:

1. In the Firebase Console, click **NEW MESSAGE**.
2. On the **Compose message** page, enter the message text and select **Single device**.
3. Copy the token string from the IDE output window and paste it into the **FCM registration token** field of the Firebase Console as before.
4. Ensure that the app is running in the foreground, then click **SEND MESSAGE** in the Firebase Console:



5. When the **Review message** dialog is displayed, click **SEND**.

6. The incoming message is logged to the IDE output window:



Add a local notification sender

In this remaining example, the incoming FCM message will be converted into a local notification that is launched while the app is running in the foreground. Edit **MyFirebaseMessageService.cs** and add the following `using` statements:

```
using FCMClient;
using System.Collections.Generic;
```

Add the following method to `MyFirebaseMessagingService`:

```
void SendNotification(string messageBody, IDictionary<string, string> data)
{
    var intent = new Intent(this, typeof(MainActivity));
    intent.AddFlags(ActivityFlags.ClearTop);
    foreach (var key in data.Keys)
    {
        intent.PutExtra(key, data[key]);
    }

    var pendingIntent = PendingIntent.GetActivity(this,
                                                  MainActivity.NOTIFICATION_ID,
                                                  intent,
                                                  PendingIntentFlags.OneShot);

    var notificationBuilder = new  NotificationCompat.Builder(this, MainActivity.CHANNEL_ID)
                              .SetSmallIcon(Resource.Drawable.ic_stat_ic_notification)
                              .SetContentTitle("FCM Message")
                              .SetContentText(messageBody)
                              .SetAutoCancel(true)
                              .SetContentIntent(pendingIntent);

    var notificationManager = NotificationManagerCompat.From(this);
    notificationManager.Notify(MainActivity.NOTIFICATION_ID, notificationBuilder.Build());
}
```

To distinguish this notification from background notifications, this code marks notifications with an icon that differs from the application icon. Add the file ic_stat_ic_notification.png to **Resources/drawable** and include it in the **FCMClient** project.

The `SendNotification` method uses `NotificationCompat.Builder` to create the notification, and `NotificationManagerCompat` is used to launch the notification. The notification holds a `PendingIntent` that will allow the user to open the app and view the contents of the string passed into `messageBody`. For more information about `NotificationCompat.Builder`, see Local Notifications.

Call the `SendNotification` method at end of the `OnMessageReceived` method:

```
public override void OnMessageReceived(RemoteMessage message)
{
    Log.Debug(TAG, "From: " + message.From);

    var body = message.GetNotification().Body;
    Log.Debug(TAG, "Notification Message Body: " + body);
    SendNotification(body, message.Data);
}
```

As a result of these changes, `SendNotification` will run whenever a notification is received while the app is in the foreground, and the notification will appear in the notification area.

When an app is in the background, the payload of the message will determine how the message is handled:

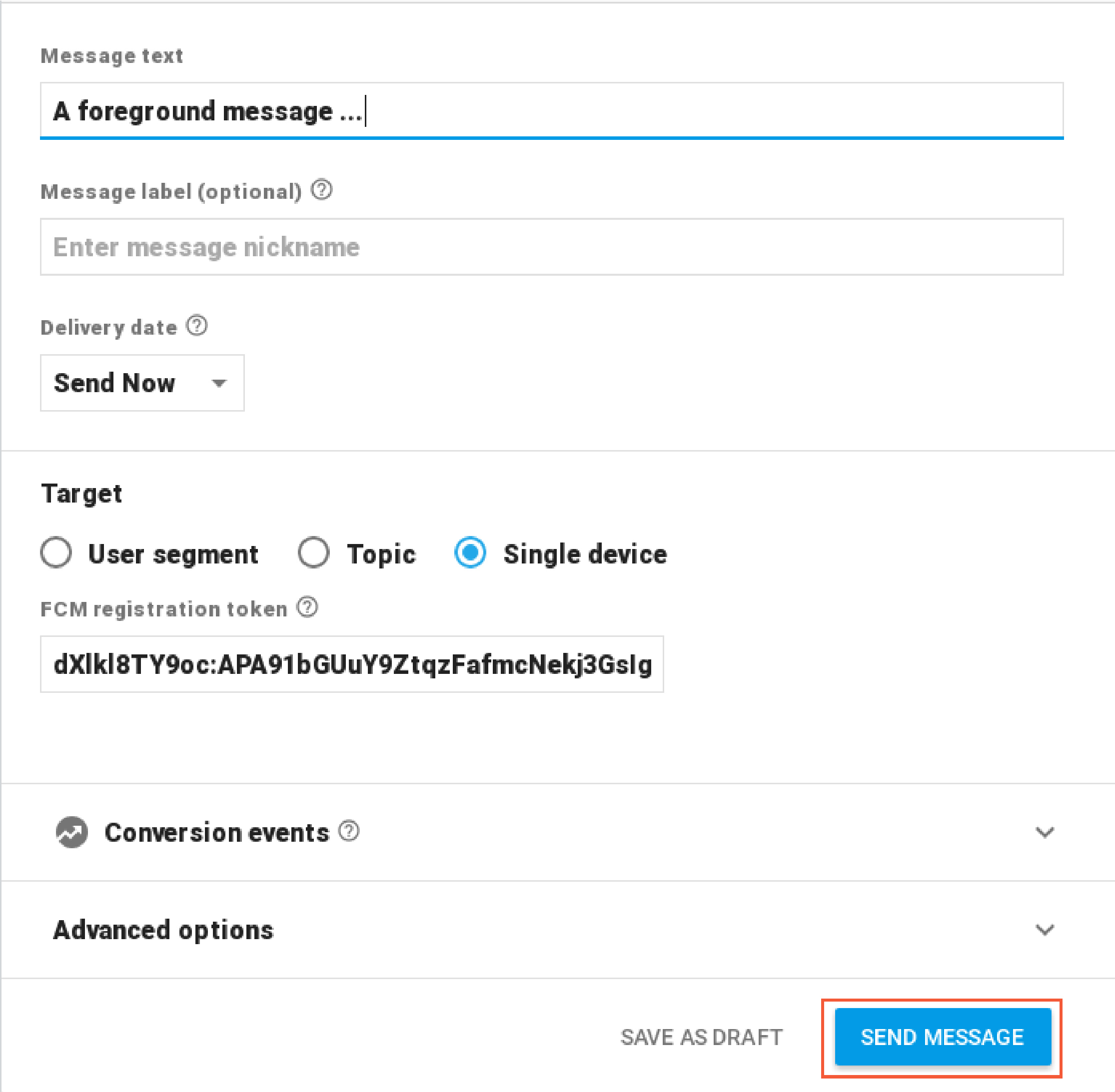
- **Notification** – messages will be sent to the **system tray**. A local notification will appear there. When the user taps on the notification the app will launch.
- **Data** – messages will be handled by `OnMessageReceived`.
- **Both** – messages that have both a notification and data payload will be delivered to the system tray. When the app launches, the data payload will appear in the `Extras` of the `Intent` that was used to start the app.

In this example, if the app is backgrounded, `SendNotification` will run if the message has a data payload. Otherwise, a background notification (illustrated earlier in this walkthrough) will be launched.

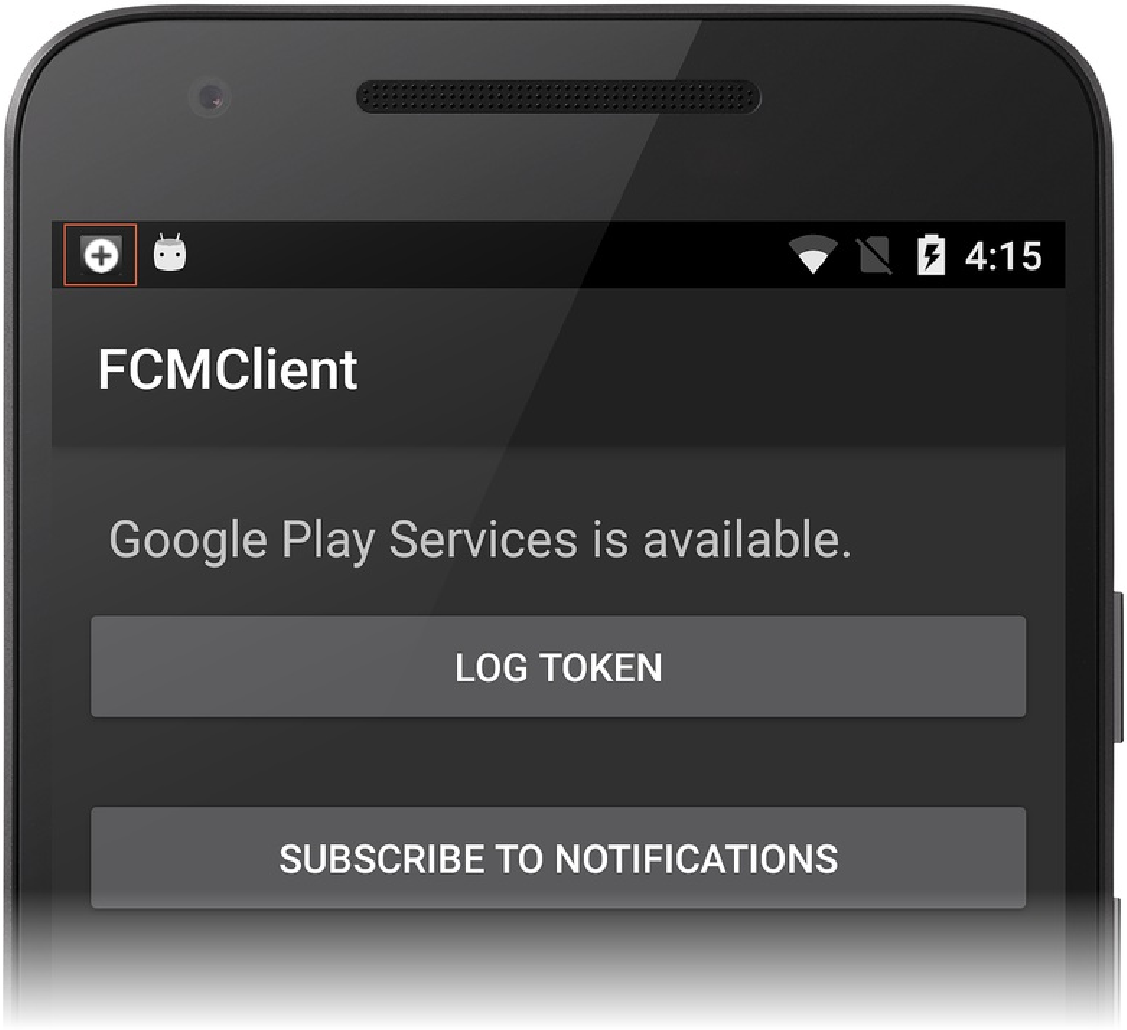
Send the last message

Uninstall the app, rebuild it, run it again, then use the following steps to send the last message:

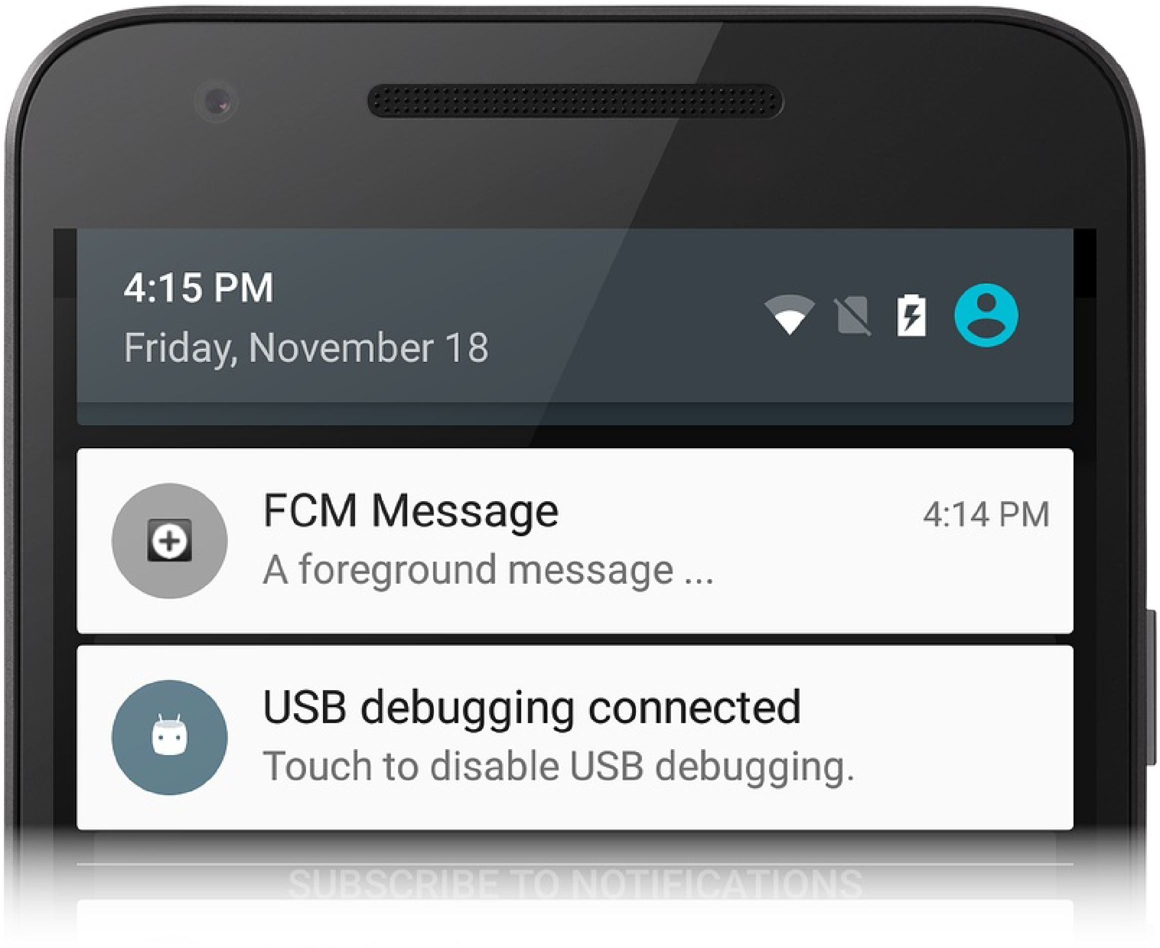
1. In the Firebase Console, click **NEW MESSAGE**.
2. On the **Compose message** page, enter the message text and select **Single device**.
3. Copy the token string from the IDE output window and paste it into the **FCM registration token** field of the Firebase Console as before.
4. Ensure that the app is running in the foreground, then click **SEND MESSAGE** in the Firebase Console:



This time, the message that was logged in the output window is also packaged in a new notification – the notification icon appears in the notification tray while the app is running in the foreground:



When you open the notification, you should see the last message that was sent from the Firebase Console Notifications GUI:



Disconnecting from FCM

To unsubscribe from a topic, call the UnsubscribeFromTopic method on the FirebaseMessaging class. For example, to unsubscribe from the *news* topic subscribed to earlier, an **Unsubscribe** button could be added to the layout with the following handler code:

```
var unSubscribeButton = FindViewById<Button>(Resource.Id.unsubscribeButton);
unSubscribeButton.Click += delegate {
    FirebaseMessaging.Instance.UnsubscribeFromTopic("news");
    Log.Debug(TAG, "Unsubscribed from remote notifications");
};
```

To unregister the device from FCM altogether, delete the instance ID by calling the DeleteInstanceId method on the FirebaseInstanceId class. For example:

```
FirebaseInstanceId.Instance.DeleteInstanceId();
```

This method call deletes the instance ID and the data associated with it. As a result, the periodic sending of FCM data to the device is halted.

Troubleshooting

The following describe issues and workarounds that may arise when using Firebase Cloud Messaging with Xamarin.Android.

FirebaseApp is not Initialized

In some cases, you may see this error message:

```
Java.Lang.IllegalStateException: Default FirebaseApp is not initialized in this process
Make sure to call FirebaseApp.initializeApp(Context) first.
```

This is a known problem that you can work around by cleaning the solution and rebuilding the project (**Build > Clean Solution**, **Build > Rebuild Solution**).

Summary

This walkthrough detailed the steps for implementing Firebase Cloud Messaging remote notifications in a Xamarin.Android application. It described how to install the required packages needed for FCM communications, and it explained how to configure the Android Manifest for access to FCM servers. It provided example code that illustrates how to check for the presence of Google Play Services. It demonstrated how to implement an instance ID listener service that negotiates with FCM for a registration token, and it explained how this code creates background notifications while the app is backgrounded. It explained how to subscribe to topic messages, and it provided an example implementation of a message listener service that is used to receive and display remote notifications while the app is running in the foreground.

Related links

- FCMNotifications (sample)
- Firebase Cloud Messaging
- About FCM Messages

Google Cloud Messaging

7/8/2021 • 8 minutes to read • Edit Online

WARNING

Google deprecated GCM as of April 10, 2018. The following docs and sample projects may no longer be maintained. Google's GCM server and client APIs will be removed as soon as May 29, 2019. Google recommends migrating GCM apps to Firebase Cloud Messaging (FCM). For more information about GCM deprecation and migration, see Google Deprecated Cloud Messaging.

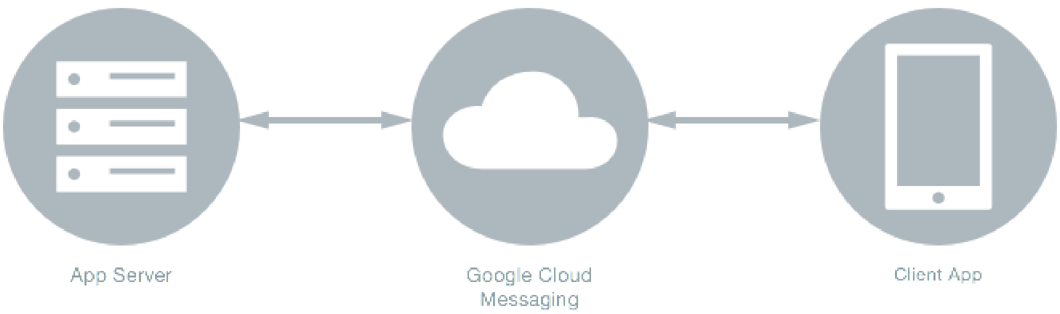
To start using Firebase Cloud Messaging with Xamarin, see Firebase Cloud Messaging.

Google Cloud Messaging (GCM) is a service that facilitates messaging between mobile apps and server applications. This article provides an overview of how GCM works, and it explains how to configure Google Services so your app can use GCM.

This topic provides a high-level overview of how Google Cloud Messaging routes messages between your app and an app server, and it provides a step-by-step procedure for acquiring credentials so that your app can use GCM services.

Overview

Google Cloud Messaging (GCM) is a service that handles the sending, routing, and queueing of messages between server applications and mobile client apps. A *client app* is a GCM-enabled app that runs on a device. The *app server* (provided by you or your company) is the GCM-enabled server that your client app communicates with through GCM:



Using GCM, app servers can send messages to a single device, a group of devices, or a number of devices that are subscribed to a topic. Your client app can use GCM to subscribe to downstream messages from an app server (for example, to receive remote notifications). Also, GCM makes it possible for client apps to send upstream messages back to the app server.

Google Cloud Messaging in Action

When downstream messages are sent from an app server to a client app, the app server sends the message to a *GCM connection server*; the GCM connection server, in turn, forwards the message to a device that is running your client app. Messages can be sent over HTTP or XMPP (Extensible Messaging and Presence Protocol). Because client apps are not always connected or running, the GCM connection server enqueues and stores messages, sending them to client apps as they reconnect and become available. Similarly, GCM will enqueue upstream messages from the client app to the app server if the app server is unavailable.

GCM uses the following credentials to identify the app server and your client app, and it uses these credentials to authorize message transactions through GCM:

- **API Key** – The *API key* gives your app server access to Google services; GCM uses this key to authenticate your app server. Before you can use the GCM service, you must first obtain an API key from the Google Developer Console by creating a *project*. The API Key should be kept secure; for more information about protecting your API key, see Best practices for securely using API keys.

- **Sender ID** – The *Sender ID* authorizes the app server to your client app – it is a unique number that identifies the app server that is permitted to send messages to your client app. The sender ID is also your project number; you obtain the sender ID from the Google Developers Console when you register your project.

- **Registration Token** – The *Registration Token* is the GCM identity of your client app on a given device. The registration token is generated at run time – your app receives a registration token when it first registers with GCM while running on a device. The registration token authorizes an instance of your client app (running on that particular device) to receive messages from GCM.

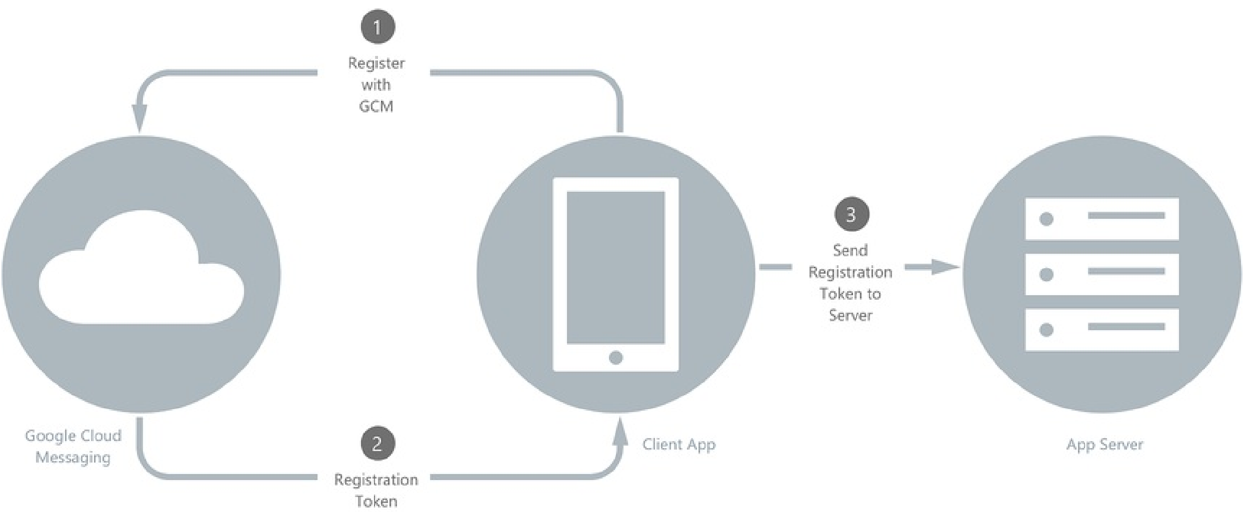
- **Application ID** – The identity of your client app (independent of any given device) that registers to receive messages from GCM. On Android, the application ID is the package name recorded in **AndroidManifest.xml**, such as `com.xamarin.gcmexample`.

Setting Up Google Cloud Messaging (later in this guide) provides detailed instructions for creating a project and generating these credentials.

The following sections explain how these credentials are used when client apps communicate with app servers through GCM.

Registration with GCM

A client app installed on a device must first register with GCM before messaging can take place. The client app must complete the registration steps shown in the following diagram:



1. The client app contacts GCM to obtain a registration token, passing the sender ID to GCM.

2. GCM returns a registration token to the client app.

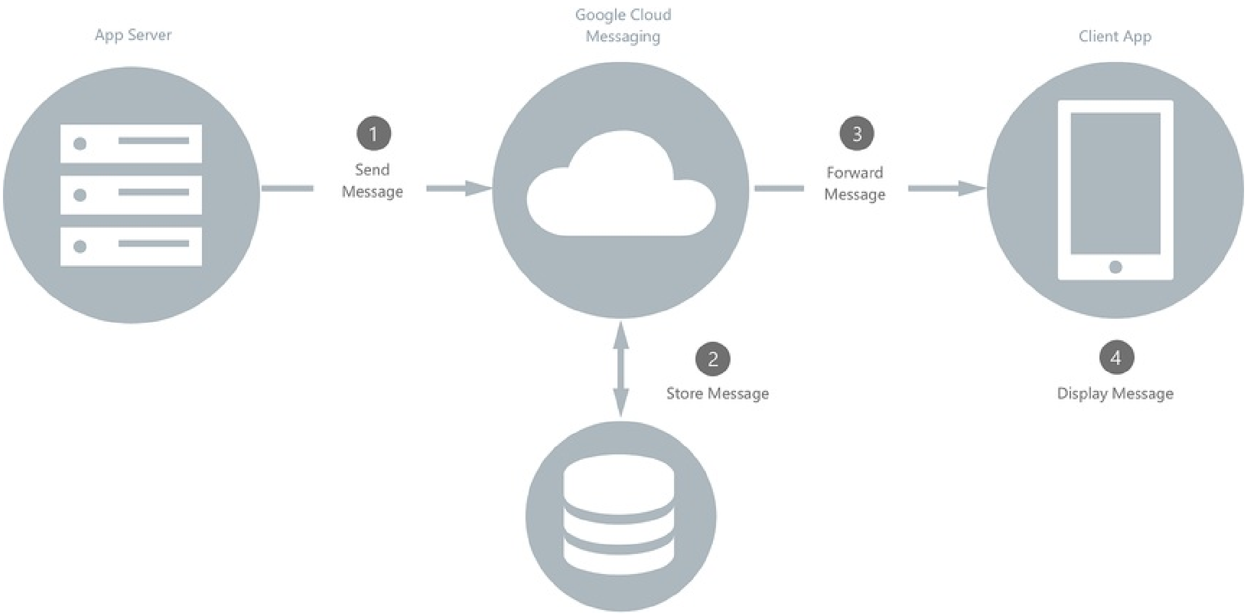
3. The client app forwards the registration token to the app server.

The app server caches the registration token for subsequent communications with the client app. Optionally, the app server can send an acknowledgement back to the client app to indicate that the registration token was received. After this handshake takes place, the client app can receive messages from (or send messages to) the app server.

When the client app no longer wants to receive messages from the app server, it can send a request to the app server to delete the registration token. If the client app is receiving topic messages (explained later in this article), it can unsubscribe from the topic. If the client app is uninstalled from a device, GCM detects this and automatically notifies the app server to delete the registration token.

Downstream Messaging

When the app server sends a downstream message to the client app, it follows the steps illustrated in the following diagram:



1. The app server sends the message to GCM.

2. If the client device is not available, the GCM server stores the message in a queue for later transmission.

3. When the client device is available, GCM sends the message to the client app on that device.

4. The client app receives the message from GCM and handles it accordingly. For example, if the message is a remote notification, it is presented to the user.

In this messaging scenario (where the app server sends a message to a single client app), messages can be up to 4kB in length.

For detailed information (including code samples) about receiving downstream GCM messages on Android, see Remote Notifications.

Topic Messaging

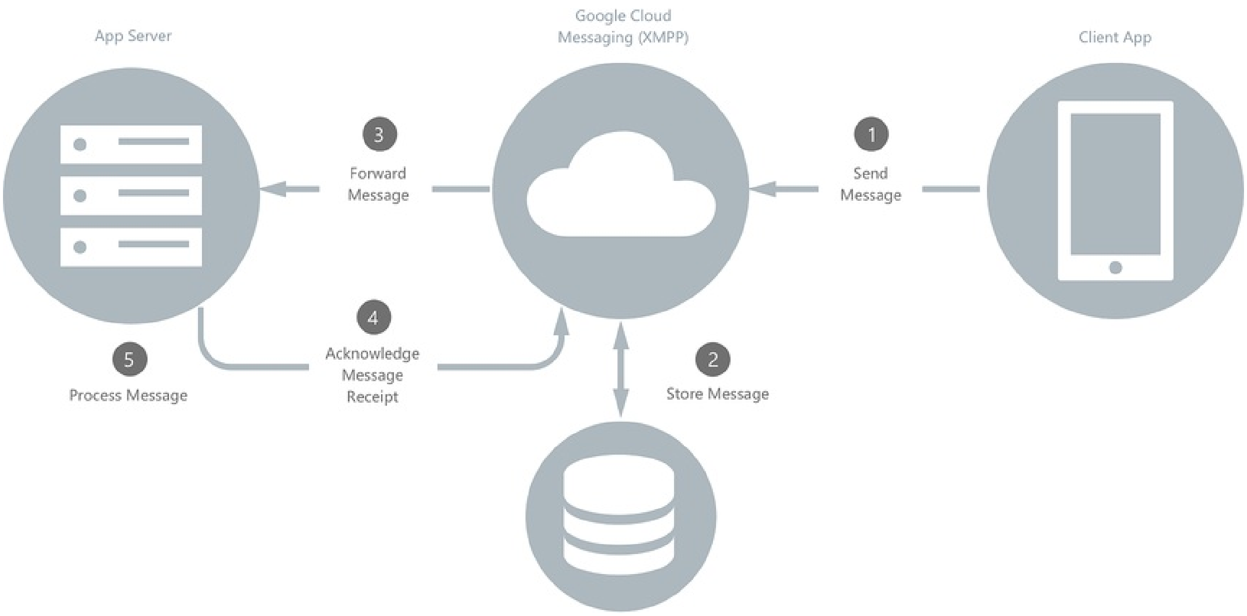
Topic Messaging is a type of downstream messaging where the app server sends a single message to multiple client app devices that subscribe to a topic (such as a weather forecast). Topic messages can be up to 2KB in length, and topic messaging supports up to one million subscriptions per app. If GCM is being used only for topic messaging, the client app is not required to send a registration token to the app server.

Group Messaging

Group Messaging is a type of downstream messaging where the app server sends a single message to multiple client app devices that belong to a group (for example, a group of devices that belong to a single user). Group messages can be up to 2KB in length for iOS devices, and up to 4KB in length for Android devices. A group is limited to a maximum of 20 members.

Upstream Messaging

If your client app connects to a server that supports XMPP, it can send messages back to the app server as illustrated in the following diagram:



1. The client app sends a message to the GCM XMPP connection server.

2. If the app server is disconnected, the GCM server stores the message in a queue for later forwarding.

3. When the app server is re-connected, GCM forwards the message to the app server.

4. The app server parses the message to verify the identity of the client app, then it sends an "ack" to GCM to acknowledge message receipt.

5. The app server processes the message.

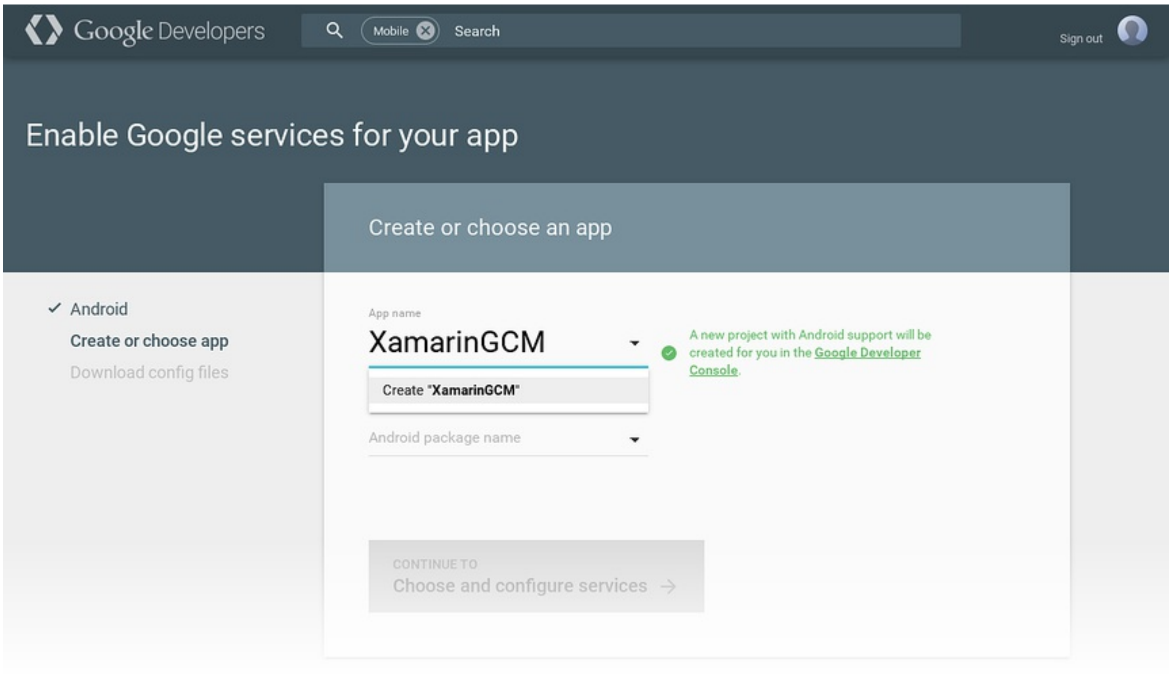
Google's Upstream Messages explains how to structure JSON-encoded messages and send them to app servers that run Google's XMPP-based Cloud Connection Server.

Setting Up Google Cloud Messaging

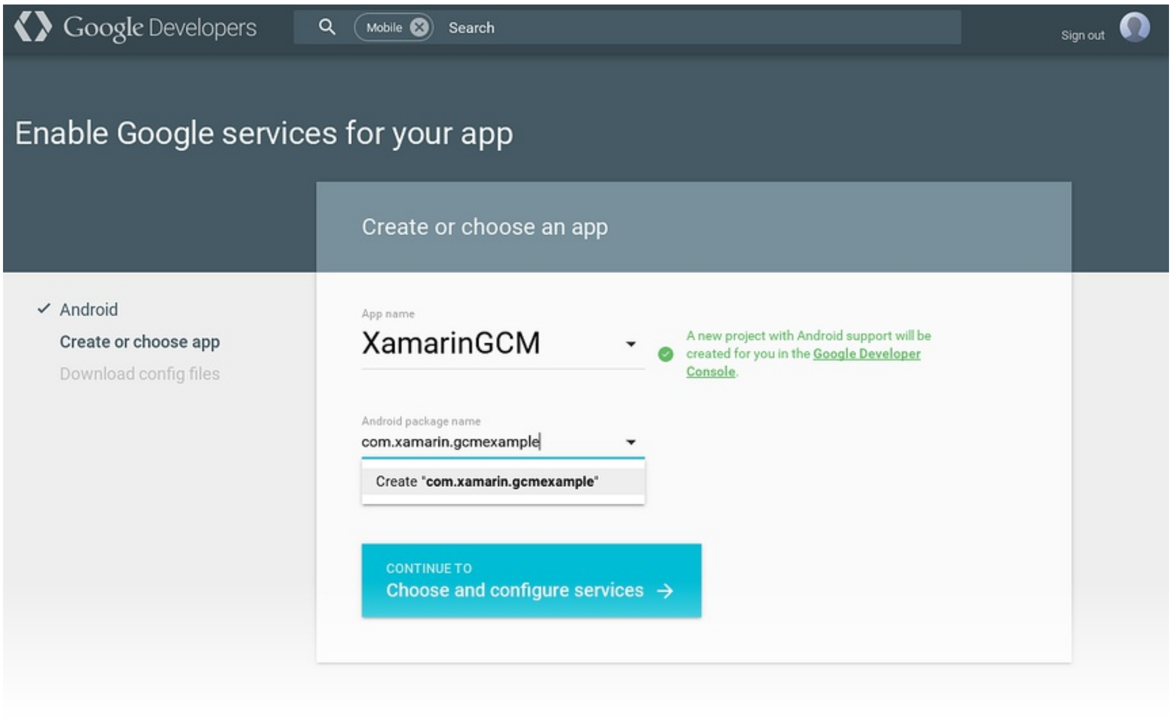
Before you can use GCM services in your app, you must first acquire credentials for access to Google's GCM servers. The following sections describe the steps required to complete this process:

Enable Google Services for Your App

1. Sign into the Google Developers Console with your Google account (i.e, your gmail address) and create a new project. If you have an existing project, choose the project that you want to become GCM-enabled. In the following example, a new project called **XamarinGCM** is created:

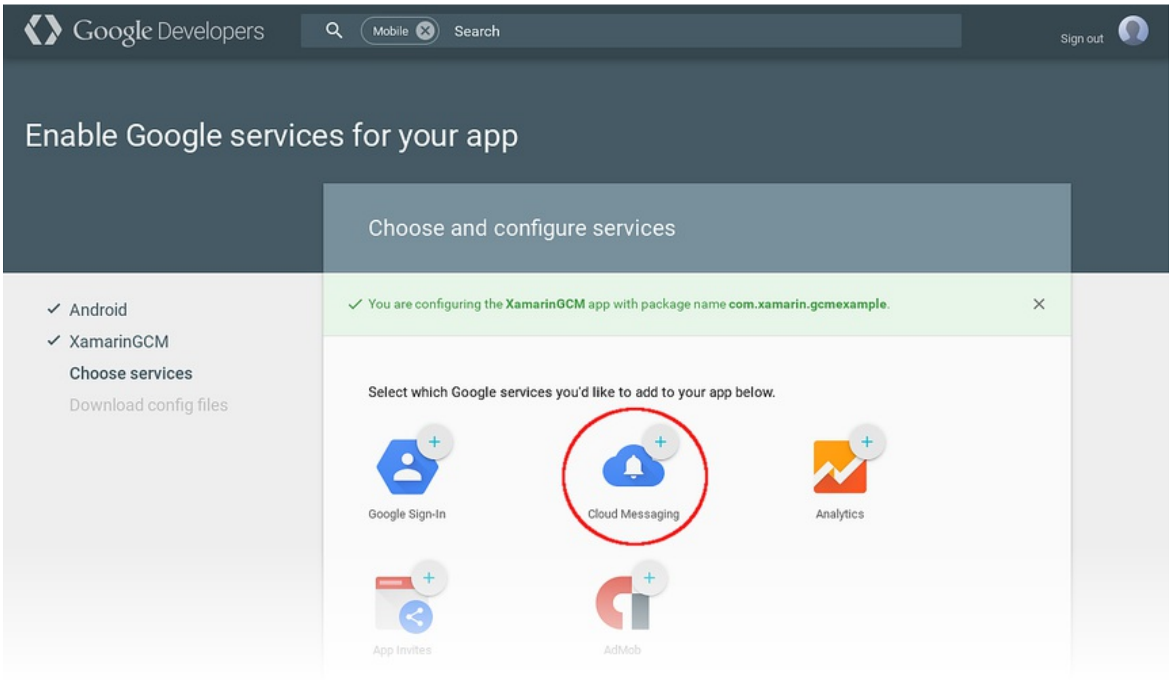


2. Next, enter the package name for your app (in this example, the package name is **com.xamarin.gcmexample**) and click **Continue to Choose and configure services**:

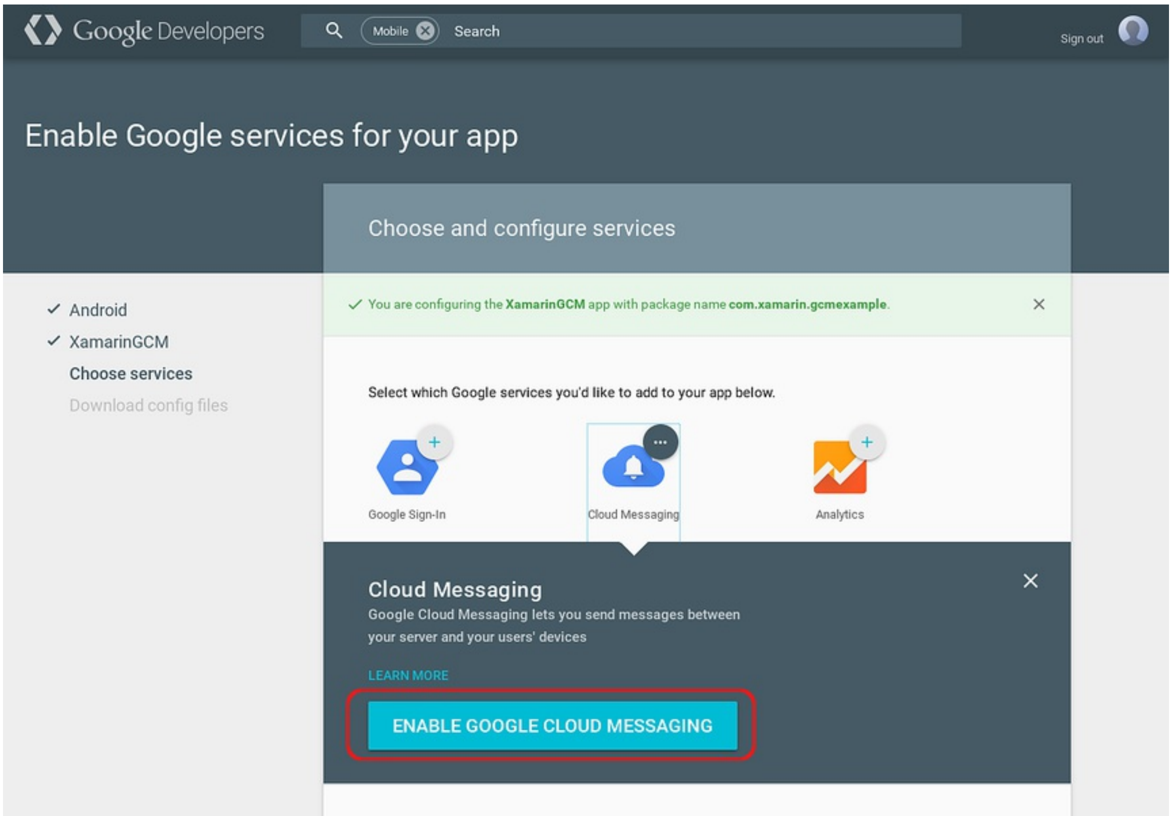


Note that this package name is also the application ID for your app.

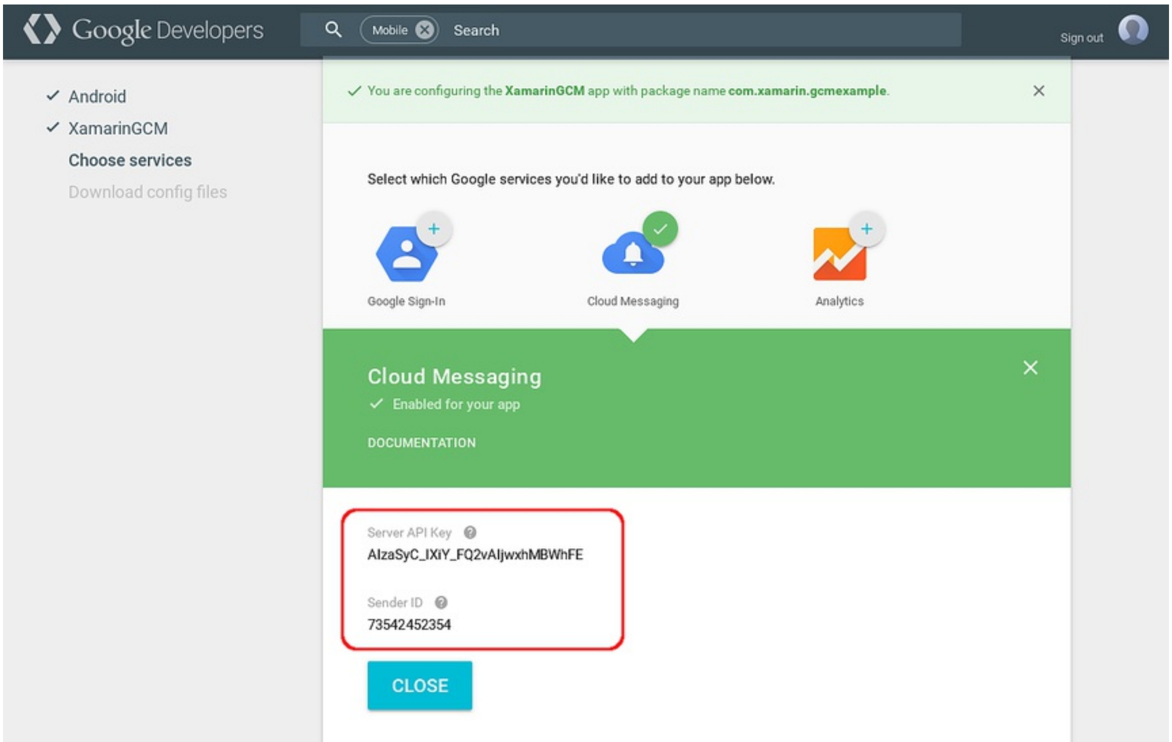
3. The **Choose and configure services** section lists the Google services that you can add to your app. Click **Cloud Messaging**:



4. Next, click **ENABLE GOOGLE CLOUD MESSAGING**:



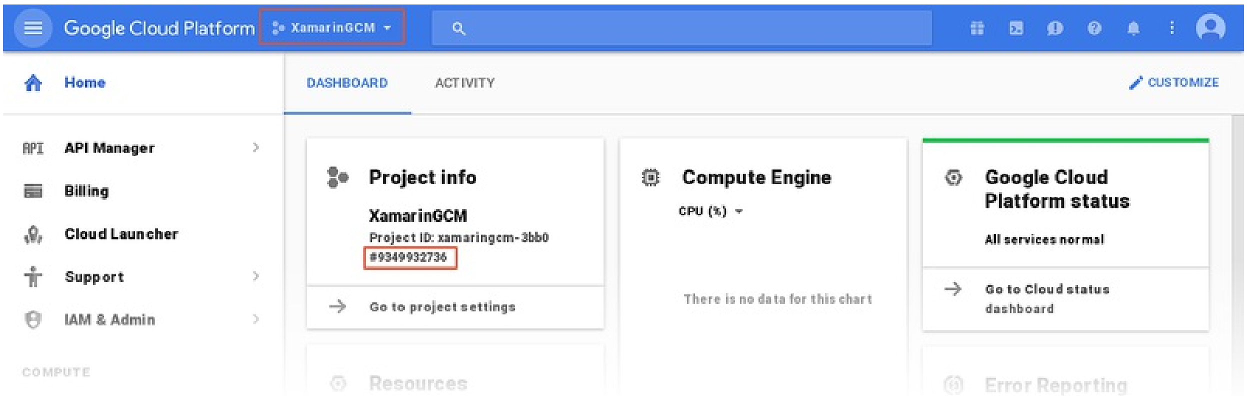
5. A **Server API key** and a **Sender ID** are generated for your app. Record these values and click **CLOSE**:



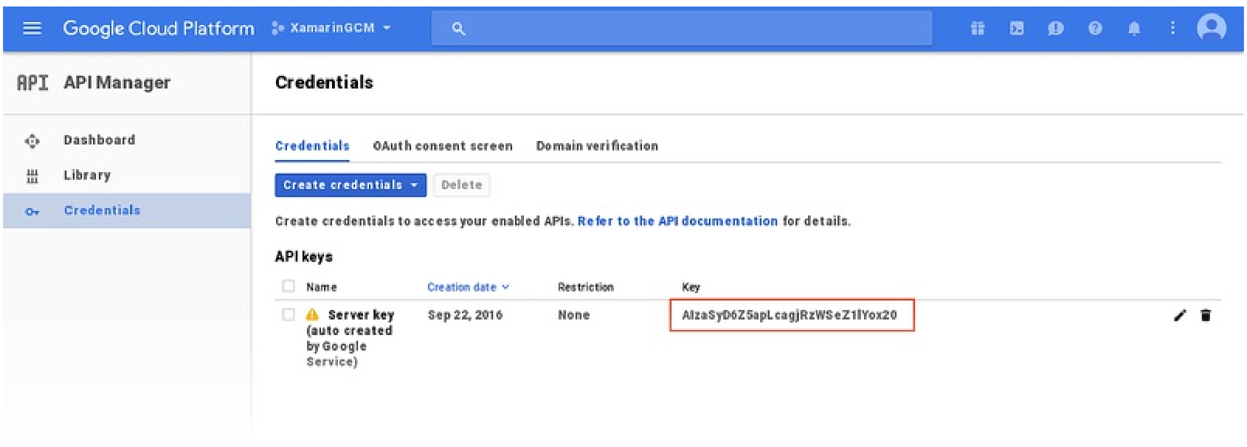
Protect the API key – it is not intended for public use. If the API key is compromised, unauthorized servers could publish messages to client applications. Best practices for securely using API keys provides useful guidelines for protecting your API Key.

View Your Project Settings

You can view your project settings at any time by signing into the Google Cloud Console and selecting your project. For example, you can view the **Sender ID** by selecting your project in the pull down menu at the top of the page (in this example, the project is called **XamarinGCM**). The Sender ID is the Project number as shown in this screenshot (the Sender ID here is **9349932736**):



To view the **API key**, click **API Manager** and then click **Credentials**:



For Further Reading

- RFC 6120 and RFC 6121 explain and define the Extensible Messaging and Presence Protocol (XMPP).

Summary

This article provided an overview of Google Cloud Messaging (GCM). It explained the various credentials that are used to identify and authorize messaging between app servers and client apps. It illustrated the most common messaging scenarios, and it detailed the steps for registering your app with GCM to use GCM services.

Related Links

- Cloud Messaging

Remote Notifications With Google Cloud Messaging

7/8/2021 • 20 minutes to read • Edit Online

WARNING

Google deprecated GCM as of April 10, 2018. The following docs and sample projects may no longer be maintained. Google's GCM server and client APIs will be removed as soon as May 29, 2019. Google recommends migrating GCM apps to Firebase Cloud Messaging (FCM). For more information about GCM deprecation and migration, see Google Cloud Messaging - DEPRECATED.

To get started with Remote Notifications using Firebase Cloud Messaging with Xamarin, see Remote Notifications with FCM.

This walkthrough provides a step-by-step explanation of how to use Google Cloud Messaging to implement remote notifications (also called push notifications) in a Xamarin.Android application. It describes the various classes that you must implement to communicate with Google Cloud Messaging (GCM), it explains how to set permissions in the Android Manifest for access to GCM, and it demonstrates end-to-end messaging with a sample test program.

GCM Notifications Overview

In this walkthrough, we'll create a Xamarin.Android application that uses Google Cloud Messaging (GCM) to implement remote notifications (also known as *push notifications*). We'll implement the various intent and listener services that use GCM for remote messaging, and we'll test our implementation with a command-line program that simulates an application server.

Before you can proceed with this walkthrough, you must acquire the necessary credentials to use Google's GCM servers; this process is explained in Google Cloud Messaging. In particular, you will need an *API Key* and a *Sender ID* to insert into the example code presented in this walkthrough.

We'll use the following steps to create a GCM-enabled Xamarin.Android client app:

1. Install additional packages required for communications with GCM servers.
2. Configure app permissions for access to GCM servers.
3. Implement code to check for the presence of Google Play Services.
4. Implement a registration intent service that negotiates with GCM for a registration token.
5. Implement an instance ID listener service that listens for registration token updates from GCM.
6. Implement a GCM listener service that receives remote messages from the app server through GCM.

This app will use a new GCM feature known as *topic messaging*. In topic messaging, the app server sends a message to a topic, rather than to a list of individual devices. Devices that subscribe to that topic can receive topic messages as push notifications.

When the client app is ready, we'll implement a command-line C# application that sends a push notification to our client app via GCM.

Walkthrough

To begin, let's create a new empty Solution called **RemoteNotifications**. Next, let's add a new Android project to this Solution that is based on the **Android App** template. Let's call this project **ClientApp**. (If you're not familiar with creating Xamarin.Android projects, see Hello, Android.) The **ClientApp** project will contain the code for the Xamarin.Android client application that receives remote notifications via GCM.

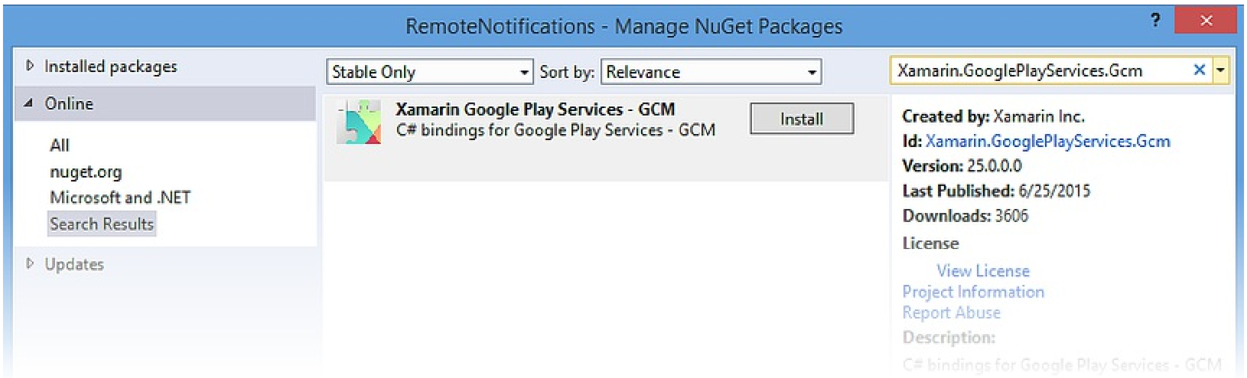
Add Required Packages

Before we can implement our client app code, we must install several packages that we'll use for communication with GCM. Also, we must add the Google Play Store application to our device if it is not already installed.

Add the Xamarin Google Play Services GCM Package

To receive messages from Google Cloud Messaging, the Google Play Services framework must be present on the device. Without this framework, an Android application cannot receive messages from GCM servers. Google Play Services runs in the background while the Android device is powered on, quietly listening for messages from GCM. When these messages arrive, Google Play Services converts the messages into intents and then broadcasts these intents to applications that have registered for them.

In Visual Studio, right-click **References > Manage NuGet Packages** ...; in Visual Studio for Mac, right-click **Packages > Add Packages...**. Search for **Xamarin Google Play Services - GCM** and install this package into the **ClientApp** project:



When you install **Xamarin Google Play Services - GCM**, **Xamarin Google Play Services - Base** is automatically installed. If you get an error, change the project's *Minimum Android to target* setting to a value other than **Compile using SDK version** and try the NuGet install again.

Next, edit **MainActivity.cs** and add the following `using` statements:

```
using Android.Gms.Common;
using Android.Util;
```

This makes types in the Google Play Services GMS package available to our code, and it adds logging functionality that we will use to track our transactions with GMS.

Google Play Store

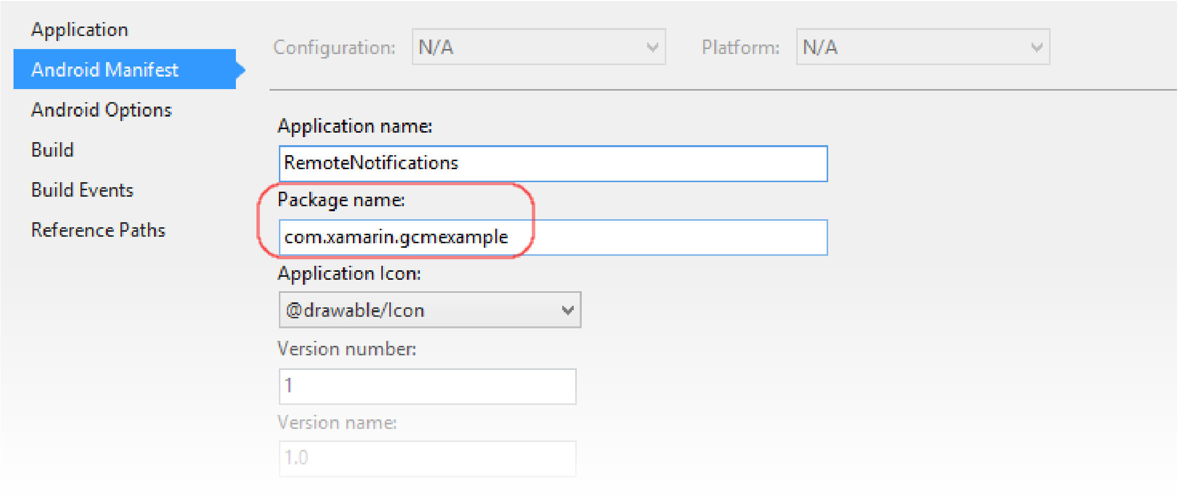
To receive messages from GCM, the Google Play Store application must be installed on the device. (Whenever a Google Play application is installed on a device, Google Play Store is also installed, so it's likely that it is already installed on your test device.) Without Google Play, an Android application cannot receive messages from GCM. If you do not yet have the Google Play Store app installed on your device, visit the Google Play web site to download and install Google Play.

Alternately, you can use an Android emulator running Android 2.2 or later instead of a test device (you do not have to install Google Play Store on an Android emulator). However, if you use an emulator, you must use Wi-Fi to connect to GCM and you must open several ports in your Wi-Fi firewall as explained later in this walkthrough.

Set the Package Name

In Google Cloud Messaging, we specified a package name for our GCM-enabled app (this package name also serves as the *application ID* that is associated with our API key and Sender ID). Let's open the properties for the

ClientApp project and set the package name to this string. In this example, we set the package name to `com.xamarin.gcmexample`:



Note that the client app will be unable to receive a registration token from GCM if this package name does not *exactly* match the package name that we entered into the Google Developer console.

Add Permissions to the Android Manifest

An Android application must have the following permissions configured before it can receive notifications from Google Cloud Messaging:

- `com.google.android.c2dm.permission.RECEIVE` – Grants permission to our app to register and receive messages from Google Cloud Messaging. (What does `c2dm` mean? This stands for *Cloud to Device Messaging*, which is the now-deprecated predecessor to GCM. GCM still uses `c2dm` in many of its permission strings.)

- `android.permission.WAKE_LOCK` – (Optional) Prevents the device CPU from going to sleep while listening for a message.

- `android.permission.INTERNET` – Grants internet access so the client app can communicate with GCM.

- *package_name* `.permission.C2D_MESSAGE` – Registers the application with Android and requests permission to exclusively receive all C2D (cloud to device) messages. The *package_name* prefix is the same as your application ID.

We'll set these permissions in the Android manifest. Let's edit **AndroidManifest.xml** and replace the contents with the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="YOUR_PACKAGE_NAME"
    android:versionCode="1"
    android:versionName="1.0"
    android:installLocation="auto">
    <uses-permission android:name="com.google.android.c2dm.permission.RECEIVE" />
    <uses-permission android:name="android.permission.WAKE_LOCK" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="YOUR_PACKAGE_NAME.permission.C2D_MESSAGE" />
    <permission android:name="YOUR_PACKAGE_NAME.permission.C2D_MESSAGE"
                android:protectionLevel="signature" />
    <application android:label="ClientApp" android:icon="@drawable/Icon">
    </application>
</manifest>
```

In the above XML, change *YOUR_PACKAGE_NAME* to the package name for your client app project. For example, `com.xamarin.gcmexample`.

Check for Google Play Services

For this walkthrough, we're creating a bare-bones app with a single `TextView` in the UI. This app doesn't directly indicate interaction with GCM. Instead, we'll watch the output window to see how our app handshakes with GCM, and we'll check the notification tray for new notifications as they arrive.

First, let's create a layout for the message area. Edit **Resources.layout.Main.axml** and replace the contents with the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="10dp">
    <TextView
        android:text=" "
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/msgText"
        android:textAppearance="?android:attr/textAppearanceMedium"
        android:padding="10dp" />
</LinearLayout>
```

Save **Main.axml** and close it.

When the client app starts, we want it to verify that Google Play Services is available before we attempt to contact GCM. Edit **MainActivity.cs** and replace the `count` instance variable declaration with the following instance variable declaration:

```
TextView msgText;
```

Next, add the following method to the **MainActivity** class:

```
public bool IsPlayServicesAvailable ()
{
    int resultCode = GoogleApiAvailability.Instance.IsGooglePlayServicesAvailable (this);
    if (resultCode != ConnectionResult.Success)
    {
        if (GoogleApiAvailability.Instance.IsUserResolvableError (resultCode))
            msgText.Text = GoogleApiAvailability.Instance.GetErrorString (resultCode);
        else
        {
            msgText.Text = "Sorry, this device is not supported";
            Finish ();
        }
        return false;
    }
    else
    {
        msgText.Text = "Google Play Services is available.";
        return true;
    }
}
```

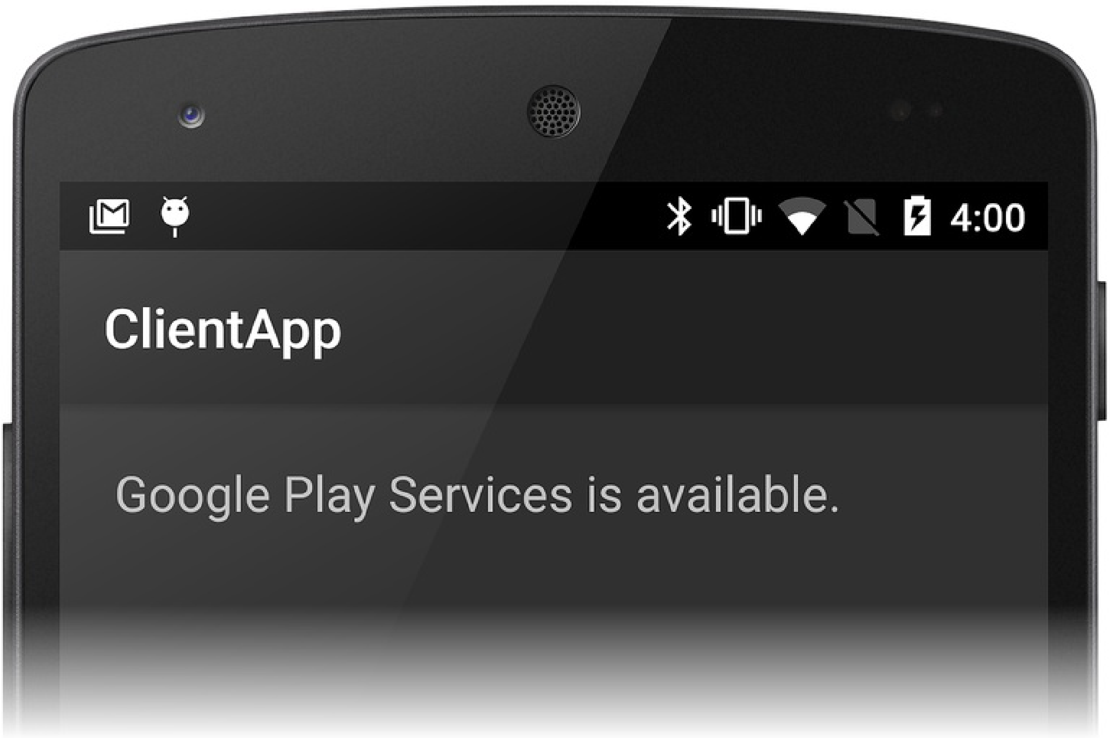
This code checks the device to see if the Google Play Services APK is installed. If it is not installed, a message is displayed in the message area that instructs the user to download an APK from the Google Play Store (or enable it in the device's system settings). Because we want to run this check when the client app starts, we'll add a call to this method at the end of `OnCreate`.

Next, replace the `OnCreate` method with the following code:

```
protected override void OnCreate (Bundle bundle)
{
    base.OnCreate (bundle);

    SetContentView (Resource.Layout.Main);
    msgText = FindViewById<TextView> (Resource.Id.msgText);

    IsPlayServicesAvailable ();
}
```

This code checks for the presence of the Google Play Services APK and writes the result to the message area.

Let's completely rebuild and run the app. You should see a screen that looks like the following screenshot:



If you don't get this result, verify that the Google Play Services APK is installed on your device and that the **Xamarin Google Play Services - GCM** package is added to your **ClientApp** project as explained earlier. If you get a build error, try cleaning the Solution and building the project again.

Next, we'll write code to contact GCM and get back a registration token.

Register with GCM

Before the app can receive remote notifications from the app server, it must register with GCM and get back a registration token. The work of registering our application with GCM is handled by an `IntentService` that we create. Our `IntentService` performs the following steps:

1. Uses the InstanceID API to generate security tokens that authorize our client app to access the app server. In return, we get back a registration token from GCM.
2. Forwards the registration token to the app server (if the app server requires it).
3. Subscribes to one or more notification topic channels.

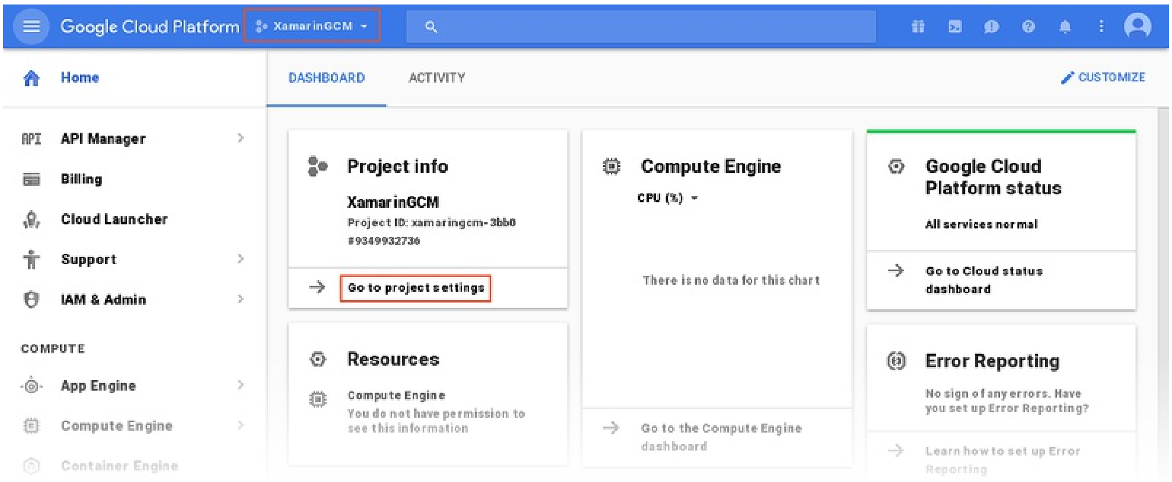
After we implement this `IntentService`, we'll test it to see if we get back a registration token from GCM.

Add a new file called **RegistrationIntentService.cs** and replace the template code with the following:

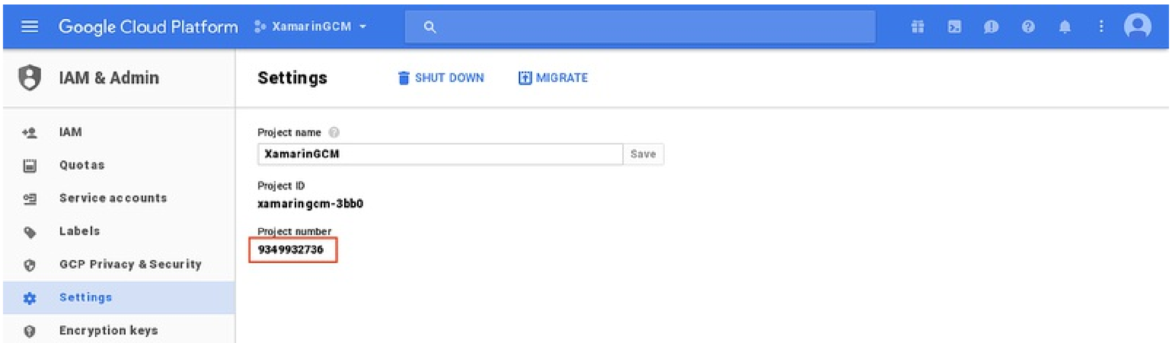
```
using System;
using Android.App;
using Android.Content;
using Android.Util;
using Android.Gms.Gcm;
using Android.Gms.Gcm.Iid;

namespace ClientApp
{
    [Service(Exported = false)]
    class RegistrationIntentService : IntentService
    {
        static object locker = new object();

        public RegistrationIntentService() : base("RegistrationIntentService") { }

        protected override void OnHandleIntent (Intent intent)
        {
            try
            {
                Log.Info ("RegistrationIntentService", "Calling InstanceID.GetToken");
                lock (locker)
                {
                    var instanceID = InstanceID.GetInstance (this);
                    var token = instanceID.GetToken (
                        "YOUR_SENDER_ID", GoogleCloudMessaging.InstanceIdScope, null);

                    Log.Info ("RegistrationIntentService", "GCM Registration Token: " + token);
                    SendRegistrationToAppServer (token);
                    Subscribe (token);
                }
            }
            catch (Exception e)
            {
                Log.Debug("RegistrationIntentService", "Failed to get a registration token");
                return;
            }
        }

        void SendRegistrationToAppServer (string token)
        {
            // Add custom implementation here as needed.
        }

        void Subscribe (string token)
        {
            var pubSub = GcmPubSub.GetInstance(this);
            pubSub.Subscribe(token, "/topics/global", null);
        }
    }
}
```

In the above sample code, change *YOUR_SENDER_ID* to the Sender ID number for your client app project. To get the Sender ID for your project:

1. Log into the Google Cloud Console and select your project name from the pull down menu. In the **Project info** pane that is displayed for your project, click **Go to project settings**:



2. On the **Settings** page, locate the **Project number** – this is the Sender ID for your project:



We want to start our `RegistrationIntentService` when our app starts running. Edit **MainActivity.cs** and modify the `OnCreate` method so that our `RegistrationIntentService` is started after we check for the presence of Google Play Services:

```
protected override void OnCreate (Bundle bundle)
{
    base.OnCreate (bundle);

    SetContentView(Resource.Layout.Main);
    msgText = FindViewById<TextView> (Resource.Id.msgText);

    if (IsPlayServicesAvailable ())
    {
        var intent = new Intent (this, typeof (RegistrationIntentService));
        StartService (intent);
    }
}
```

Now let's take a look at each section of `RegistrationIntentService` to understand how it works.

First, we annotate our `RegistrationIntentService` with the following attribute to indicate that our service is not to be instantiated by the system:

```
[Service (Exported = false)]
```

The `RegistrationIntentService` constructor names the worker thread *RegistrationIntentService* to make debugging easier.

```
public RegistrationIntentService() : base ("RegistrationIntentService") { }
```

The core functionality of `RegistrationIntentService` resides in the `OnHandleIntent` method. Let's walk through this code to see how it registers our app with GCM.

Request a Registration Token

`OnHandleIntent` first calls Google's InstanceID.GetToken method to request a registration token from GCM. We wrap this code in a `lock` to guard against the possibility of multiple registration intents occurring simultaneously – the `lock` ensures that these intents are processed sequentially. If we fail to get a registration token, an exception is thrown and we log an error. If the registration succeeds, `token` is set to the registration token we got back from GCM:

```
static object locker = new object ();
...
try
{
    lock (locker)
    {
        var instanceID = InstanceID.GetInstance (this);
        var token = instanceID.GetToken (
            "YOUR_SENDER_ID", GoogleCloudMessaging.InstanceIdScope, null);
        ...
    }
}
catch (Exception e)
{
    Log.Debug ...
```

Forward the Registration Token to the App Server

If we get a registration token (that is, no exception was thrown), we call `SendRegistrationToAppServer` to associate the user's registration token with the server-side account (if any) that is maintained by our application. Because this implementation depends on the design of the app server, an empty method is provided here:

```
void SendRegistrationToAppServer (string token)
{
    // Add custom implementation here as needed.
}
```

In some cases, the app server does not need the user's registration token; in that case, this method can be omitted. When a registration token is sent to the app server, `SendRegistrationToAppServer` should maintain a boolean to indicate whether the token has been sent to the server. If this boolean is false, `SendRegistrationToAppServer` sends the token to the app server – otherwise, the token was already sent to the app server in a previous call.

Subscribe to the Notification Topic

Next, we call our `Subscribe` method to indicate to GCM that we want to subscribe to a notification topic. In `Subscribe`, we call the GcmPubSub.Subscribe API to subscribe our client app to all messages under `/topics/global`:

```
void Subscribe (string token)
{
    var pubSub = GcmPubSub.GetInstance(this);
    pubSub.Subscribe(token, "/topics/global", null);
}
```

The app server must send notification messages to `/topics/global` if we are to receive them. Note that the topic name under `/topics` can be anything you want, as long as the app server and the client app both agree on these names. (Here, we chose the name `global` to indicate that we want to receive messages on all topics supported by the app server.)

Implement an Instance ID Listener Service

Registration tokens are unique and secure; however, the client app (or GCM) may need to refresh the registration token in the event of app reinstallation or a security issue. For this reason, we must implement an `InstanceIdListenerService` that responds to token refresh requests from GCM.

Add a new file called **InstanceIdListenerService.cs** and replace the template code with the following:

```
using Android.App;
using Android.Content;
using Android.Gms.Gcm.Iid;

namespace ClientApp
{
    [Service(Exported = false), IntentFilter(new[] { "com.google.android.gms.iid.InstanceID" })]
    class MyInstanceIDListenerService : InstanceIDListenerService
    {
        public override void OnTokenRefresh()
        {
            var intent = new Intent (this, typeof (RegistrationIntentService));
            StartService (intent);
        }
    }
}
```

Annotate `InstanceIdListenerService` with the following attribute to indicate that the service is not to be instantiated by the system and that it can receive GCM registration token (also called *instance ID*) refresh requests:

```
[Service(Exported = false), IntentFilter(new[] { "com.google.android.gms.iid.InstanceID" })]
```

The `OnTokenRefresh` method in our service starts the `RegistrationIntentService` so that it can intercept the new registration token.

Test Registration with GCM

Let's completely rebuild and run the app. If you successfully receive a registration token from GCM, the registration token should be displayed in the output window. For example:

```
D/Mono    ( 1934): Assembly Ref addref ClientApp[0xb4ac2400] -> Xamarin.GooglePlayServices.Gcm[0xb4ac2640]: 2
I/RegistrationIntentService( 1934): Calling InstanceID.GetToken
I/RegistrationIntentService( 1934): GCM Registration Token: f8LdveCvXig:APA91bFIsjUAbP-V8TPQdLR89qQbEJh1SYG38AcCbBUf34z5gSdUc5OsXrgs93YFiGcRSRafPfzkz23lf3-LvYV1CwrFheMjHgwPeFSh12MywnRIhz
```

Handle Downstream Messages

The code we have implemented thus far is only "set-up" code; it checks to see if Google Play Services is installed and negotiates with GCM and the app server to prepare our client app for receiving remote notifications. However, we have yet to implement code that actually receives and processes downstream notification messages. To do this, we must implement a *GCM Listener Service*. This service receives topic messages from the app server and locally broadcasts them as notifications. After we implement this service, we'll create a test program to send messages to GCM so that we can see if our implementation works correctly.

Add a Notification Icon

Let's first add a small icon that will appear in the notification area when our notification is launched. You can copy this icon to your project or create your own custom icon. We'll name the icon file

ic_stat_button_click.png and copy it to the **Resources/drawable** folder. Remember to use **Add > Existing Item ...** to include this icon file in your project.

Implement a GCM Listener Service

Add a new file called **GcmListenerService.cs** and replace the template code with the following:

```
using Android.App;
using Android.Content;
using Android.OS;
using Android.Gms.Gcm;
using Android.Util;

namespace ClientApp
{
    [Service (Exported = false), IntentFilter (new [] { "com.google.android.c2dm.intent.RECEIVE" })]
    public class MyGcmListenerService : GcmListenerService
    {
        public override void OnMessageReceived (string from, Bundle data)
        {
            var message = data.GetString ("message");
            Log.Debug ("MyGcmListenerService", "From:    " + from);
            Log.Debug ("MyGcmListenerService", "Message: " + message);
            SendNotification (message);
        }

        void SendNotification (string message)
        {
            var intent = new Intent (this, typeof(MainActivity));
            intent.AddFlags (ActivityFlags.ClearTop);
            var pendingIntent = PendingIntent.GetActivity (this, 0, intent, PendingIntentFlags.OneShot);

            var notificationBuilder = new Notification.Builder(this)
                .SetSmallIcon (Resource.Drawable.ic_stat_ic_notification)
                .SetContentTitle ("GCM Message")
                .SetContentText (message)
                .SetAutoCancel (true)
                .SetContentIntent (pendingIntent);

            var notificationManager = (NotificationManager)GetSystemService(Context.NotificationService);
            notificationManager.Notify (0, notificationBuilder.Build());
        }
    }
}
```

Let's take a look at each section of our `GcmListenerService` to understand how it works.

First, we annotate `GcmListenerService` with an attribute to indicate that this service is not to be instantiated by the system, and we include an intent filter to indicate that it receives GCM messages:

```
[Service (Exported = false), IntentFilter (new [] { "com.google.android.c2dm.intent.RECEIVE" })]
```

When `GcmListenerService` receives a message from GCM, the `OnMessageReceived` method is invoked. This method extracts the message content from the passed-in `Bundle`, logs the message content (so we can view it in the output window), and calls `SendNotification` to launch a local notification with the received message content:

```
var message = data.GetString ("message");
Log.Debug ("MyGcmListenerService", "From:    " + from);
Log.Debug ("MyGcmListenerService", "Message: " + message);
SendNotification (message);
```

The `SendNotification` method uses `Notification.Builder` to create the notification, and then it uses the `NotificationManager` to launch the notification. Effectively, this converts the remote notification message into a local notification to be presented to the user. For more information about using `Notification.Builder` and `NotificationManager`, see Local Notifications.

Declare the Receiver in the Manifest

Before we can receive messages from GCM, we must declare the GCM listener in the Android manifest. Let's edit **AndroidManifest.xml** and replace the `<application>` section with the following XML:

```
<application android:label="RemoteNotifications" android:icon="@drawable/Icon">
    <receiver android:name="com.google.android.gms.gcm.GcmReceiver"
              android:exported="true"
              android:permission="com.google.android.c2dm.permission.SEND">
        <intent-filter>
            <action android:name="com.google.android.c2dm.intent.RECEIVE" />
            <action android:name="com.google.android.c2dm.intent.REGISTRATION" />
            <category android:name="YOUR_PACKAGE_NAME" />
        </intent-filter>
    </receiver>
</application>
```

In the above XML, change *YOUR_PACKAGE_NAME* to the package name for your client app project. In our walkthrough example, the package name is `com.xamarin.gcmexample`.

Let's look at what each setting in this XML does:

SETTING	DESCRIPTION
`com.google.android.gms.gcm.GcmReceiver`	Declares that our app implements a GCM receiver that captures and processes incoming push notification messages.
`com.google.android.c2dm.permission.SEND`	Declares that only GCM servers can send messages directly to the app.
`com.google.android.c2dm.intent.RECEIVE`	Intent filter advertising that our app handles broadcast messages from GCM.
`com.google.android.c2dm.intent.REGISTRATION`	Intent filter advertising that our app handles new registration intents (that is, we have implemented an Instance ID Listener Service).

Alternatively, you can decorate `GcmListenerService` with these attributes rather than specifying them in XML; here we specify them in **AndroidManifest.xml** so that the code samples are easier to follow.

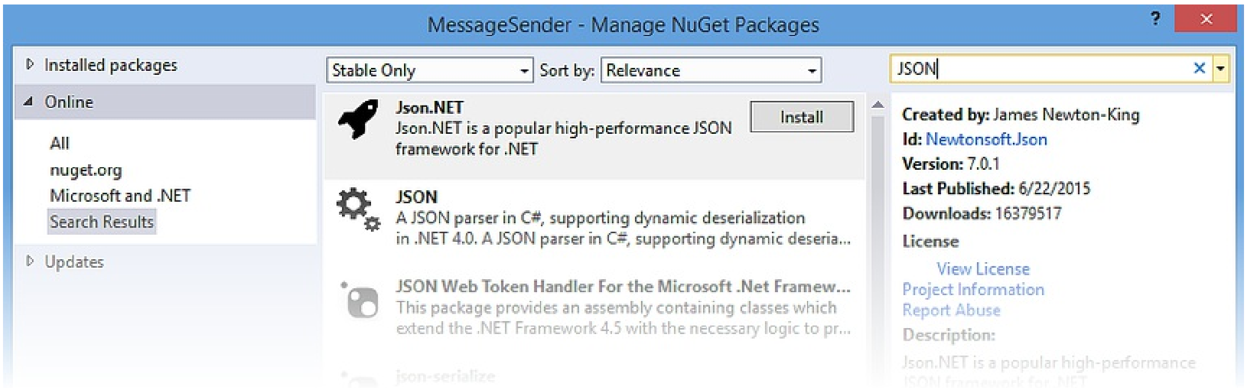
Create a Message Sender to Test the App

Let's add a C# desktop console application project to the Solution and call it **MessageSender**. We'll use this console application to simulate an application server – it will send notification messages to **ClientApp** via GCM.

Add the Json.NET Package

In this console app, we're building a JSON payload that contains the notification message we want to send to the client app. We'll use the **Json.NET** package in **MessageSender** to make it easier to build the JSON object required by GCM. In Visual Studio, right-click **References > Manage NuGet Packages ...**; in Visual Studio for Mac, right-click **Packages > Add Packages...**.

Let's search for the **Json.NET** package and install it in the project:



Add a Reference to System.Net.Http

We'll also need to add a reference to `System.Net.Http` so that we can instantiate an `HttpClient` for sending our test message to GCM. In the **MessageSender** project, Right-click **References > Add Reference** and scroll down until you see **System.Net.Http**. Put a check mark next to **System.Net.Http** and click **OK**.

Implement Code that Sends a Test Message

In **MessageSender**, edit **Program.cs** and replace the contents with the following code:

```
using System;
using System.Net.Http;
using System.Net.Http.Headers;
using System.Text;
using System.Threading.Tasks;
using Newtonsoft.Json.Linq;

namespace MessageSender
{
    class MessageSender
    {
        public const string API_KEY = "YOUR_API_KEY";
        public const string MESSAGE = "Hello, Xamarin!";

        static void Main (string[] args)
        {
            var jGcmData = new JObject();
            var jData = new JObject();

            jData.Add ("message", MESSAGE);
            jGcmData.Add ("to", "/topics/global");
            jGcmData.Add ("data", jData);

            var url = new Uri ("https://gcm-http.googleapis.com/gcm/send");
            try
            {
                using (var client = new HttpClient())
                {
                    client.DefaultRequestHeaders.Accept.Add(
                        new MediaTypeWithQualityHeaderValue("application/json"));

                    client.DefaultRequestHeaders.TryAddWithoutValidation (
                        "Authorization", "key=" + API_KEY);

                    Task.WaitAll(client.PostAsync (url,
                        new StringContent(jGcmData.ToString(), Encoding.Default, "application/json"))
                            .ContinueWith(response =>
                            {
                                Console.WriteLine(response);
                                Console.WriteLine("Message sent: check the client device notification
tray.");
                            }));
                }
            }
            catch (Exception e)
            {
                Console.WriteLine("Unable to send GCM message:");
                Console.Error.WriteLine(e.StackTrace);
            }
        }
    }
}
```

In the above code, change *YOUR_API_KEY* to the API Key for your client app project.

This test app server sends the following JSON-formatted message to GCM:

```
{
  "to": "/topics/global",
  "data": {
    "message": "Hello, Xamarin!"
  }
}
```

GCM, in turn, forwards this message to your client app. Let's build **MessageSender** and open a console window where we can run it from the command line.

Try It!

Now we're ready to test our client app. If you're using an emulator or if your device is communicating with GCM over Wi-Fi, you must open the following TCP ports on your firewall for GCM messages to get through: 5228, 5229, and 5230.

Start your client app and watch the output window. After the `RegistrationIntentService` successfully receives a registration token from GCM, the output window should display the token with log output resembling the following:

```
I/RegistrationIntentService(16103): GCM Registration Token: eX9ggabZV1Q:APA91bHjBnQXMUeBOT6JDiLpRt8m2YWtY
...
```

At this point the client app is ready to receive a remote notification message. From the command line, run the **MessageSender.exe** program to send a "Hello, Xamarin" notification message to the client app. If you have not yet built the **MessageSender** project, do so now.

To run **MessageSender.exe** under Visual Studio, open a command prompt, change to the **MessageSender/bin/Debug** directory, and run the command directly:

```
MessageSender.exe
```

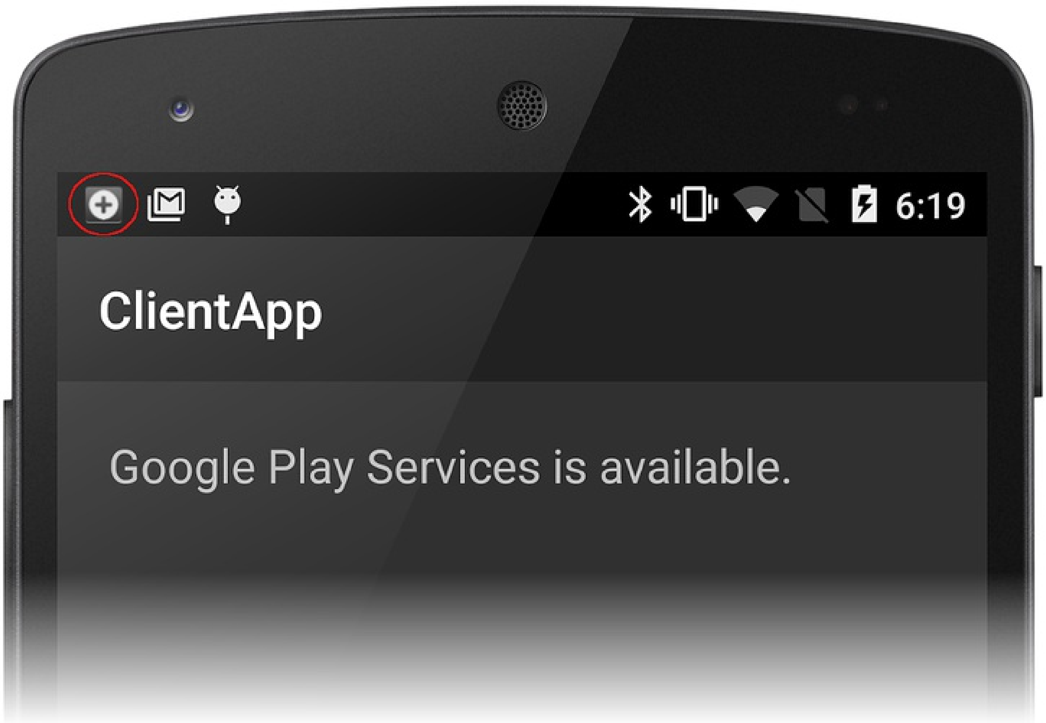
To run **MessageSender.exe** under Visual Studio for Mac, open a Terminal session, change to **MessageSender/bin/Debug** the directory, and use mono to run **MessageSender.exe**

```
mono MessageSender.exe
```

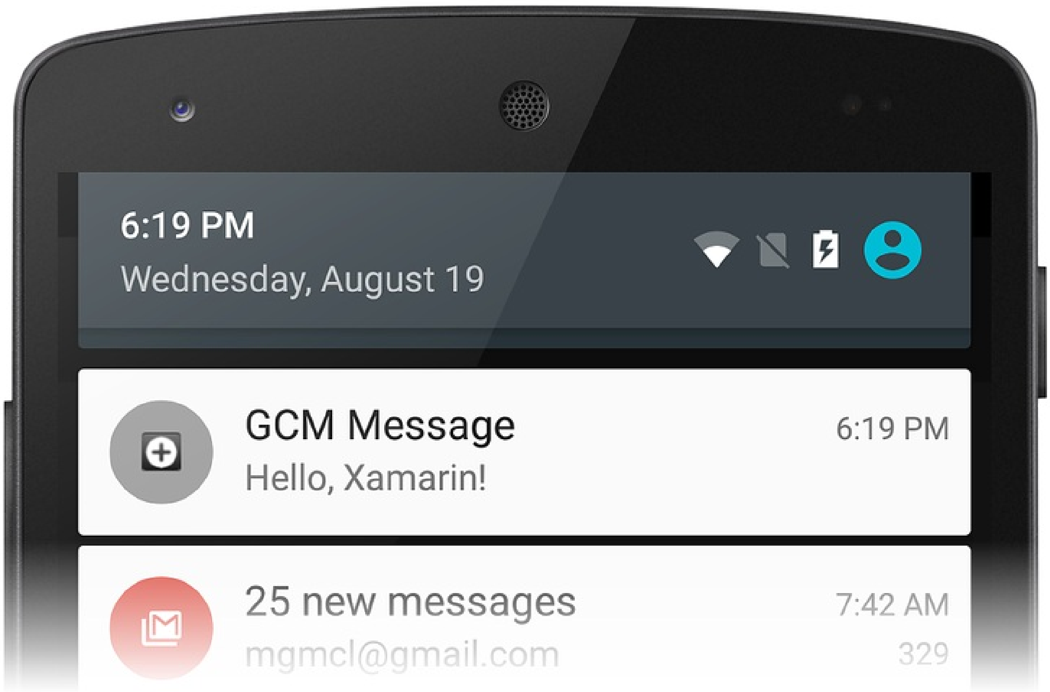
It may take up to a minute for the message to propagate through GCM and back down to your client app. If the message is received successfully, we should see output resembling the following in the output window:

```
D/MyGcmListenerService(16103): From:    /topics/global
D/MyGcmListenerService(16103): Message: Hello, Xamarin!
```

In addition, you should notice that a new notification icon has appeared in the notification tray:



When you open the notification tray to view notifications, you should see our remote notification:



Congratulations, your app has received its first remote notification!

Note that GCM messages will no longer be received if the app is force-stopped. To resume notifications after a force-stop, the app must be manually restarted. For more information about this Android policy, see Launch controls on stopped applications and this stack overflow post.

Summary

This walkthrough detailed the steps for implementing remote notifications in a Xamarin.Android application. It described how to install additional packages needed for GCM communications, and it explained how to configure app permissions for access to GCM servers. It provided example code that illustrates how to check for the presence of Google Play Services, how to implement a registration intent service and instance ID listener service that negotiates with GCM for a registration token, and how to implement a GCM listener service that receives and processes remote notification messages. Finally, we implemented a command-line test program to send test notifications to our client app through GCM.

Related Links

- Google Cloud Messaging

Introduction to Web Services

7/12/2021 • 15 minutes to read • Edit Online

This guide demonstrates how to consume different web service technologies. Topics covered include communicating with REST services, SOAP services, and Windows Communication Foundation services.

To function correctly, many mobile applications are dependent on the cloud, and so integrating web services into mobile applications is a common scenario. The Xamarin platform supports consuming different web service technologies, and includes in-built and third-party support for consuming RESTful, ASMX, and Windows Communication Foundation (WCF) services.

For customers using Xamarin.Forms, there are complete examples using each of these technologies in the Xamarin.Forms Web Services documentation.

> **IMPORTANT**
> In iOS 9, App Transport Security (ATS) enforces secure connections between internet resources (such as the app's back-end server) and the app, thereby preventing accidental disclosure of sensitive information. Since ATS is enabled by default in apps built for iOS 9, all connections will be subject to ATS security requirements. If connections do not meet these requirements, they will fail with an exception.

You can opt-out of ATS if it is not possible to use the `HTTPS` protocol and secure communication for internet resources. This can be achieved by updating the app's **Info.plist** file. For more information see App Transport Security.

REST

Representational State Transfer (REST) is an architectural style for building web services. REST requests are made over HTTP using the same HTTP verbs that web browsers use to retrieve web pages and to send data to servers. The verbs are:

- **GET** – this operation is used to retrieve data from the web service.
- **POST** – this operation is used to create a new item of data on the web service.
- **PUT** – this operation is used to update an item of data on the web service.
- **PATCH** – this operation is used to update an item of data on the web service by describing a set of instructions about how the item should be modified. This verb is not used in the sample application.
- **DELETE** – this operation is used to delete an item of data on the web service.

Web service APIs that adhere to REST are called RESTful APIs, and are defined using:

- A base URI.
- HTTP methods, such as GET, POST, PUT, PATCH, or DELETE.
- A media type for the data, such as JavaScript Object Notation (JSON).

The simplicity of REST has helped make it the primary method for accessing web services in mobile applications.

Consuming REST Services

There are a number of libraries and classes that can be used to consume REST services, and the following subsections discuss them. For more information about consuming a REST service, see Consume a RESTful Web

Service.

HttpClient

The Microsoft HTTP Client Libraries provides the `HttpClient` class, which is used to send and receive requests over HTTP. It provides functionality for sending HTTP requests and receiving HTTP responses from a URI-identified resource. Each request is sent as an asynchronous operation. For more information about asynchronous operations, see Async Support Overview.

The `HttpResponseMessage` class represents an HTTP response message received from the web service after an HTTP request has been made. It contains information about the response, including the status code, headers, and body. The `HttpContent` class represents the HTTP body and content headers, such as `Content-Type` and `Content-Encoding`. The content can be read using any of the `ReadAs` methods, such as `ReadAsStringAsync` and `ReadAsByteArrayAsync`, depending upon the format of the data.

For more information about the `HttpClient` class, see Creating the HTTPClient Object.

HTTPWebRequest

Calling web services with `HTTPWebRequest` involves:

- Creating the request instance for a particular URI.
- Setting various HTTP properties on the request instance.
- Retrieving an `HttpWebResponse` from the request.
- Reading data out of the response.

For example, the following code retrieves data from the U.S. National Library of Medicine web service:

```
var rxcui = "198440";
var request =
HttpWebRequest.Create(string.Format(@"https://rxnav.nlm.nih.gov/REST/RxTerms/rxcui/{0}/allinfo", rxcui));
request.ContentType = "application/json";
request.Method = "GET";

using (HttpWebResponse response = request.GetResponse() as HttpWebResponse)
{
  if (response.StatusCode != HttpStatusCode.OK)
     Console.Out.WriteLine("Error fetching data. Server returned status code: {0}", response.StatusCode);
  using (StreamReader reader = new StreamReader(response.GetResponseStream()))
  {
               var content = reader.ReadToEnd();
               if(string.IsNullOrWhiteSpace(content)) {
                       Console.Out.WriteLine("Response contained empty body...");
               }
               else {
                       Console.Out.WriteLine("Response Body: \r\n {0}", content);
               }

               Assert.NotNull(content);
  }
}
```

The above example creates an `HttpWebRequest` that will return data formatted as JSON. The data is returned in an `HttpWebResponse`, from which a `StreamReader` can be obtained to read the data.

RestSharp

Another approach to consuming REST services is using the RestSharp library. RestSharp encapsulates HTTP requests, including support for retrieving results either as raw string content or as a deserialized C# object. For example, the following code makes a request to the U.S. National Library of Medicine web service, and retrieves the results as a JSON formatted string:

```
var request = new RestRequest(string.Format("{0}/allinfo", rxcui));
request.RequestFormat = DataFormat.Json;
var response = Client.Execute(request);
if(string.IsNullOrWhiteSpace(response.Content) || response.StatusCode != System.Net.HttpStatusCode.OK) {
       return null;
}
rxTerm = DeserializeRxTerm(response.Content);
```

`DeserializeRxTerm` is a method that will take the raw JSON string from the `RestSharp.RestResponse.Content` property and convert it into a C# object. Deserializing data returned from web services is discussed later in this article.

NSUrlConnection

In addition to classes available in the Mono base class library (BCL), such as `HttpWebRequest`, and third party C# libraries, such as RestSharp, platform-specific classes are also available for consuming web services. For example, in iOS, the `NSUrlConnection` and `NSMutableUrlRequest` classes can be used.

The following code example shows how to call the U.S. National Library of Medicine web service using iOS classes:

```
var rxcui = "198440";
var request = new NSMutableUrlRequest(new
NSUrl(string.Format("https://rxnav.nlm.nih.gov/REST/RxTerms/rxcui/{0}/allinfo", rxcui)),
       NSUrlRequestCachePolicy.ReloadRevalidatingCacheData, 20);
request["Accept"] = "application/json";

var connectionDelegate = new RxTermNSURLConnectionDelegate();
var connection = new NSUrlConnection(request, connectionDelegate);
connection.Start();

public class RxTermNSURLConnectionDelegate : NSUrlConnectionDelegate
{
       StringBuilder _ResponseBuilder;
       public bool IsFinishedLoading { get; set; }
       public string ResponseContent { get; set; }

       public RxTermNSURLConnectionDelegate()
               : base()
       {
               _ResponseBuilder = new StringBuilder();
       }

       public override void ReceivedData(NSUrlConnection connection, NSData data)
       {
               if(data != null) {
                       _ResponseBuilder.Append(data.ToString());
               }
       }
       public override void FinishedLoading(NSUrlConnection connection)
       {
               IsFinishedLoading = true;
               ResponseContent = _ResponseBuilder.ToString();
       }
}
```

Generally, platform-specific classes for consuming web services should be limited to scenarios where native code is being ported to C#. Where possible, web service access code should be portable so that it can be shared cross-platform.

ServiceStack

Another option for calling web services is the Service Stack library. For example, the following code shows how to use Service Stack's `IServiceClient.GetAsync` method to issue a service request:

```
client.GetAsync<CustomersResponse>("",
          (response) => {
               foreach(var c in response.Customers) {
                       Console.WriteLine(c.CompanyName);
               }
        },
        (response, ex) => {
                Console.WriteLine(ex.Message);
        });
```

IMPORTANT

While tools like ServiceStack and RestSharp make it easy to call and consume REST services, it is sometimes non-trivial to consume XML or JSON that does not conform to the standard *DataContract* serialization conventions. If necessary, invoke the request and handle the appropriate serialization explicitly using the ServiceStack.Text library discussed below.

Consuming RESTful Data

RESTful web services typically use JSON messages to return data to the client. JSON is a text-based, data-interchange format that produces compact payloads, which results in reduced bandwidth requirements when sending data. In this section, mechanisms for consuming RESTful responses in JSON and Plain-Old-XML (POX) will be examined.

System.JSON

The Xamarin platform ships with support for JSON out of the box. By using a `JsonObject`, results can be retrieved as shown in the following code example:

```
var obj = JsonObject.Parse(json);
var properties = obj["rxtermsProperties"];
term.BrandName = properties["brandName"];
term.DisplayName = properties["displayName"];
term.Synonym = properties["synonym"];
term.FullName = properties["fullName"];
term.FullGenericName = properties["fullGenericName"];
term.Strength = properties["strength"];
```

However, it's important to be aware that the `System.Json` tools load the entirety of the data into memory.

JSON.NET

The NewtonSoft JSON.NET library is a widely used library for serializing and deserializing JSON messages. The following code example shows how to use JSON.NET to deserialize a JSON message into a C# object:

```
var term = new RxTerm();
var properties = JObject.Parse(json)["rxtermsProperties"];
term.BrandName = properties["brandName"].Value<string>();
term.DisplayName = properties["displayName"].Value<string>();
term.Synonym = properties["synonym"].Value<string>();;
term.FullName = properties["fullName"].Value<string>();;
term.FullGenericName = properties["fullGenericName"].Value<string>();;
term.Strength = properties["strength"].Value<string>();
term.RxCUI = properties["rxcui"].Value<string>();
```

ServiceStack.Text

ServiceStack.Text is a JSON serialization library designed to work with the ServiceStack library. The following code example shows how to parse JSON using a `ServiceStack.Text.JsonObject`:

```
var result = JsonObject.Parse(json).Object("rxtermsProperties")
       .ConvertTo(x => new RxTerm {
               BrandName = x.Get("brandName"),
               DisplayName = x.Get("displayName"),
               Synonym = x.Get("synonym"),
               FullName = x.Get("fullName"),
               FullGenericName = x.Get("fullGenericName"),
               Strength = x.Get("strength"),
               RxTermDoseForm = x.Get("rxtermsDoseForm"),
               Route = x.Get("route"),
               RxCUI = x.Get("rxcui"),
               RxNormDoseForm = x.Get("rxnormDoseForm"),
       });
```

System.Xml.Linq

In the event of consuming an XML-based REST web service, LINQ to XML can be used to parse the XML and populate a C# object inline, as demonstrated in the following code example:

```
var doc = XDocument.Parse(xml);
var result = doc.Root.Descendants("rxtermsProperties")
.Select(x=> new RxTerm()
       {
               BrandName = x.Element("brandName").Value,
               DisplayName = x.Element("displayName").Value,
               Synonym = x.Element("synonym").Value,
               FullName = x.Element("fullName").Value,
               FullGenericName = x.Element("fullGenericName").Value,
               //bind more here...
               RxCUI = x.Element("rxcui").Value,
       });
```

ASP.NET Web Service (ASMX)

ASMX provides the ability to build web services that send messages using the Simple Object Access Protocol (SOAP). SOAP is a platform-independent and language-independent protocol for building and accessing web services. Consumers of an ASMX service do not need to know anything about the platform, object model, or programming language used to implement the service. They only need to understand how to send and receive SOAP messages.

A SOAP message is an XML document containing the following elements:

- A root element named *Envelope* that identifies the XML document as a SOAP message.

- An optional *Header* element that contains application-specific information such as authentication data. If the *Header* element is present it must be the first child element of the *Envelope* element.
- A required *Body* element that contains the SOAP message intended for the recipient.
- An optional *Fault* element that's used to indicate error messages. If the *Fault* element is present, it must be a child element of the *Body* element.

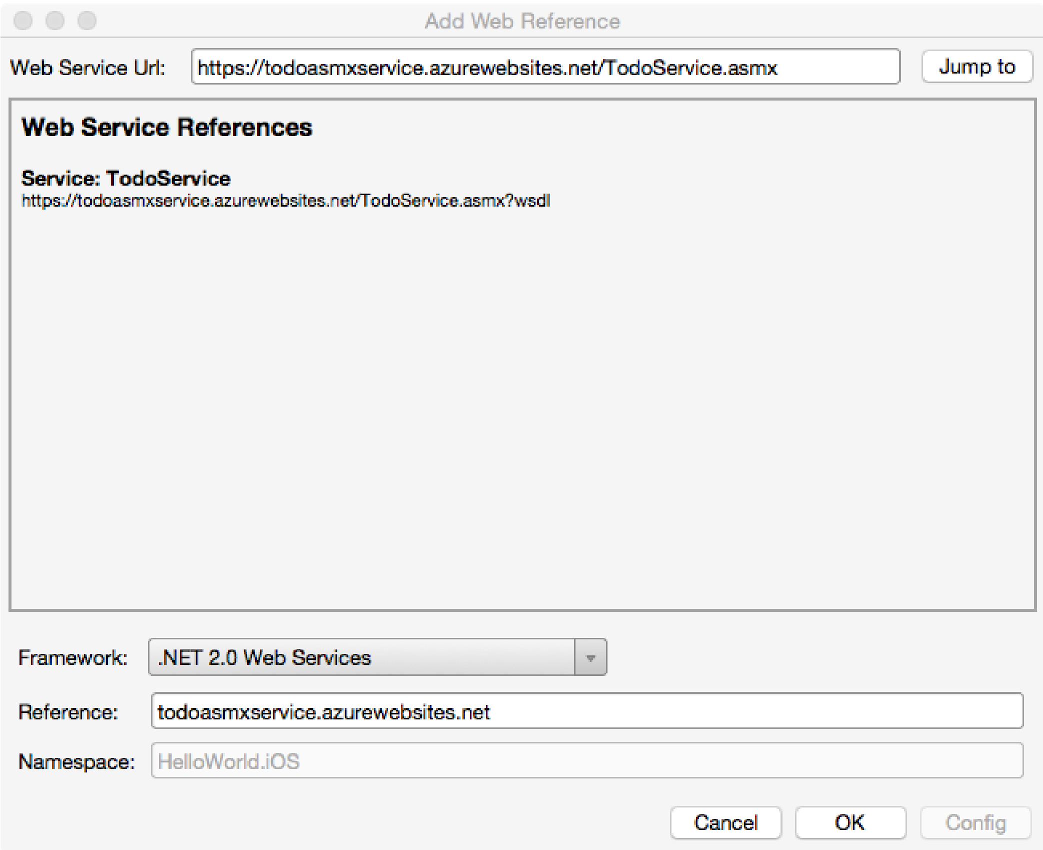
SOAP can operate over many transport protocols, including HTTP, SMTP, TCP, and UDP. However, an ASMX service can only operate over HTTP. The Xamarin platform supports standard SOAP 1.1 implementations over HTTP, and this includes support for many of the standard ASMX service configurations.

Generating a Proxy

A *proxy* must be generated to consume an ASMX service, which allows the application to connect to the service. The proxy is constructed by consuming service metadata that defines the methods and associated service configuration. This metadata is exposed as a Web Services Description Language (WSDL) document that is generated by the web service. The proxy is built by using Visual Studio for Mac or Visual Studio to add a web reference for the web service to the platform-specific projects.

The web service URL can either be a hosted remote source or local file system resource accessible via the `file:///` path prefix, for example:

```
file:///Users/myUserName/projects/MyProjectName/service.wsdl
```



This generates the proxy in the Web or Service References folder of the project. Since a proxy is generated code, it should not be modified.

Manually Adding a Proxy to a Project

If you have an existing proxy that has been generated using compatible tools, this output can be consumed when included as part of your project. In Visual Studio for Mac, use the **Add files…** menu option to add the proxy. In addition, this requires *System.Web.Services.dll* to be referenced explicitly using the **Add References…** dialog.

Consuming the Proxy

The generated proxy classes provide methods for consuming the web service that use the Asynchronous Programming Model (APM) design pattern. In this pattern an asynchronous operation is implemented as two methods named *BeginOperationName* and *EndOperationName*, which begin and end the asynchronous operation.

The *BeginOperationName* method begins the asynchronous operation and returns an object that implements the `IAsyncResult` interface. After calling *BeginOperationName*, an application can continue executing instructions on the calling thread, while the asynchronous operation takes place on a thread pool thread.

For each call to *BeginOperationName*, the application should also call *EndOperationName* to get the results of the operation. The return value of *EndOperationName* is the same type returned by the synchronous web service method. The following code example shows an example of this:

```
public async Task<List<TodoItem>> RefreshDataAsync ()
{
  ...
  var todoItems = await Task.Factory.FromAsync<ASMXService.TodoItem[]> (
    todoService.BeginGetTodoItems,
    todoService.EndGetTodoItems,
    null,
    TaskCreationOptions.None);
  ...
}
```

The Task Parallel Library (TPL) can simplify the process of consuming an APM begin/end method pair by encapsulating the asynchronous operations in the same `Task` object. This encapsulation is provided by multiple overloads of the `Task.Factory.FromAsync` method. This method creates a `Task` that executes the `TodoService.EndGetTodoItems` method once the `TodoService.BeginGetTodoItems` method completes, with the `null` parameter indicating that no data is being passed into the `BeginGetTodoItems` delegate. Finally, the value of the `TaskCreationOptions` enumeration specifies that the default behavior for the creation and execution of tasks should be used.

For more information about APM, see Asynchronous Programming Model and TPL and Traditional .NET Framework Asynchronous Programming on MSDN.

For more information about consuming an ASMX service, see Consume an ASP.NET Web Service (ASMX).

Windows Communication Foundation (WCF)

WCF is Microsoft's unified framework for building service-oriented applications. It enables developers to build secure, reliable, transacted, and interoperable distributed applications.

WCF describes a service with a variety of different contracts which include the following:

- **Data contracts** – define the data structures that form the basis for the content within a message.
- **Message contracts** – compose messages from existing data contracts.
- **Fault contracts** – allow custom SOAP faults to be specified.
- **Service contracts** – specify the operations that services support and the messages required for interacting with each operation. They also specify any custom fault behavior that can be associated with operations on each service.

There are differences between ASP.NET Web Services (ASMX) and WCF, but it is important to understand that WCF supports the same capabilities that ASMX provides – SOAP messages over HTTP.

> **IMPORTANT**
> The Xamarin platform support for WCF is limited to text-encoded SOAP messages over HTTP/HTTPS using the `BasicHttpBinding` class. In addition, WCF support requires the use of tools only available in a Windows environment to generate the proxy.

Generating a Proxy

A *proxy* must be generated to consume a WCF service, which allows the application to connect to the service. The proxy is constructed by consuming service metadata that define the methods and associated service configuration. This metadata is exposed in the form of a Web Services Description Language (WSDL) document that is generated by the web service. The proxy can be built by using the Microsoft WCF Web Service Reference Provider in Visual Studio 2017 to add a service reference for the web service to a .NET Standard Library.

An alternative to creating the proxy using the Microsoft WCF Web Service Reference Provider in Visual Studio 2017 is to use the ServiceModel Metadata Utility Tool (svcutil.exe). For more information, see ServiceModel Metadata Utility Tool (Svcutil.exe).

Configuring the Proxy

Configuring the generated proxy will generally take two configuration arguments (depending on SOAP 1.1/ASMX or WCF) during initialization: the `EndpointAddress` and/or the associated binding information, as shown in the example below:

```
var binding = new BasicHttpBinding () {
       Name= "basicHttpBinding",
       MaxReceivedMessageSize = 67108864,
};

binding.ReaderQuotas = new System.Xml.XmlDictionaryReaderQuotas() {
       MaxArrayLength = 2147483646,
       MaxStringContentLength = 5242880,
};

var timeout = new TimeSpan(0,1,0);
binding.SendTimeout= timeout;
binding.OpenTimeout = timeout;
binding.ReceiveTimeout = timeout;

client = new Service1Client (binding, new EndpointAddress ("http://192.168.1.100/Service1.svc"));
```

A binding is used to specify the transport, encoding, and protocol details required for applications and services to communicate with each other. The `BasicHttpBinding` specifies that text-encoded SOAP messages will be sent over the HTTP transport protocol. Specifying an endpoint address enables the application to connect to different instances of the WCF service, provided that there are multiple published instances.

Consuming the Proxy

The generated proxy classes provide methods for consuming the web services that use the Asynchronous Programming Model (APM) design pattern. In this pattern, an asynchronous operation is implemented as two methods named *BeginOperationName* and *EndOperationName*, which begin and end the asynchronous operation.

The *BeginOperationName* method begins the asynchronous operation and returns an object that implements the `IAsyncResult` interface. After calling *BeginOperationName*, an application can continue executing instructions on the calling thread, while the asynchronous operation takes place on a thread pool thread.

For each call to *BeginOperationName*, the application should also call *EndOperationName* to get the results of the operation. The return value of *EndOperationName* is the same type returned by the synchronous web service method. The following code example shows an example of this:

```
public async Task<List<TodoItem>> RefreshDataAsync ()
{
  ...
  var todoItems = await Task.Factory.FromAsync <ObservableCollection<TodoWCFService.TodoItem>> (
    todoService.BeginGetTodoItems,
    todoService.EndGetTodoItems,
    null,
    TaskCreationOptions.None);
  ...
}
```

The Task Parallel Library (TPL) can simplify the process of consuming an APM begin/end method pair by encapsulating the asynchronous operations in the same `Task` object. This encapsulation is provided by multiple overloads of the `Task.Factory.FromAsync` method. This method creates a `Task` that executes the `TodoServiceClient.EndGetTodoItems` method once the `TodoServiceClient.BeginGetTodoItems` method completes, with the `null` parameter indicating that no data is being passed into the `BeginGetTodoItems` delegate. Finally, the value of the `TaskCreationOptions` enumeration specifies that the default behavior for the creation and execution of tasks should be used.

For more information about APM, see Asynchronous Programming Model and TPL and Traditional .NET Framework Asynchronous Programming on MSDN.

For more information about consuming a WCF service, see Consume a Windows Communication Foundation (WCF) Web Service.

Using Transport Security

WCF Services may employ transport level security to protect against interception of messages. The Xamarin platform supports bindings that employ transport level security using SSL. However, there may be cases in which the stack may need to validate the certificate, which results in unanticipated behavior. The validation can be overridden by registering a `ServerCertificateValidationCallback` delegate before invoking the service, as demonstrated in the following code example:

```
System.Net.ServicePointManager.ServerCertificateValidationCallback +=
(se, cert, chain, sslerror) => { return true; };
```

This maintains transport encryption while ignoring the server-side certificate validation. However, this approach effectively disregards the trust concerns associated with the certificate and may not be appropriate. For more information, see Using Trusted Roots Respectfully on mono-project.com.

Using Client Credential Security

WCF services may also require the service clients to authenticate using credentials. The Xamarin platform does not support the WS-Security Protocol, which allows clients to send credentials inside the SOAP message envelope. However, the Xamarin platform does support the ability to send HTTP Basic Authentication credentials to the server by specifying the appropriate `ClientCredentialType`:

```
basicHttpBinding.Security.Transport.ClientCredentialType = HttpClientCredentialType.Basic;
```

Then, basic authentication credentials can be specified:

```
client.ClientCredentials.UserName.UserName = @"foo";
client.ClientCredentials.UserName.Password = @"mrsnuggles";
```

For more information about HTTP basic authentication, although in the context of a REST web service, see Authenticating a RESTful Web Service.

Related Links

- Web Services in Xamarin.Forms
- ServiceModel Metadata Utility Tool (svcutil.exe)
- BasicHttpBinding

Deployment and Testing of Xamarin.Android Apps

11/2/2020 • 2 minutes to read • Edit Online

This section includes guides that explain how to test an application, optimize its performance, prepare it for release, sign it with a certificate, and publish it to an app store.

Application Package Sizes

This article examines the constituent parts of a Xamarin.Android application package and the associated strategies that can be used for efficient package deployment during debug and release stages of development.

Apply Changes

This guide covers the Apply Changes feature which lets you push resource changes to your running app without restarting your app.

Building Apps

This section describes how the build process works and explains how to build ABI-specific APKs.

Command Line Emulator

This article briefly touches starting the emulator via the command line.

Debugging

The guides in the section help you to debug your app using Android emulators, real Android devices, and the debug log.

Setting the Debuggable Attribute

This article explains how to set the debuggable attribute so that tools such as `adb` can communicate with the JVM.

Environment

This article describes the Xamarin.Android execution environment and the Android system properties that influence program execution.

GDB

This article explains how to use `gdb` for debugging a Xamarin.Android application.

Installing a System App

This guide explains how to install a Xamarin.Android app as a System Application on an Android device or as part of a custom ROM.

Linking on Android

This article discusses the linking process used by Xamarin.Android to reduce the final size of an application. It describes the various levels of linking that can be performed and provides some guidance and troubleshooting advice to mitigate errors that might result from using the linker.

Xamarin.Android Performance

There are many techniques for increasing the performance of applications built with Xamarin.Android. Collectively these techniques can greatly reduce the amount of work being performed by a CPU and the amount of memory consumed by an application.

Profiling Android Apps

This guide explains how to use profiler tools to examine the performance and memory usage of an Android app.

Preparing an Application for Release

After an application has been coded and tested, it is necessary to prepare a package for distribution. The first task in preparing this package is to build the application for release, which mainly entails setting some application attributes.

Signing the Android Application Package

Learn how to create an Android signing identity, create a new signing certificate for Android applications, and sign the application with the signing certificate. In addition, this topic explains how to export the app to disk for *ad-hoc* distribution. The resulting APK can be sideloaded into Android devices without going through an app store.

Publishing an Application

This series of articles explains the steps for public distribution of an application created with Xamarin.Android. Distribution can take place via channels such as e-mail, a private web server, Google Play, or the Amazon App Store for Android.

Application Package Size

7/8/2021 • 4 minutes to read • Edit Online

This article examines the constituent parts of a Xamarin.Android application package and the associated strategies that can be used for efficient package deployment during debug and release stages of development.

Overview

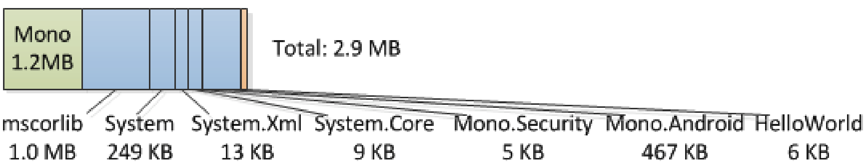
Xamarin.Android uses a variety of mechanisms to minimize package size while maintaining an efficient debug and release deploy process. In this article, we look at the Xamarin.Android release and debug deployment workflow and how the Xamarin.Android platform ensures that we build and release small application packages.

Release Packages

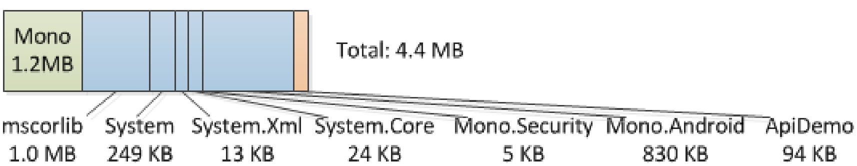
To ship a fully contained application, the package must include the application, the associated libraries, the content, the Mono runtime, and the required Base Class Library (BCL) assemblies. For example, if we take the default “Hello World” template, the contents of a complete package build would look like this:

15.8 MB is a larger download size than we’d like. The problem is the BCL libraries, as they include mscorlib, System, and Mono.Android, which provide a lot of the necessary components to run your application. However, they also provide functionality that you may not be using in your application, so it may be preferable to exclude these components.

When we build an application for distribution, we execute a process, known as Linking, that examines the application and removes any code that is not directly used. This process is similar to the functionality that Garbage Collection provides for heap-allocated memory. But instead of operating over objects, linking operates over your code. For example, there is a whole namespace in System.dll for sending and receiving email, but if your application does not make use of this functionality, that code is just wasting space. After running the linker on the Hello World application, our package now looks like this:



As we can see, this removes a significant amount of the BCL that was not being used. Note that the final BCL size is dependent on what the application actually uses. For example, if we take a look at a more substantial sample application called ApiDemo, we can see that the BCL component has increased in size because ApiDemo uses more of the BCL than Hello, World does:



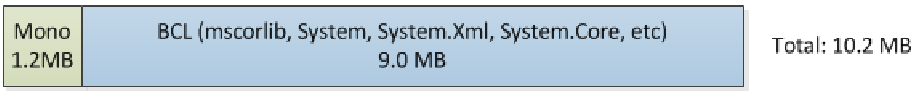
As illustrated here, your application package size will generally be about 2.9 MB larger than your application and its dependencies.

Debug Packages

Things are handled slightly differently for debug builds. When redeploying repeatedly to a device, an application needs to be as fast as possible, so we optimize debug packages for speed of deployment rather than size.

Android is relatively slow to copy and install a package, so we want the package size to be as small as possible. As we discussed above, one possible way to minimize package size is via the linker. However, linking is slow and we generally want to deploy only the parts of the application that have changed since the last deployment. To accomplish this, we separate our application from the core Xamarin.Android components.

The first time we debug on device, we copy two large packages called *Shared Runtime* and *Shared Platform*. Shared Runtime contains the Mono Runtime and BCL, while Shared Platform contains Android API level specific assemblies:



Copying these core components is only done once as it takes quite a bit of time, but allows any subsequent applications running in debug mode to utilize them. Finally, we copy the actual application, which is small and quick:

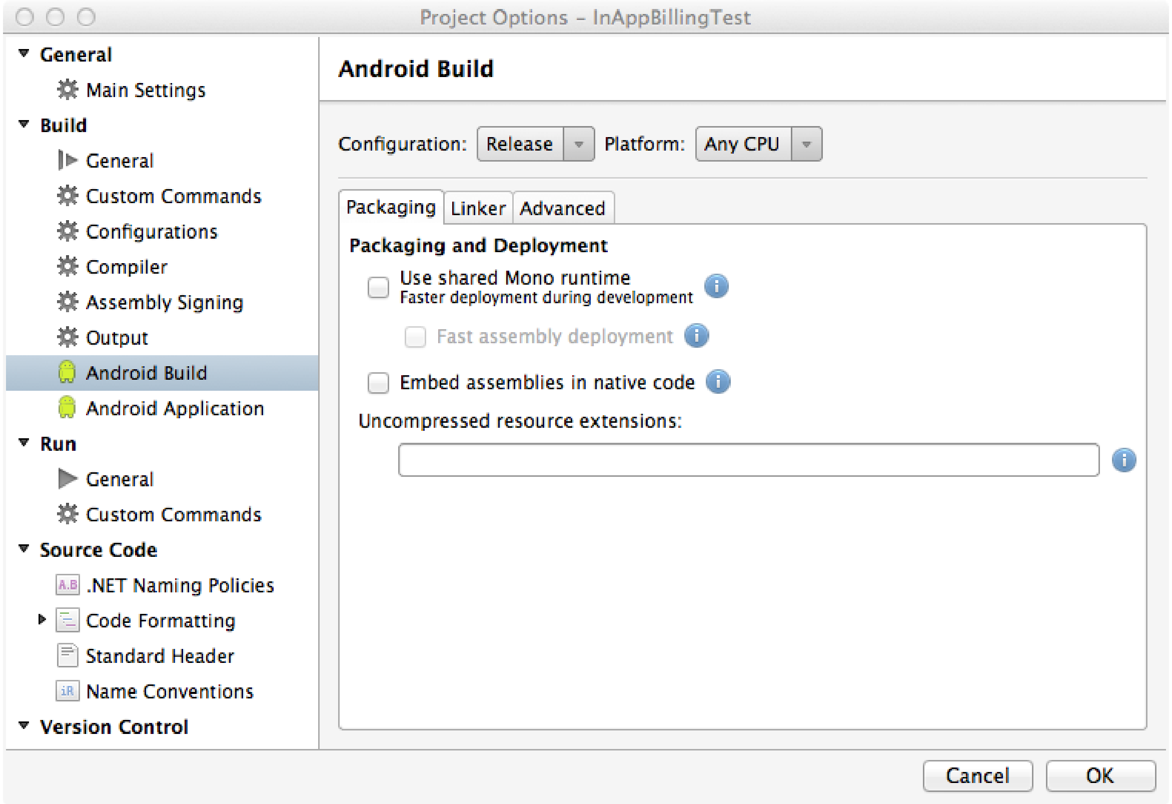
Fast Assembly Deployment

The *Fast Assembly Deployment* build option can be used to further decrease the size of the debug install package by not including the assemblies in the application's package, installing the assemblies directly on the device only once and only copying over files that have been modified since the last deployment.

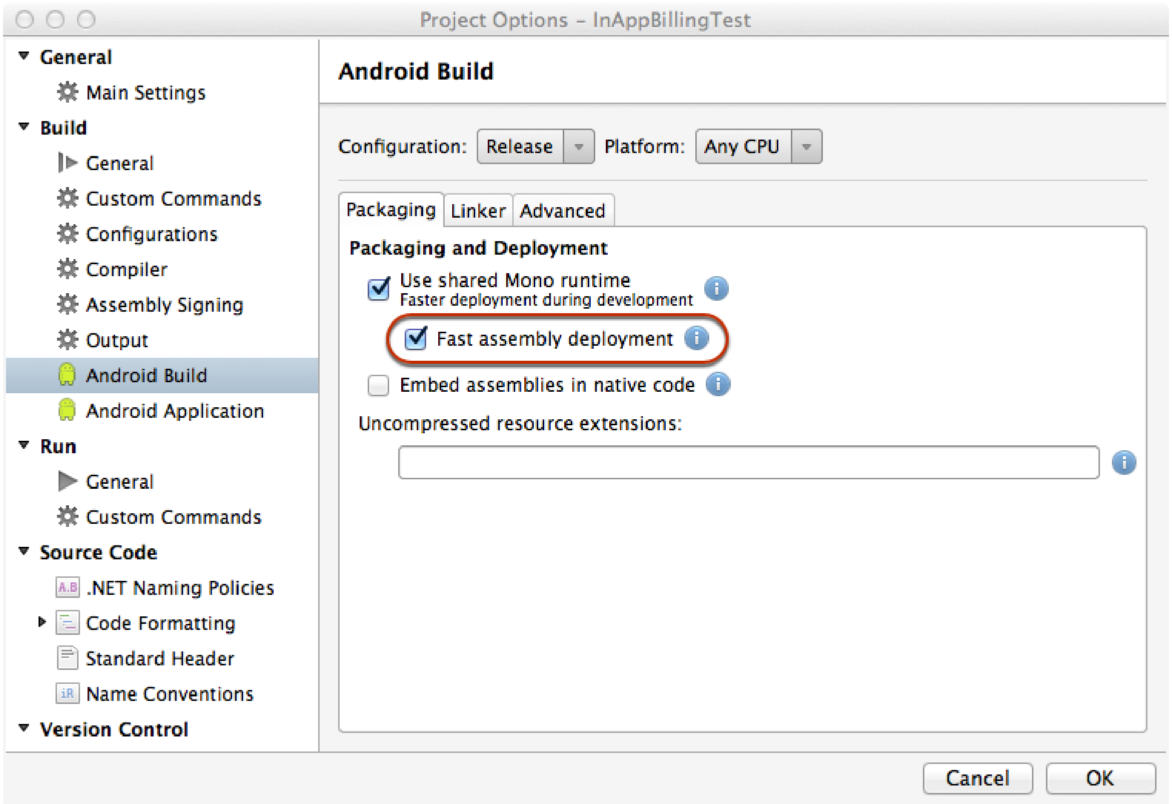
To enable *Fast Assembly Deployment*, do the following:

1. Right click on the Android Project in the Solution Explorer and select **Options**.

2. From the Project Options dialog select **Android Build** :



3. Check the **Use shared Mono runtime checkbox** and the **Fast assembly deployment** checkboxes:



4. Click the **OK** button to save the changes and close the Project Options dialog.

The next time the application is built for debug, the assemblies will be installed directly on the device (if they haven't already been) and a smaller application package (that does not include the assemblies) will be installed on the device. This will shorten the time it takes to get changes to the application up and running for testing.

By enduring the long first deploy of the shared runtime and shared platform, every time we make a change to the application, we can deploy the new version quickly and painlessly, so we can have a fast change/deploy/run cycle.

Summary

In this article we examined the facets of Xamarin.Android Release and Debug profile packaging. Additionally, we looked at the strategies that the Mono for Android platform uses to facilitate efficient package deployment during debug and release stages of development.

Apply Changes

7/8/2021 • 2 minutes to read • Edit Online

Apply Changes lets you push resource changes to your running app without restarting your app. This helps you control how much of your app is restarted when you want to deploy and test small, incremental changes while preserving your device or emulator's current state.

Apply Changes uses capabilities in the Android JVMTI implementation which is supported on devices or emulators running Android 8.0 (API level 26) or higher.

Requirements

The following list shows the requirements for using Apply Changes:

- **Visual Studio** - On Windows, update to Visual Studio 2019 version 16.5 or later. On macOS, update to Visual Studio 2019 for Mac version 8.5 or later.
- **Xamarin.Android** - Xamarin.Android 10.2 or later must be installed with Visual Studio (Xamarin.Android is automatically installed as part of the **Mobile Development With .NET** workload on Windows and installed as part of the **Visual Studio for Mac Installer**).
- **Android SDK** - Android API 28 or higher must be installed via the Android SDK Manager.
- **Target Device or Emulator** - Your device or emulator must run Android 8.0 (API level 26) or higher.

- Visual Studio
- Visual Studio for Mac

Get started

To get started with Apply Changes, you will need to ensure a device or emulator is running Android 8.0 (API level 26) or higher. Then run your Android application with or without debugging.

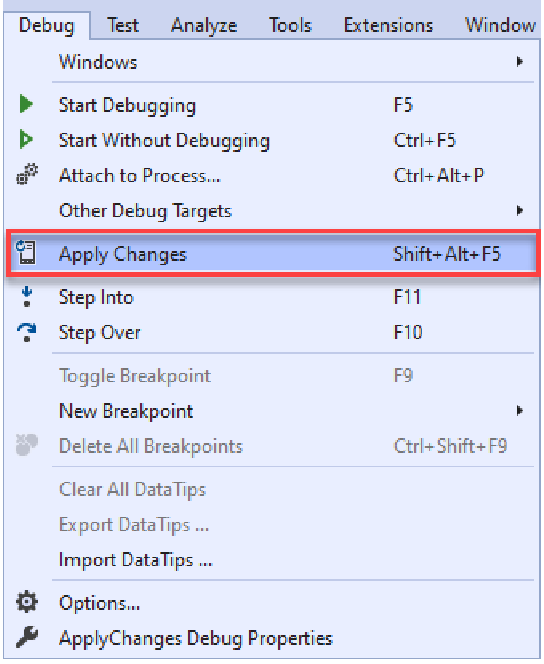
You can then interact with Apply Changes with the following approaches:

1. **Toolbar icon.** You can click on the Apply Changes toolbar icon to apply changes to your target device or emulator.

2. **Keyboard shortcut.** You can use the **Shift + Alt + F5** keyboard shortcut to apply changes to your target device or emulator.

3. **Debug menu.** You can use the **Debug > Apply Changes** menu item to apply changes to your target device or emulator.



Limitations

The following changes require an application restart:

- Changing C# code.
- Adding or removing a resource.
- Changing the AndroidManifest.xml.
- Changing native libraries (.so files).

Related links

- Apply Changes

Building Apps

11/2/2020 • 2 minutes to read • Edit Online

This section describes how the build process works and explains how to build ABI-specific APKs.

Build Process

This topic explains the steps and processes involved with the source code, resources, and assets of a Xamarin.Android application and producing an APK that can be installed on Android devices.

Build Targets

This topic explains the MSBuild targets that are involved for building, signing, installing, and running Android packages.

Build Properties

This topic explains the available MSBuild properties that can alter the behavior of the Build Targets.

Build Items

This topic explains the available MSBuild items that can alter the behavior of the Build Targets.

Building ABI Specific APKs

This guide discusses how to create Android APK's that support a single CPU architecture and ABI.

Build Process

3/12/2021 • 4 minutes to read • Edit Online

The Xamarin.Android build process is responsible for gluing everything together: generating `Resource.designer.cs`, supporting the `@(AndroidAsset)`, `@(AndroidResource)`, and other build actions, generating Android-callable wrappers, and generating a `.apk` for execution on Android devices.

Application Packages

In broad terms, there are two types of Android application packages (`.apk` files) which the Xamarin.Android build system can generate:

- **Release** builds, which are fully self-contained and don't require extra packages to execute. These are the packages that are provided to an App store.
- **Debug** builds, which are not.

These package types match the MSBuild `Configuration` which produces the package.

Shared Runtime

Prior to Xamarin.Android 11.2, the *shared runtime* was a pair of extra Android packages which provide the Base Class Library (`mscorlib.dll`, etc.) and the Android binding library (`Mono.Android.dll`, etc.). Debug builds rely upon the shared runtime in lieu of including the Base Class Library and Binding assemblies within the Android application package, allowing the Debug package to be smaller.

The shared runtime could be disabled in Debug builds by setting the `$(AndroidUseSharedRuntime)` property to `False`.

Support for the Shared Runtime was removed in Xamarin.Android 11.2.

Fast Deployment

Fast deployment works by further shrinking Android application package size. This is done by excluding the app's assemblies from the package, and instead deploying the app's assemblies directly to the application's internal `files` directory, usually located in `/data/data/com.some.package`. The internal `files` directory is not a globally writable folder, so the `run-as` tool is used to execute all the commands to copy the files into that directory.

This process speeds up the build/deploy/debug cycle because the package is not reinstalled when *only* assemblies are changed. Only the updated assemblies are resynchronized to the target device.

> **WARNING**
>
> Fast deployment is known to fail on devices which block `run-as`, which often includes devices older than Android 5.0. Fast deployment also fails for system applications (android:sharedUserId="android.uid.system") since `run-as` is also blocked for system applications.

Fast deployment is enabled by default, and may be disabled in Debug builds by setting the `$(EmbedAssembliesIntoApk)` property to `True`.

The Enhanced Fast Deployment mode can be used in conjunction with this feature to speed up deployments even further. This will deploy both assemblies, native libraries, typemaps and dexes to the `files` directory. But you should only really need to enable this if you are changing native libraries, bindings or Java code.

MSBuild Projects

The Xamarin.Android build process is based on MSBuild, which is also the project file format used by Visual Studio for Mac and Visual Studio. Ordinarily, users will not need to edit the MSBuild files by hand – the IDE creates fully functional projects and updates them with any changes made, and automatically invoke build targets as needed.

Advanced users may wish to do things not supported by the IDE's GUI, so the build process is customizable by editing the project file directly. This page documents only the Xamarin.Android-specific features and customizations – many more things are possible with the normal MSBuild items, properties and targets.

Binding Projects

The following MSBuild properties are used with Binding projects:

- `$(AndroidClassParser)`
- `$(AndroidCodegenTarget)`

`Resource.designer.cs` Generation

The Following MSBuild properties are used to control generation of the `Resource.designer.cs` file:

- `$(AndroidAapt2CompileExtraArgs)`
- `$(AndroidAapt2LinkExtraArgs)`
- `$(AndroidExplicitCrunch)`
- `$(AndroidR8IgnoreWarnings)`
- `$(AndroidResgenExtraArgs)`
- `$(AndroidResgenFile)`
- `$(AndroidUseAapt2)`
- `$(MonoAndroidResourcePrefix)`

Signing Properties

Signing properties control how the Application package is signed so that it may be installed onto an Android device. To allow quicker build iteration, the Xamarin.Android tasks do not sign packages during the build process, because signing is quite slow. Instead, they are signed (if necessary) before installation or during export, by the IDE or the *Install* build target. Invoking the *SignAndroidPackage* target will produce a package with the `-Signed.apk` suffix in the output directory.

By default, the signing target generates a new debug-signing key if necessary. If you wish to use a specific key, for example on a build server, the following MSBuild properties are used:

- `$(AndroidDebugKeyAlgorithm)`
- `$(AndroidDebugKeyValidity)`
- `$(AndroidDebugStoreType)`
- `$(AndroidKeyStore)`
- `$(AndroidSigningKeyAlias)`
- `$(AndroidSigningKeyPass)`

- `$(AndroidSigningKeyStore)`
- `$(AndroidSigningStorePass)`
- `$(JarsignerTimestampAuthorityCertificateAlias)`
- `$(JarsignerTimestampAuthorityUrl)`

`keytool` Option Mapping

Consider the following `keytool` invocation:

```
$ keytool -genkey -v -keystore filename.keystore -alias keystore.alias -keyalg RSA -keysize 2048 -validity
10000
Enter keystore password: keystore.filename password
Re-enter new password: keystore.filename password
...
Is CN=... correct?
  [no]:  yes

Generating 2,048 bit RSA key pair and self-signed certificate (SHA1withRSA) with a validity of 10,000 days
        for: ...
Enter key password for keystore.alias
        (RETURN if same as keystore password): keystore.alias password
[Storing filename.keystore]
```

To use the keystore generated above, use the property group:

```
<PropertyGroup>
    <AndroidKeyStore>True</AndroidKeyStore>
    <AndroidSigningKeyStore>filename.keystore</AndroidSigningKeyStore>
    <AndroidSigningStorePass>keystore.filename password</AndroidSigningStorePass>
    <AndroidSigningKeyAlias>keystore.alias</AndroidSigningKeyAlias>
    <AndroidSigningKeyPass>keystore.alias password</AndroidSigningKeyPass>
</PropertyGroup>
```

Build Extension Points

The Xamarin.Android build system exposes a few public extension points for users wanting to hook into our build process. To use one of these extension points you will need to add your custom target to the appropriate MSBuild property in a `PropertyGroup`. For example:

```
<PropertyGroup>
   <AfterGenerateAndroidManifest>
      $(AfterGenerateAndroidManifest);
      YourTarget;
   </AfterGenerateAndroidManifest>
</PropertyGroup>
```

Extension points include:

- `$(AfterGenerateAndroidManifest)
- `$(BeforeGenerateAndroidManifest)

A word of caution about extending the build process: If not written correctly, build extensions can affect your build performance, especially if they run on every build. It is highly recommended that you read the MSBuild documentation before implementing such extensions.

Target Definitions

The Xamarin.Android-specific parts of the build process are defined in `$(MSBuildExtensionsPath)\Xamarin\Android\Xamarin.Android.CSharp.targets`, but normal language-specific targets such as *Microsoft.CSharp.targets* are also required to build the assembly.

The following build properties must be set before importing any language targets:

```
<PropertyGroup>
  <TargetFrameworkIdentifier>MonoDroid</TargetFrameworkIdentifier>
  <MonoDroidVersion>v1.0</MonoDroidVersion>
  <TargetFrameworkVersion>v2.2</TargetFrameworkVersion>
</PropertyGroup>
```

All of these targets and properties can be included for C# by importing *Xamarin.Android.CSharp.targets*:

```
<Import Project="$(MSBuildExtensionsPath)\Xamarin\Android\Xamarin.Android.CSharp.targets" />
```

This file can easily be adapted for other languages.

Build Items

3/5/2021 • 7 minutes to read • Edit Online

Build items control how a Xamarin.Android application or library project is built.

AndroidAsset

Supports Android Assets, files that would be included in the `assets` folder in a Java Android project.

AndroidAarLibrary

The Build action of `AndroidAarLibrary` should be used to directly reference `.aar` files. This build action will be most commonly used by Xamarin Components. Namely to include references to `.aar` files which are required to get Google Play and other services working.

Files with this Build action will be treated in a similar fashion to the embedded resources found in Library projects. The `.aar` will be extracted into the intermediate directory. Then any assets, resource and `.jar` files will be included in the appropriate item groups.

AndroidAotProfile

Used to provide an AOT profile, for use with profile-guided AOT.

It can be also used from Visual Studio by setting the `AndroidAotProfile` build action to a file containing an AOT profile.

AndroidBoundLayout

Indicates that the layout file is to have code-behind generated for it in case when the `AndroidGenerateLayoutBindings` property is set to `false`. In all other aspects it is identical to `AndroidResource` described above. This action can be used **only** with layout files:

```
<AndroidBoundLayout Include="Resources\layout\Main.axml" />
```

AndroidEnvironment

Files with a Build action of `AndroidEnvironment` are used to initialize environment variables and system properties during process startup. The `AndroidEnvironment` Build action may be applied to multiple files, and they will be evaluated in no particular order (so don't specify the same environment variable or system property in multiple files).

AndroidFragmentType

Specifies the default fully qualified type to be used for all `<fragment>` layout elements when generating the layout bindings code. The property defaults to the standard Android `Android.App.Fragment` type.

AndroidJavaLibrary

Files with a Build action of `AndroidJavaLibrary` are Java Archives (`.jar` files) which will be included in the final

Android package.

AndroidJavaSource

Files with a Build action of `AndroidJavaSource` are Java source code which will be included in the final Android package.

AndroidLibrary

AndroidLibrary is a new build action for simplifying how `.jar` and `.aar` files are included in projects.

Any project can specify:

```
<ItemGroup>
  <AndroidLibrary Include="foo.jar" />
  <AndroidLibrary Include="bar.aar" />
</ItemGroup>
```

The result of the above code snippet has a different effect for each Xamarin.Android project type:

- Application and class library projects:
  - `foo.jar` maps to **AndroidJavaLibrary**.
  - `bar.aar` maps to **AndroidAarLibrary**.
- Java binding projects:
  - `foo.jar` maps to **EmbeddedJar**.
  - `foo.jar` maps to **EmbeddedReferenceJar** if `Bind="false"` metadata is added.
  - `bar.aar` maps to **LibraryProjectZip**.

This simplification means you can use **AndroidLibrary** everywhere.

This build action was added in Xamarin.Android 11.2.

AndroidLintConfig

The Build action 'AndroidLintConfig' should be used in conjunction with the `$(AndroidLintEnabled)` property. Files with this build action will be merged together and passed to the android `lint` tooling. They should be XML files which contain information on which tests to enable and disable.

See the lint documentation for more details.

AndroidManifestOverlay

The `AndroidManifestOverlay` build action can be used to provide additional `AndroidManifest.xml` files to the Manifest Merger tool. Files with this build action will be passed to the Manifest Merger along with the main `AndroidManifest.xml` file and any additional manifest files from references. These will then be merged into the final manifest.

You can use this build action to provide additional changes and settings to your app depending on your build configuration. For example, if you need to have a specific permission only while debugging, you can use the overlay to inject that permission when debugging. For example, given the following overlay file contents:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
  <uses-permission android:name="android.permission.CAMERA" />
</manifest>
```

You can use the following to add this for a debug build:

```
<ItemGroup>
  <AndroidManifestOverlay Include="DebugPermissions.xml" Condition=" '$(Configuration)' == 'Debug' " />
</ItemGroup>
```

This build action was introduced in Xamarin.Android 11.2.

AndroidNativeLibrary

Native libraries are added to the build by setting their Build action to `AndroidNativeLibrary`.

Note that since Android supports multiple Application Binary Interfaces (ABIs), the build system must know which ABI the native library is built for. There are two ways this can be done:

1. Path "sniffing".
2. Using the `Abi` item attribute.

With path sniffing, the parent directory name of the native library is used to specify the ABI that the library targets. Thus, if you add `lib/armeabi-v7a/libfoo.so` to the build, then the ABI will be "sniffed" as `armeabi-v7a`.

Item Attribute Name

Abi – Specifies the ABI of the native library.

```
<ItemGroup>
  <AndroidNativeLibrary Include="path/to/libfoo.so">
    <Abi>armeabi-v7a</Abi>
  </AndroidNativeLibrary>
</ItemGroup>
```

AndroidResource

All files with an *AndroidResource* build action are compiled into Android resources during the build process and made accessible via `$(AndroidResgenFile)`.

```
<ItemGroup>
  <AndroidResource Include="Resources\values\strings.xml" />
</ItemGroup>
```

More advanced users might perhaps wish to have different resources used in different configurations but with the same effective path. This can be achieved by having multiple resource directories and having files with the same relative paths within these different directories, and using MSBuild conditions to conditionally include different files in different configurations. For example:

```
<ItemGroup Condition="'$(Configuration)'!='Debug'">
  <AndroidResource Include="Resources\values\strings.xml" />
</ItemGroup>
<ItemGroup  Condition="'$(Configuration)'=='Debug'">
  <AndroidResource Include="Resources-Debug\values\strings.xml"/>
</ItemGroup>
<PropertyGroup>
  <MonoAndroidResourcePrefix>Resources;Resources-Debug</MonoAndroidResourcePrefix>
</PropertyGroup>
```

LogicalName – Specifies the resource path explicitly. Allows “aliasing” files so that they will be available as multiple distinct resource names.

```
<ItemGroup Condition="'$(Configuration)'!='Debug'">
  <AndroidResource Include="Resources/values/strings.xml"/>
</ItemGroup>
<ItemGroup Condition="'$(Configuration)'=='Debug'">
  <AndroidResource Include="Resources-Debug/values/strings.xml">
    <LogicalName>values/strings.xml</LogicalName>
  </AndroidResource>
</ItemGroup>
```

AndroidResourceAnalysisConfig

The Build action `AndroidResourceAnalysisConfig` marks a file as a severity level configuration file for the Xamarin Android Designer layout diagnostics tool. This is currently only used in the layout editor and not for build messages.

See the Android Resource Analysis documentation for more details.

Added in Xamarin.Android 10.2.

Content

The normal `Content` Build action is not supported (as we haven't figured out how to support it without a possibly costly first-run step).

Starting in Xamarin.Android 5.1, attempting to use the `@(Content)` Build action will result in a `XA0101` warning.

EmbeddedJar

In a Xamarin.Android binding project, the **EmbeddedJar** build action binds the Java/Kotlin library and embeds the `.jar` file into the library. When a Xamarin.Android application project consumes the library, it will have access to the Java/Kotlin APIs from C# as well as include the Java/Kotlin code in the final Android application.

Since Xamarin.Android 11.2, you can use the **AndroidLibrary** build action as an alternative such as:

```
<Project>
  <ItemGroup>
    <AndroidLibrary Include="Library.jar" />
  </ItemGroup>
</Project>
```

EmbeddedNativeLibrary

In a Xamarin.Android class library or Java binding project, the **EmbeddedNativeLibrary** build action bundles a native library such as `lib/armeabi-v7a/libfoo.so` into the library. When a Xamarin.Android application consumes the library, the `libfoo.so` file will be included in the final Android application.

Since Xamarin.Android 11.2, you can use the **AndroidNativeLibrary** build action as an alternative.

EmbeddedReferenceJar

In a Xamarin.Android binding project, the **EmbeddedReferenceJar** build action embeds the `.jar` file into the library but does not create a C# binding as **EmbeddedJar** does. When a Xamarin.Android application project consumes the library, it will include the Java/Kotlin code in the final Android application.

Since Xamarin.Android 11.2, you can use the **AndroidLibrary** build action as an alternative such as `<AndroidLibrary Include="..." Bind="false" />`:

```
<Project>
  <ItemGroup>
    <!-- A .jar file to bind & embed -->
    <AndroidLibrary Include="Library.jar" />
    <!-- A .jar file to only embed -->
    <AndroidLibrary Include="Dependency.jar" Bind="false" />
  </ItemGroup>
</Project>
```

JavaDocJar

In a Xamarin.Android binding project, the **JavaDocJar** build action is used on `.jar` files which contain *Javadoc HTML*. The Javadoc HTML is parsed in order to extract parameter names.

Only certain "Javadoc HTML dialects" are supported, including:

- JDK 1.7 `javadoc` output.
- JDK 1.8 `javadoc` output.
- Droiddoc output.

This build action is deprecated in Xamarin.Android 11.3, and will not be supported in .NET 6. The `@(JavaSourceJar)` build action is preferred.

JavaSourceJar

In a Xamarin.Android binding project, the **JavaSourceJar** build action is used on `.jar` files that contain *Java source code*, which contain Javadoc documentation comments.

Prior to Xamarin.Android 11.3, the Javadoc would be converted into HTML via the `javadoc` utility during build time, and later turned into XML documentation.

Starting with Xamarin.Android 11.3, Javadoc will instead be converted into C# XML Documentation Comments within the generated binding source code.

`$(AndroidJavadocVerbosity)` controls how "verbose" or "complete" the imported Javadoc is.

Starting in Xamarin.Android 11.3, the following MSBuild metadata is supported:

- `%(CopyrightFile)`: A path to a file that contains copyright information for the Javadoc contents, which will be appended to all imported documentation.
- `%(UrlPrefix)`: A URL prefix to support linking to online documentation within imported documentation.
- `%(UrlStyle)`: The "style" of URLs to generate when linking to online documentation. Only one style is currently supported: `developer.android.com/reference@2020-Nov`.

LibraryProjectZip

In a Xamarin.Android binding project, the **LibraryProjectZip** build action binds the Java/Kotlin library and embeds the `.zip` or `.aar` file into the library. When a Xamarin.Android application project consumes the library, it will have access to the Java/Kotlin APIs from C# as well as include the Java/Kotlin code in the final Android application.

NOTE

Only a single **LibraryProjectZip** can be included in a Xamarin.Android binding project. This limitation will be removed in .NET 6.

LinkDescription

Files with a *LinkDescription* build action are used to control linker behavior.

ProguardConfiguration

Files with a *ProguardConfiguration* build action contain options which are used to control `proguard` behavior. For more information about this build action, see ProGuard.

These files are ignored unless the `$(EnableProguard)` MSBuild property is `True`.

Build Properties

3/5/2021 • 28 minutes to read • Edit Online

MSBuild properties control the behavior of the targets. They are specified within the project file, for example **MyApp.csproj**, within an MSBuild PropertyGroup.

AdbTarget

The `$(AdbTarget)` property specifies the Android target device the Android package may be installed to or removed from. The value of this property is the same as the `adb` Target Device option.

AfterGenerateAndroidManifest

MSBuild Targets listed in this property will run directly after the internal `_GenerateJavaStubs` target, which is where the `AndroidManifest.xml` file is generated in the `$(IntermediateOutputPath)`. If you want to make any modifications to the generated `AndroidManifest.xml` file, you can do that using this extension point.

Added in Xamarin.Android 9.4.

AndroidAapt2CompileExtraArgs

Specifies additional command-line options to pass to the **aapt2 compile** command when processing Android assets and resources.

Added in Xamarin.Android 9.1.

AndroidAapt2LinkExtraArgs

Specifies additional command-line options to pass to the **aapt2 link** command when processing Android assets and resources.

Added in Xamarin.Android 9.1.

AndroidAddKeepAlives

A boolean property that controls whether the linker will insert `GC.KeepAlive()` invocations within binding projects to prevent premature object collection.

Defaults to `True` for Release configuration builds.

This property was added in Xamarin.Android 11.2.

AndroidAotCustomProfilePath

The file that `aprofutil` should create to hold profiler data.

AndroidAotProfiles

A string property that allows the developer to add AOT profiles from the command line. It is a semicolon or comma-separated list of absolute paths. Added in Xamarin.Android 10.1.

AndroidAotProfilerPort

The port that `aprofutil` should connect to when obtaining profiling data.

AndroidApkDigestAlgorithm

A string value which specifies the digest algorithm to use with `jarsigner -digestalg`.

The default value is `SHA-256`. In Xamarin.Android 10.0 and earlier, the default value was `SHA1`.

Added in Xamarin.Android 9.4.

AndroidApkSignerAdditionalArguments

A string property which allows the developer to provide additional arguments to the `apksigner` tool.

Added in Xamarin.Android 8.2.

AndroidApkSigningAlgorithm

A string value which specifies the signing algorithm to use with `jarsigner -sigalg`.

The default value is `SHA256withRSA`. In Xamarin.Android 10.0 and earlier, the default value was `md5withRSA`.

Added in Xamarin.Android 8.2.

AndroidApplication

A boolean value that indicates whether the project is for an Android Application (`True`) or for an Android Library Project (`False` or not present).

Only one project with `<AndroidApplication>True</AndroidApplication>` may be present within an Android package. (Unfortunately this is not yet verified, which can result in subtle and bizarre errors regarding Android resources.)

AndroidApplicationJavaClass

The full Java class name to use in place of `android.app.Application` when a class inherits from Android.App.Application.

This property is generally set by *other* properties, such as the `$(AndroidEnableMultiDex)` MSBuild property.

Added in Xamarin.Android 6.1.

AndroidBinUtilsPath

A path to a directory containing the Android binutils such as `ld`, the native linker, and `as`, the native assembler. These tools are part of the Android NDK and are also included in the Xamarin.Android installation.

The default value is `$(MonoAndroidBinDirectory)\ndk\`.

Added in Xamarin.Android 10.0.

AndroidBoundExceptionType

A string value that specifies how exceptions should be propagated when a Xamarin.Android-provided type implements a .NET type or interface in terms of Java types, for example `Android.Runtime.InputStreamInvoker` and

`System.IO.Stream`, or `Android.Runtime.JavaDictionary` and `System.Collections.IDictionary`.

- `Java`: The original Java exception type is propagated as-is.

  This means that, for example, `InputStreamInvoker` does not properly implement the `System.IO.Stream` API because `Java.IO.IOException` may be thrown from `Stream.Read()` instead of `System.IO.IOException`.

  This is the exception propagation behavior in all releases of Xamarin.Android prior to 11.1.

  This is the default value in Xamarin.Android 11.1.

- `System`: The original Java exception type is caught and wrapped in an appropriate .NET exception type.

  This means that, for example, `InputStreamInvoker` properly implements `System.IO.Stream`, and `Stream.Read()` will *not* throw `Java.IO.IOException` instances. (It may instead throw a `System.IO.IOException` which has a `Java.IO.IOException` as the `Exception.InnerException` value.)

  This will become the default value in .NET 6.0.

Added in Xamarin.Android 10.2.

AndroidBuildApplicationPackage

A boolean value that indicates whether to create and sign the package (.apk). Setting this value to `True` is equivalent to using the `SignAndroidPackage` build target.

Support for this property was added after Xamarin.Android 7.1.

This property is `False` by default.

AndroidBundleConfigurationFile

Specifies a filename to use as a configuration file for `bundletool` when building an Android App Bundle. This file controls some aspects of how APKs are generated from the bundle, such as on what dimensions the bundle is split to produce APKs. Note that Xamarin.Android configures some of these settings automatically, including the list of file extensions to leave uncompressed.

This property is only relevant if `$(AndroidPackageFormat)` is set to `aab`.

Added in Xamarin.Android 10.3.

AndroidClassParser

A string property which controls how `.jar` files are parsed. Possible values include:

- **class-parse**: Uses `class-parse.exe` to parse Java bytecode directly, without assistance of a JVM. This value is experimental.

- **jar2xml**: Use `jar2xml.jar` to use Java reflection to extract types and members from a `.jar` file.

The advantages of `class-parse` over `jar2xml` are:

- `class-parse` can extract parameter names from Java bytecode which contains *debug* symbols (bytecode compiled with `javac -g`).

- `class-parse` doesn't "skip" classes which inherit from or contain members of unresolvable types.

Experimental. Added in Xamarin.Android 6.0.

The default value is `jar2xml`.

Support for `jar2xml` is obsolete, and support for `jar2xml` will be removed as part of .NET 6.

AndroidCodegenTarget

A string property which controls the code generation target ABI. Possible values include:

- **XamarinAndroid**: Uses the JNI binding API present since Mono for Android 1.0. Binding assemblies built with Xamarin.Android 5.0 or later can only run on Xamarin.Android 5.0 or later (API/ABI additions), but the *source* is compatible with prior product versions.

- **XAJavaInterop1**: Use Java.Interop for JNI invocations. Binding assemblies using `XAJavaInterop1` can only build and execute with Xamarin.Android 6.1 or later. Xamarin.Android 6.1 and later bind `Mono.Android.dll` with this value.

The benefits of `XAJavaInterop1` include:

- Smaller assemblies.

- `jmethodID` caching for `base` method invocations, so long as all other binding types in the inheritance hierarchy are built with `XAJavaInterop1` or later.

- `jmethodID` caching of Java Callable Wrapper constructors for managed subclasses.

The default value is `XAJavaInterop1`.

AndroidCreatePackagePerAbi

A boolean property that determines if a *set* of files -- on per ABI specified in `$(AndroidSupportedAbis)` -- should be created instead of having support for all ABIs in a single `.apk`.

See also the Building ABI-Specific APKs guide.

AndroidDebugKeyAlgorithm

Specifies the default algorithm to use for the `debug.keystore`. It defaults to `RSA`.

AndroidDebugKeyValidity

Specifies the default validity to use for the `debug.keystore`. It defaults to `10950` or `30 * 365` or `30 years`.

AndroidDebugStoreType

Specifies the key store file format to use for the `debug.keystore`. It defaults to `pkcs12`.

Added in Xamarin.Android 10.2.

AndroidDeviceUserId

Allows deploying and debugging the application under guest or work accounts. The value is the `uid` value you get from the following adb command:

```
adb shell pm list users
```

This will return the following data:

```
Users:
 UserInfo{0:Owner:c13} running
 UserInfo{10:Guest:404}
```

The `uid` is the first integer value. In the example they are `0` and `10`.

This property was added in Xamarin.Android 11.2.

AndroidDexTool

An enum-style property with valid values of `dx` or `d8`. Indicates which Android dex compiler is used during the Xamarin.Android build process. Currently defaults to `dx`. For further information see our documentation on D8 and R8.

AndroidEnableDesugar

A boolean property that determines if `desugar` is enabled. Android does not currently support all Java 8 features, and the default toolchain implements the new language features by performing bytecode transformations, called `desugar`, on the output of the `javac` compiler. Defaults to `False` if using `AndroidDexTool=dx` and defaults to `True` if using `$(AndroidDexTool)` = `d8`.

AndroidEnableGooglePlayStoreChecks

A bool property which allows developers to disable the following Google Play Store checks: XA1004, XA1005 and XA1006. This is useful for developers who are not targeting the Google Play Store and do not wish to run those checks.

Added in Xamarin.Android 9.4.

AndroidEnableMultiDex

A boolean property that determines whether or not multi-dex support will be used in the final `.apk`.

Support for this property was added in Xamarin.Android 5.1.

This property is `False` by default.

AndroidEnablePreloadAssemblies

A boolean property which controls whether or not all managed assemblies bundled within the application package are loaded during process startup or not.

When set to `True`, all assemblies bundled within the application package will be loaded during process startup, before any application code is invoked. This is consistent with what Xamarin.Android did in releases prior to Xamarin.Android 9.2.

When set to `False`, assemblies will only be loaded on an as-needed basis. This allows applications to startup faster, and is also more consistent with desktop .NET semantics. To see the time savings, set the `debug.mono.log` System Property to include `timing`, and look for the `Finished loading assemblies: preloaded` message within `adb logcat`.

Applications or libraries which use dependency injection may *require* that this property be `True` if they in turn require that `AppDomain.CurrentDomain.GetAssemblies()` return all assemblies within the application bundle, even if the assembly wouldn't otherwise have been needed.

By default this value will be set to `True`.

Added in Xamarin.Android 9.2.

AndroidEnableProfiledAot

A boolean property that determines whether or not the AOT profiles are used during Ahead-of-Time compilation.

The profiles are listed in `@(AndroidAotProfile)` item group. This ItemGroup contains default profile(s). It can be overridden by removing the existing one(s) and adding your own AOT profiles.

Support for this property was added in Xamarin.Android 9.4.

This property is `False` by default.

AndroidEnableSGenConcurrent

A boolean property that determines whether or not Mono's concurrent GC collector will be used.

Support for this property was added in Xamarin.Android 7.2.

This property is `False` by default.

AndroidErrorOnCustomJavaObject

A boolean property that determines whether types may implement `Android.Runtime.IJavaObject` *without* also inheriting from `Java.Lang.Object` or `Java.Lang.Throwable`:

```
class BadType : IJavaObject {
    public IntPtr Handle {
        get {return IntPtr.Zero;}
    }

    public void Dispose()
    {
    }
}
```

When True, such types will generate an XA4212 error, otherwise an XA4212 warning will be generated.

Support for this property was added in Xamarin.Android 8.1.

This property is `True` by default.

AndroidExplicitCrunch

No longer supported in Xamarin.Android 11.0.

AndroidExtraAotOptions

A string property that allows passing additional options to the Mono compiler during the `Aot` task for projects that have either `$(AndroidEnableProfiledAot)` or `$(AotAssemblies)` set to `true`. The string value of the property is added to the response file when calling the Mono cross-compiler.

In general, this property should be left blank, but in certain special scenarios it might provide useful flexibility.

Note that this property is different from the related `$(AndroidAotAdditionalArguments)` property. That property places comma-separated arguments into the `--aot` option of the Mono compiler. `$(AndroidExtraAotOptions)` instead passes full standalone space-separated options like `--verbose` or `--debug` to the compiler.

Added in Xamarin.Android 10.2.

AndroidFastDeploymentType

A `:` (colon)-separated list of values to control what types can be deployed to the Fast Deployment directory on the target device when the `$(EmbedAssembliesIntoApk)` MSBuild property is `False`. If a resource is fast deployed, it is *not* embedded into the generated `.apk`, which can speed up deployment times. (The more that is fast deployed, then the less frequently the `.apk` needs to be rebuilt, and the install process can be faster.) Valid values include:

- `Assemblies`: Deploy application assemblies.
- `Dexes`: Deploy `.dex` files, native libraries and typemaps. **This value can *only* be used on devices running Android 4.4 or later (API-19).**

The default value is `Assemblies`.

Support for Fast Deploying resources and assets via that system was removed in commit f0d565fe. This was becuase it required the use of deprecated API's to work.

Experimental. This property was added in Xamarin.Android 6.1.

AndroidGenerateJniMarshalMethods

A bool property which enables generating of JNI marshal methods as part of the build process. This greatly reduces the `System.Reflection` usage in the binding helper code.

By default this will be set to False. If the developers wish to use the new JNI marshal methods feature, they can set

```
<AndroidGenerateJniMarshalMethods>True</AndroidGenerateJniMarshalMethods>
```

in their `.csproj`. Alternatively provide the property on the command line via

```
/p:AndroidGenerateJniMarshalMethods=True
```

Experimental. Added in Xamarin.Android 9.2. The default value is False.

AndroidGenerateJniMarshalMethodsAdditionalArguments

A string property which can be used to add additional parameters to the `jnimarshalmethod-gen.exe` invocation. This is useful for debugging, so that options such as `-v`, `-d`, or `--keeptemp` can be used.

Default value is empty string. It can be set in the `.csproj` file or on the command line. For example:

```
<AndroidGenerateJniMarshalMethodsAdditionalArguments>-v -d --
keeptemp</AndroidGenerateJniMarshalMethodsAdditionalArguments>
```

or:

```
/p:AndroidGenerateJniMarshalMethodsAdditionalArguments="-v -d --keeptemp"
```

Added in Xamarin.Android 9.2.

AndroidGenerateLayoutBindings

Enables generation of layout code-behind if set to `true` or disables it completely if set to `false`. The default value is `false`.

AndroidHttpClientHandlerType

Controls the default `System.Net.Http.HttpMessageHandler` implementation which will be used by the `System.Net.Http.HttpClient` default constructor. The value is an assembly-qualified type name of an `HttpMessageHandler` subclass, suitable for use with `System.Type.GetType(string)`. The most common values for this property are:

- `Xamarin.Android.Net.AndroidClientHandler`: Use the Android Java APIs to perform network requests. This allows accessing TLS 1.2 URLs when the underlying Android version supports TLS 1.2. Only Android 5.0 and later reliably provide TLS 1.2 support through Java.

  This corresponds to the **Android** option in the Visual Studio property pages and the **AndroidClientHandler** option in the Visual Studio for Mac property pages.

  The new project wizard selects this option for new projects when the **Minimum Android Version** is configured to **Android 5.0 (Lollipop)** or higher in Visual Studio or when **Target Platforms** is set to **Latest and Greatest** in Visual Studio for Mac.

- Unset/the empty string: This is equivalent to `System.Net.Http.HttpClientHandler, System.Net.Http`

  This corresponds to the **Default** option in the Visual Studio property pages.

  The new project wizard selects this option for new projects when the **Minimum Android Version** is configured to **Android 4.4.87** or lower in Visual Studio or when **Target Platforms** is set to **Modern Development** or **Maximum Compatibility** in Visual Studio for Mac.

- `System.Net.Http.HttpClientHandler, System.Net.Http`: Use the managed `HttpMessageHandler`.

  This corresponds to the **Managed** option in the Visual Studio property pages.

> **NOTE**
>
> If TLS 1.2 support is required on Android versions prior to 5.0, *or* if TLS 1.2 support is required with the `System.Net.WebClient` and related APIs, then `$(AndroidTlsProvider)` should be used.

> **NOTE**
>
> Support for this property works by setting the `XA_HTTP_CLIENT_HANDLER_TYPE` environment variable. A `$XA_HTTP_CLIENT_HANDLER_TYPE` value found in a file with a Build action of `@(AndroidEnvironment)` will take precedence.

Added in Xamarin.Android 6.1.

AndroidIncludeWrapSh

A boolean value which indicates whether the Android wrapper script (`wrap.sh`) should be packaged into the APK. The property defaults to `false` since the wrapper script may significantly influence the way the application starts up and works and the script should be included only when necessary e.g. for debugging or otherwise changing the application startup/runtime behavior.

The script is added to the project using the `@(AndroidNativeLibrary)` build action, because it is placed in the same directory as architecture-specific native libraries, and must be named `wrap.sh`.

The easiest way to specify path to the `wrap.sh` script is to put it in a directory named after the target architecture. This approach will work if you have just one `wrap.sh` per architecture:

```
<AndroidNativeLibrary Include="path/to/arm64-v8a/wrap.sh" />
```

However, if your project needs more than one `wrap.sh` per architecture, for different purposes, this approach won't work. Instead, in such cases the name can be specified using the `Link` metadata of the `AndroidNativeLibrary`:

```
<AndroidNativeLibrary Include="/path/to/my/arm64-wrap.sh">
  <Link>lib\arm64-v8a\wrap.sh</Link>
</AndroidNativeLibrary>
```

If the `Link` metadata is used, the path specified in its value must be a valid native architecture-specific library path, relative to the APK root directory. The format of the path is `lib\ARCH\wrap.sh` where `ARCH` can be one of:

- `arm64-v8a`
- `armeabi-v7a`
- `x86_64`
- `x86`

AndroidKeyStore

A boolean value which indicates whether custom signing information should be used. The default value is `False`, meaning that the default debug-signing key will be used to sign packages.

AndroidLaunchActivity

The Android activity to launch.

AndroidLinkMode

Specifies which type of linking should be performed on assemblies contained within the Android package. Only used in Android Application projects. The default value is *SdkOnly*. Valid values are:

- **None**: No linking will be attempted.
- **SdkOnly**: Linking will be performed on the base class libraries only, not user's assemblies.
- **Full**: Linking will be performed on base class libraries and user assemblies.

> **NOTE**
>
> Using an `AndroidLinkMode` value of *Full* often results in broken apps, particularly when Reflection is used. Avoid unless you *really* know what you're doing.

```
<AndroidLinkMode>SdkOnly</AndroidLinkMode>
```

AndroidLinkSkip

Specifies a semicolon-delimited (`;`) list of assembly names, without file extensions, of assemblies that should not be linked. Only used within Android Application projects.

```
<AndroidLinkSkip>Assembly1;Assembly2</AndroidLinkSkip>
```

AndroidLinkTool

An enum-style property with valid values of `proguard` or `r8`. Indicates which code shrinker is used for Java code. Currently defaults to an empty string, or `proguard` if `$(AndroidEnableProguard)` is `True`. For further information see our documentation on D8 and R8.

AndroidLintEnabled

A bool property which allows the developer to run the android `lint` tool as part of the packaging process.

When `$(AndroidLintEnabled)`=True, the following properties are used:

- `$(AndroidLintEnabledIssues)`:
- `$(AndroidLintDisabledIssues)`:
- `$(AndroidLintCheckIssues)`:

The following build actions may also be used:

- `@(AndroidLintConfig)`:

See Lint Help for more details on the android `lint` tooling.

AndroidLintEnabledIssues

This property is only used when `$(AndroidLintEnabled)`=True.

A comma-separated list of lint issues to enable.

AndroidLintDisabledIssues

This property is only used when `$(AndroidLintEnabled)`=True.

A comma-separated list of lint issues to disable.

AndroidLintCheckIssues

This property is only used when `$(AndroidLintEnabled)`=True.

A comma-separated list of lint issues to check.

Note: only these issues will be checked.

AndroidManagedSymbols

A boolean property that controls whether sequence points are generated so that file name and line number information can be extracted from `Release` stack traces.

Added in Xamarin.Android 6.1.

AndroidManifest

Specifies a filename to use as the template for the app's `AndroidManifest.xml`. During the build, any other necessary values will be merged into to produce the actual `AndroidManifest.xml`. The `$(AndroidManifest)` must contain the package name in the `/manifest/@package` attribute.

AndroidManifestMerger

Specifies the implementation for merging *AndroidManifest.xml* files. This is an enum-style property where `legacy` selects the original C# implementation and `manifestmerger.jar` selects Google's Java implementation.

The default value is currently `legacy`. This will change to `manifestmerger.jar` in a future release to align behavior with Android Studio.

Google's merger enables support for `xmlns:tools="http://schemas.android.com/tools"` as described in the Android documentation.

Introduced in Xamarin.Android 10.2

AndroidManifestPlaceholders

A semicolon-separated list of key-value replacement pairs for *AndroidManifest.xml*, where each pair has the format `key=value`.

For example, a property value of `assemblyName=$(AssemblyName)` defines an `${assemblyName}` placeholder that can then appear in *AndroidManifest.xml*:

```
<application android:label="${assemblyName}"
```

This provides a way to insert variables from the build process into the *AndroidManifest.xml* file.

AndroidMultiDexClassListExtraArgs

A string property which allows developers to pass additional arguments to the `com.android.multidex.MainDexListBuilder` when generating the `multidex.keep` file.

One specific case is if you are getting the following error during the `dx` compilation.

```
com.android.dex.DexException: Too many classes in --main-dex-list, main dex capacity exceeded
```

If you are getting this error you can add the following to the `.csproj`.

```
<DxExtraArguments>--force-jumbo </DxExtraArguments>
<AndroidMultiDexClassListExtraArgs>--disable-annotation-resolution-
workaround</AndroidMultiDexClassListExtraArgs>
```

this should allow the `dx` step to succeed.

Added in Xamarin.Android 8.3.

AndroidPackageFormat

An enum-style property with valid values of `apk` or `aab`. This indicates if you want to package the Android application as an APK file or Android App Bundle. App Bundles are a new format for `Release` builds that are intended for submission on Google Play. This value currently defaults to `apk`.

When `$(AndroidPackageFormat)` is set to `aab`, other MSBuild properties are set, which are required for Android App Bundles:

- `$(AndroidUseAapt2)` is `True`.
- `$(AndroidUseApkSigner)` is `False`.
- `$(AndroidCreatePackagePerAbi)` is `False`.

AndroidPackageNamingPolicy

An enum-style property for specifying the Java package names of generated Java source code.

In Xamarin.Android 10.2 and later, the only supported value is `LowercaseCrc64`.

In Xamarin.Android 10.1, a transitional `LowercaseMD5` value was also available that allowed switching back to the original Java package name style as used in Xamarin.Android 10.0 and earlier. That option was removed in Xamarin.Android 10.2 to improve compatibility with build environments that have FIPS compliance enforced.

Added in Xamarin.Android 10.1.

AndroidProguardMappingFile

Specifies the `-printmapping` proguard rule for `r8`. This will mean the `mapping.txt` file will be produced in the `$(OutputPath)` folder. This file can then be used when uploading packages to the Google Play Store.

The default value is `$(OutputPath)mapping.txt`.

This property was added in Xamarin.Android 11.2.

AndroidR8IgnoreWarnings

Specifies the `-ignorewarnings` proguard rule for `r8`. This allows `r8` to continue with dex compilation even if certain warnings are encountered. Defaults to `True`, but can be set to `False` to enforce more strict behavior. See the ProGuard manual for details.

Added in Xamarin.Android 10.3.

AndroidR8JarPath

The path to `r8.jar` for use with the r8 dex-compiler and shrinker. Defaults to a path in the Xamarin.Android installation. For further information see our documentation on D8 and R8.

AndroidResgenExtraArgs

Specifies additional command-line options to pass to the **aapt** command when processing Android assets and resources.

AndroidResgenFile

Specifies the name of the Resource file to generate. The default template sets this to `Resource.designer.cs`.

AndroidSdkBuildToolsVersion

The Android SDK build-tools package provides the **aapt** and **zipalign** tools, among others. Multiple different versions of the build-tools package may be installed simultaneously. The build-tools package chosen for packaging is done by checking for and using a "preferred" build-tools version if it is present; if the "preferred" version is *not* present, then the highest versioned installed build-tools package is used.

The `$(AndroidSdkBuildToolsVersion)` MSBuild property contains the preferred build-tools version. The Xamarin.Android build system provides a default value in `Xamarin.Android.Common.targets`, and the default value may be overridden within your project file to choose an alternate build-tools version, if (for example) the latest aapt is crashing out while a previous aapt version is known to work.

AndroidSigningKeyAlias

Specifies the alias for the key in the keystore. This is the **keytool -alias** value used when creating the keystore.

AndroidSigningKeyPass

Specifies the password of the key within the keystore file. This is the value entered when `keytool` asks **Enter key password for $(AndroidSigningKeyAlias)**.

In Xamarin.Android 10.0 and earlier, this property only supports plain text passwords.

In Xamarin.Android 10.1 and later, this property also supports `env:` and `file:` prefixes that can be used to specify an environment variable or file that contains the password. These options provide a way to prevent the password from appearing in build logs.

For example, to use an environment variable named *AndroidSigningPassword*:

```
<PropertyGroup>
    <AndroidSigningKeyPass>env:AndroidSigningPassword</AndroidSigningKeyPass>
</PropertyGroup>
```

To use a file located at `C:\Users\user1\AndroidSigningPassword.txt`:

```
<PropertyGroup>
    <AndroidSigningKeyPass>file:C:\Users\user1\AndroidSigningPassword.txt</AndroidSigningKeyPass>
</PropertyGroup>
```

> **NOTE**
> The `env:` prefix is not supported when `$(AndroidPackageFormat)` is set to `aab`.

AndroidSigningKeyStore

Specifies the filename of the keystore file created by `keytool`. This corresponds to the value provided to the **keytool -keystore** option.

AndroidSigningStorePass

Specifies the password to `$(AndroidSigningKeyStore)`. This is the value provided to `keytool` when creating the keystore file and asked **Enter keystore password:**.

In Xamarin.Android 10.0 and earlier, this property only supports plain text passwords.

In Xamarin.Android 10.1 and later, this property also supports `env:` and `file:` prefixes that can be used to specify an environment variable or file that contains the password. These options provide a way to prevent the password from appearing in build logs.

For example, to use an environment variable named *AndroidSigningPassword*:

```
<PropertyGroup>
    <AndroidSigningStorePass>env:AndroidSigningPassword</AndroidSigningStorePass>
</PropertyGroup>
```

To use a file located at `C:\Users\user1\AndroidSigningPassword.txt`:

```
<PropertyGroup>
    <AndroidSigningStorePass>file:C:\Users\user1\AndroidSigningPassword.txt</AndroidSigningStorePass>
</PropertyGroup>
```

NOTE

The `env:` prefix is not supported when `$(AndroidPackageFormat)` is set to `aab`.

AndroidSupportedAbis

A string property that contains a semicolon (`;`)-delimited list of ABIs which should be included into the `.apk`.

Supported values include:

- `armeabi-v7a`
- `x86`
- `arm64-v8a`: Requires Xamarin.Android 5.1 and later.
- `x86_64`: Requires Xamarin.Android 5.1 and later.

AndroidTlsProvider

A string value which specifies which TLS provider should be used in an application. Possible values are:

- Unset/the empty string: In Xamarin.Android 7.3 and higher, this is equivalent to `btls`.

  In Xamarin.Android 7.1, this is equivalent to `legacy`.

  This corresponds to the **Default** setting in the Visual Studio property pages.

- `btls`: Use Boring SSL for TLS communication with HttpWebRequest.

  This allows use of TLS 1.2 on all Android versions.

  This corresponds to the **Native TLS 1.2+** setting in the Visual Studio property pages.

- `legacy`: In Xamarin.Android 10.1 and earlier, use the historical managed SSL implementation for network interaction. This *does not* support TLS 1.2.

  This corresponds to the **Managed TLS 1.0** setting in the Visual Studio property pages.

  In Xamarin.Android 10.2 and later, this value is ignored and the `btls` setting is used.

- `default`: This value is unlikely to be used in Xamarin.Android projects. The recommended value to use instead is the empty string, which corresponds to the **Default** setting in the Visual Studio property pages.

The `default` value is not offered in the Visual Studio property pages.

This is currently equivalent to `legacy`.

Added in Xamarin.Android 7.1.

AndroidUseAapt2

A boolean property which allows the developer to control the use of the `aapt2` tool for packaging. By default this will be False and Xamarin.Android will use `aapt`. If the developer wishes to use the new `aapt2` functionality, add:

```
<AndroidUseAapt2>True</AndroidUseAapt2>
```

in their `.csproj`. Alternatively provide the property on the command line:

```
/p:AndroidUseAapt2=True
```

This property was added in Xamarin.Android 8.3. Setting `AndroidUseAapt2` to `false` is deprecated in Xamarin.Android 11.2.

AndroidUseApkSigner

A bool property which allows the developer to use the `apksigner` tool rather than `jarsigner`.

Added in Xamarin.Android 8.2.

AndroidUseDefaultAotProfile

A bool property that allows the developer to suppress usage of the default AOT profiles.

To suppress the default AOT profiles, set the property to `false`.

Added in Xamarin.Android 10.1.

AndroidUseLegacyVersionCode

A boolean property which allows the developer to revert the versionCode calculation back to its old pre Xamarin.Android 8.2 behavior. This should ONLY be used for developers with existing applications in the Google Play Store. It is highly recommended that the new `$(AndroidVersionCodePattern)` property is used.

Added in Xamarin.Android 8.2.

AndroidUseManagedDesignTimeResourceGenerator

A boolean property which will switch over the design time builds to use the managed resource parser rather than `aapt`.

Added in Xamarin.Android 8.1.

AndroidUseSharedRuntime

A boolean property that determines whether the *shared runtime packages* are required in order to run the Application on the target device. Relying on the shared runtime packages allows the Application package to be smaller, speeding up the package creation and deployment process, resulting in a faster build/deploy/debug turnaround cycle.

Prior to Xamarin.Android 11.2, this property should be `True` for Debug builds, and `False` for Release projects.

This property was *removed* in Xamarin.Android 11.2.

AndroidVersionCodePattern

A string property which allows the developer to customize the `versionCode` in the manifest. See Creating the Version Code for the APK for information on deciding a `versionCode`.

ome examples, if `abi` is `armeabi` and `versionCode` in the manifest s `123`, `{abi}{versionCode}` ill produce a versionCode of `1123` when `$(AndroidCreatePackagePerAbi)` s True, otherwise will produce a value of 123. f `abi` is `x86_64` and `versionCode` in the manifest s `44`. This will produce `544` when `$(AndroidCreatePackagePerAbi)` s True, otherwise will produce a value of `44`.

If we include a left padding format string `{abi}{versionCode:0000}`, it would produce `50044` because we are left padding the `versionCode` with `0`. Alternatively, you can use the decimal padding such as `{abi}{versionCode:D4}` which does the same as the previous example.

Only '0' and 'Dx' padding format strings are supported since the value MUST be an integer.

Pre-defined key items

- **abi** – Inserts the targeted abi for the app
  - 2 – `armeabi-v7a`
  - 3 – `x86`
  - 4 – `arm64-v8a`
  - 5 – `x86_64`
- **minSDK** – Inserts the minimum supported Sdk value from the `AndroidManifest.xml` or `11` if none is defined.
- **versionCode** – Uses the version code directly from `Properties\AndroidManifest.xml`.

You can define custom items using the `$(AndroidVersionCodeProperties)` property (defined next).

By default the value will be set to `{abi}{versionCode:D6}`. If a developer wants to keep the old behavior you can override the default by setting the `$(AndroidUseLegacyVersionCode)` property to `true`

Added in Xamarin.Android 7.2.

AndroidVersionCodeProperties

A string property which allows the developer to define custom items to use with the `$(AndroidVersionCodePattern)`. They are in the form of a `key=value` pair. All items in the `value` should be integer values. For example: `screen=23;target=$(_AndroidApiLevel)`. As you can see you can make use of existing or custom MSBuild properties in the string.

Added in Xamarin.Android 7.2.

AotAssemblies

A boolean property that determines whether or not assemblies will be Ahead-of-Time compiled into native code and included in the `.apk`.

Support for this property was added in Xamarin.Android 5.1.

This property is `False` by default.

AProfUtilExtraOptions

Extra options to pass to `aprofutil`.

BeforeGenerateAndroidManifest

MSBuild Targets listed in this property will run directly before `_GenerateJavaStubs`.

Added in Xamarin.Android 9.4.

Configuration

Specifies the build configuration to use, such as "Debug" or "Release". The Configuration property is used to determine default values for other properties which determine target behavior. Additional configurations may be created within your IDE.

By default, the `Debug` configuration will result in the `Install` and `SignAndroidPackage` targets creating a smaller Android package which requires the presence of other files and packages to operate.

The default `Release` configuration will result in the `Install` and `SignAndroidPackage` targets creating an Android package which is *stand-alone*, and may be used without installing any other packages or files.

DebugSymbols

A boolean value which determines whether the Android package is *debuggable*, in combination with the `$(DebugType)` property. A debuggable package contains debug symbols, sets the `//application/@android:debuggable` attribute to `true`, and automatically adds the `INTERNET` permission so that a debugger can attach to the process. An application is debuggable if `DebugSymbols` is `True` *and* `DebugType` is either the empty string or `Full`.

DebugType

Specifies the type of debug symbols to generate as part of the build, which also impacts whether the Application is debuggable. Possible values include:

- **Full**: Full symbols are generated. If the `DebugSymbols` MSBuild property is also `True`, then the Application package is debuggable.
- **PdbOnly**: "PDB" symbols are generated. The Application package is not debuggable.

If `DebugType` is not set or is the empty string, then the `DebugSymbols` property controls whether or not the Application is debuggable.

EmbedAssembliesIntoApk

A boolean property that determines whether or not the app's assemblies should be embedded into the Application package.

This property should be `True` for Release builds and `False` for Debug builds. It *may* need to be `True` in Debug builds if Fast Deployment doesn't support the target device.

When this property is `False`, then the `$(AndroidFastDeploymentType)` MSBuild property also controls what will be embedded into the `.apk`, which can impact deployment and rebuild times.

EnableLLVM

A boolean property that determines whether or not LLVM will be used when Ahead-of-Time compiling assemblies into native code.

The Android NDK must be installed to build a project that has this property enabled.

Support for this property was added in Xamarin.Android 5.1.

This property is `False` by default.

This property is ignored unless the `$(AotAssemblies)` MSBuild property is `True`.

EnableProguard

A boolean property that determines whether or not proguard is run as part of the packaging process to link Java code.

Support for this property was added in Xamarin.Android 5.1.

This property is `False` by default.

When `True`, @(ProguardConfiguration) files will be used to control `proguard` execution.

JavaMaximumHeapSize

Specifies the value of the **java** `-Xmx` parameter value to use when building the `.dex` file as part of the packaging process. If not specified, then the `-Xmx` option supplies **java** with a value of `1G`. This was found to be commonly required on Windows in comparison to other platforms.

Specifying this property is necessary if the `_CompileDex` target throws a `java.lang.OutOfMemoryError`.

Customize the value by changing:

```
<JavaMaximumHeapSize>1G</JavaMaximumHeapSize>
```

JavaOptions

Specifies additional command-line options to pass to **java** when building the `.dex` file.

JarsignerTimestampAuthorityCertificateAlias

This property allows you to specify an alias in the keystore for a timestamp authority. See the Java Signature Timestamp Support documentation for more details.

```
<PropertyGroup>
    <JarsignerTimestampAuthorityCertificateAlias>Alias</JarsignerTimestampAuthorityCertificateAlias>
</PropertyGroup>
```

JarsignerTimestampAuthorityUrl

This property allows you to specify a URL to a timestamp authority service. This can be used to make sure your `.apk` signature includes a timestamp. See the Java Signature Timestamp Support documentation for more details.

```
<PropertyGroup>
    <JarsignerTimestampAuthorityUrl>http://example.tsa.url</JarsignerTimestampAuthorityUrl>
</PropertyGroup>
```

LinkerDumpDependencies

A bool property which enables generating of linker dependencies file. This file can be used as input for illinkanalyzer tool.

The dependencies file named `linker-dependencies.xml.gz` is written to the project directory. On .NET5/6 it is written next to the linked assemblies in `obj/<Configuration>/android<ABI>/linked` directory.

The default value is False.

MandroidI18n

Specifies the internationalization support included with the Application, such as collation and sorting tables. The value is a comma- or semicolon-separated list of one or more of the following case-insensitive values:

- **None**: Include no additional encodings.
- **All**: Include all available encodings.
- **CJK**: Include Chinese, Japanese, and Korean encodings such as *Japanese (EUC)* [enc-jp, CP51932], *Japanese (Shift-JIS)* [iso-2022-jp, shift_jis, CP932], *Japanese (JIS)* [CP50220], *Chinese Simplified (GB2312)* [gb2312, CP936], *Korean (UHC)* [ks_c_5601-1987, CP949], *Korean (EUC)* [euc-kr, CP51949], *Chinese Traditional (Big5)* [big5, CP950], and *Chinese Simplified (GB18030)* [GB18030, CP54936].
- **MidEast**: Include Middle-Eastern encodings such as *Turkish (Windows)* [iso-8859-9, CP1254], *Hebrew (Windows)* [windows-1255, CP1255], *Arabic (Windows)* [windows-1256, CP1256], *Arabic (ISO)* [iso-8859-6, CP28596], *Hebrew (ISO)* [iso-8859-8, CP28598], *Latin 5 (ISO)* [iso-8859-9, CP28599], and *Hebrew (Iso Alternative)* [iso-8859-8, CP38598].
- **Other**: Include Other encodings such as *Cyrillic (Windows)* [CP1251], *Baltic (Windows)* [iso-8859-4, CP1257], *Vietnamese (Windows)* [CP1258], *Cyrillic (KOI8-R)* [koi8-r, CP1251], *Ukrainian (KOI8-U)* [koi8-u, CP1251], *Baltic (ISO)* [iso-8859-4, CP1257], *Cyrillic (ISO)* [iso-8859-5, CP1251], *ISCII Davenagari* [x-iscii-de, CP57002], *ISCII Bengali* [x-iscii-be, CP57003], *ISCII Tamil* [x-iscii-ta, CP57004], *ISCII Telugu* [x-iscii-te, CP57005], *ISCII Assamese* [x-iscii-as, CP57006], *ISCII Oriya* [x-iscii-or, CP57007], *ISCII Kannada* [x-iscii-ka, CP57008], *ISCII Malayalam* [x-iscii-ma, CP57009], *ISCII Gujarati* [x-iscii-gu, CP57010], *ISCII Punjabi* [x-iscii-pa, CP57011], and *Thai (Windows)* [CP874].
- **Rare**: Include Rare encodings such as *IBM EBCDIC (Turkish)* [CP1026], *IBM EBCDIC (Open Systems Latin 1)* [CP1047], *IBM EBCDIC (US-Canada with Euro)* [CP1140], *IBM EBCDIC (Germany with Euro)* [CP1141], *IBM EBCDIC (Denmark/Norway with Euro)* [CP1142], *IBM EBCDIC (Finland/Sweden with Euro)* [CP1143], *IBM EBCDIC (Italy with Euro)* [CP1144], *IBM EBCDIC (Latin America/Spain with Euro)* [CP1145], *IBM EBCDIC (United Kingdom with Euro)* [CP1146], *IBM EBCDIC (France with Euro)* [CP1147], *IBM EBCDIC (International with Euro)* [CP1148], *IBM EBCDIC (Icelandic with Euro)* [CP1149], *IBM EBCDIC (Germany)* [CP20273], *IBM EBCDIC (Denmark/Norway)* [CP20277], *IBM EBCDIC (Finland/Sweden)* [CP20278], *IBM EBCDIC (Italy)* [CP20280], *IBM EBCDIC (Latin America/Spain)* [CP20284], *IBM EBCDIC (United Kingdom)* [CP20285], *IBM EBCDIC (Japanese Katakana Extended)* [CP20290], *IBM EBCDIC (France)* [CP20297], *IBM EBCDIC (Arabic)* [CP20420], *IBM EBCDIC (Hebrew)* [CP20424], *IBM EBCDIC (Icelandic)* [CP20871], *IBM EBCDIC (Cyrillic - Serbian, Bulgarian)* [CP21025], *IBM EBCDIC (US-Canada)* [CP37], *IBM EBCDIC (International)* [CP500], *Arabic (ASMO 708)* [CP708], *Central European (DOS)* [CP852]*, Cyrillic (DOS)* [CP855], *Turkish (DOS)* [CP857], *Western European (DOS with Euro)* [CP858], *Hebrew (DOS)* [CP862], *Arabic (DOS)* [CP864], *Russian (DOS)* [CP866], *Greek (DOS)* [CP869], *IBM EBCDIC (Latin 2)* [CP870], and

IBM EBCDIC (Greek) [CP875].

- **West**: Include Western encodings such as *Western European (Mac)* [macintosh, CP10000], *Icelandic (Mac)* [x-mac-icelandic, CP10079], *Central European (Windows)* [iso-8859-2, CP1250], *Western European (Windows)* [iso-8859-1, CP1252], *Greek (Windows)* [iso-8859-7, CP1253], *Central European (ISO)* [iso-8859-2, CP28592], *Latin 3 (ISO)* [iso-8859-3, CP28593], *Greek (ISO)* [iso-8859-7, CP28597], *Latin 9 (ISO)* [iso-8859-15, CP28605], *OEM United States* [CP437], *Western European (DOS)* [CP850], *Portuguese (DOS)* [CP860], *Icelandic (DOS)* [CP861], *French Canadian (DOS)* [CP863], and *Nordic (DOS)* [CP865].

```
<MandroidI18n>West</MandroidI18n>
```

MonoAndroidResourcePrefix

Specifies a *path prefix* that is removed from the start of filenames with a Build action of `AndroidResource`. This is to allow changing where resources are located.

The default value is `Resources`. Change this to `res` for the Java project structure.

MonoSymbolArchive

A boolean property which controls whether `.mSYM` artifacts are created for later use with `mono-symbolicate`, to extract "real" filename and line number information from Release stack traces.

This is True by default for "Release" apps which have debugging symbols enabled: `$(EmbedAssembliesIntoApk)` is True, `$(DebugSymbols)` is True, and `$(Optimize)` is True.

Added in Xamarin.Android 7.1.

Build Targets

11/2/2020 • 2 minutes to read • Edit Online

The following build targets are defined for Xamarin.Android projects.

Build

Builds the source code within a project and all dependencies.

This target *does not* create an Android package (`.apk` file). To create an Android package, use the SignAndroidPackage target, *or* set the `$(AndroidBuildApplicationPackage) property to True when building:

```
msbuild /p:AndroidBuildApplicationPackage=True App.sln
```

BuildAndStartAotProfiling

Builds the app with an embedded AOT profiler, sets the profiler TCP port to `$(AndroidAotProfilerPort)`, and starts the default activity.

The default TCP port is `9999`.

Added in Xamarin.Android 10.2.

Clean

Removes all files generated by the build process.

FinishAotProfiling

Must be called *after* the BuildAndStartAotProfiling target.

Collects the AOT profiler data from the device or emulator through the TCP port `$(AndroidAotProfilerPort)` and writes them to `$(AndroidAotCustomProfilePath)`.

The default values for port and custom profile are `9999` and `custom.aprof`.

To pass additional options to `aprofutil`, set them in the `$(AProfUtilExtraOptions)` property.

This is equivalent to:

```
aprofutil $(AProfUtilExtraOptions) -s -v -f -p $(AndroidAotProfilerPort) -o "$(AndroidAotCustomProfilePath)"
```

Added in Xamarin.Android 10.2.

Install

Creates, signs, and installs the Android package onto the default device or virtual device.

The `$(AdbTarget)` property specifies the Android target device the Android package may be installed to or removed from.

```
# Install package onto emulator via -e
# Use `/Library/Frameworks/Mono.framework/Commands/msbuild` on OS X
MSBuild /t:Install ProjectName.csproj /p:AdbTarget=-e
```

SignAndroidPackage

Creates and signs the Android package (`.apk`) file.

Use with `/p:Configuration=Release` to generate self-contained "Release" packages.

StartAndroidActivity

Starts the default activity on the device or the running emulator.

To start a different activity, set the `$(AndroidLaunchActivity)` property to the activity name.

This is equivalent to:

```
adb shell am start @PACKAGE_NAME@/$(AndroidLaunchActivity)
```

Added in Xamarin.Android 10.2.

StopAndroidPackage

Completely stops the application package on the device or the running emulator.

This is equivalent to:

```
adb shell am force-stop @PACKAGE_NAME@
```

Added in Xamarin.Android 10.2.

Uninstall

Uninstalls the Android package from the default device or virtual device.

The `$(AdbTarget)` property specifies the Android target device the Android package may be installed to or removed from.

UpdateAndroidResources

Updates the `Resource.designer.cs` file.

This target is usually called by the IDE when new resources are added to the project.

Building ABI-Specific APKs

7/8/2021 • 7 minutes to read • Edit Online

This document will discuss how to build an APK that will target a single ABI using Xamarin.Android.

Overview

In some situations it may be advantageous for an application to have multiple APKs - each APK is signed with the same keystore and shares the same package name but it is compiled for a specific device or Android configuration. This is not the recommended approach - it is much simpler to have one APK that can support multiple devices and configurations. There are some situations where creating multiple APKs can be useful, such as:

- **Reduce the size of the APK** - Google Play imposes a 100MB size limit on APK files. Creating device specific APK's can reduce the size of the APK as you only need to supply a subset of assets and resources for the application.
- **Support different CPU architectures** - If your application has shared libraries for specific CPU's, you can distribute only the shared libraries for that CPU.

Multiple APKs can complicate distribution - a problem that is addressed by Google Play. Google Play will ensure that the correct APK is delivered to a device based on the application's version code and other metadata contained with **AndroidManifest.XML**. For specific details and restrictions on how Google Play supports multiple APKs for an application, consult Google's documentation on multiple APK support.

This guide will address how to script the building multiple APKs for a Xamarin.Android application, each APK targeting a specific ABI. It will cover the following topics:

1. Create a unique *version code* for the APK.
2. Create a temporary version of **AndroidManifest.XML** that will be used for this APK.
3. Build the application using the **AndroidManifest.XML** from the previous step.
4. Prepare the APK for release by signing and zip-aligning it.

At the end of this guide is a walkthrough that will demonstrate how to script these steps using Rake.

Creating the Version Code for the APK

Google recommends a particular algorithm for the version code that uses a seven digit version code (please see the section *Using a version code scheme* in the Multiple APK support document). By expanding this version code scheme to eight digits, it is possible to include some ABI information into the version code that will ensure that Google Play will distribute the correct APK to a device. The following list explains this eight digit version code format (indexed from left to right):

- **Index 0** (red in diagram below) – An integer for the ABI:
  - 1 – `armeabi`
  - 2 – `armeabi-v7a`
  - 6 – `x86`
- **Index 1-2** (orange in diagram below) – The minimum API level supported by the application.
- **Index 3-4** (blue in diagram below) – The screen sizes supported:
  - 1 – small

- 2 – normal
- 3 – large
- 4 – xlarge

- **Index 5-7** (green in diagram below) – A unique number for the version code. This is set by the developer. It should increase for each public release of the application.

The following diagram illustrates the position of each code described in the above list:

Google Play will ensure that the correct APK is delivered to the device based on the `versionCode` and APK configuration. The APK with the highest version code will be delivered to the device. As an example, an application could have three APKs with the following version codes:

- 11413456 - The ABI is `armeabi` ; targeting API level 14; small to large screens; with a version number of 456.
- 21423456 - The ABI is `armeabi-v7a` ; targeting API level 14; normal & large screens; with a version number of 456.
- 61423456 - The ABI is `x86` ; targeting API level 14; normal & large screens; with a version number of 456.

To continue on with this example, imagine that a bug was fixed which was specific to `armeabi-v7a`. The app version increases to 457, and an new APK is built with the `android:versionCode` set to 21423457. The versionCodes for the `armeabi` and `x86` versions would remain the same.

Now imagine that the x86 version receives some updates or bug fixes that target a newer API (API level 19), making this version 500 of the app. The new `versionCode` would change to 61923500 while the armeabi/armeabi-v7a remain unchanged. At this point in time, the version codes would be:

- 11413456 - The ABI is `armeabi` ; targeting API level 14; small to large screens; with a version name of 456.
- 21423457 - The ABI is `armeabi-v7a` ; targeting API level 14; normal & large screens; with a version name of 457.
- 61923500 - The ABI is `x86` ; targeting API level 19; normal & large screens; with a version name of 500.

Maintaining these version codes manually can be a significant burden on the developer. The process of calculating the correct `android:versionCode` and then building the APK's should be automated. An example of how to do so will be covered in the walkthrough at the end of this document.

Create A Temporary AndroidManifest.XML

Although not strictly necessary, creating an temporary **AndroidManifest.XML** for each ABI can help prevent issues that might arise with information leaking from one APK to the other. For example, it is crucial that the `android:versionCode` attribute is unique for each APK.

How this is done depends on the scripting system involved, but typically involves taking a copy of the Android manifest used during development, modifying it, and then using that modify manifest during the build process.

Compiling the APK

Building the APK per ABI is best accomplished by using either `xbuild` or `msbuild` as shown in the following sample command line:

```
/Library/Frameworks/Mono.framework/Commands/xbuild /t:Package /p:AndroidSupportedAbis=<TARGET_ABI>
/p:IntermediateOutputPath=obj.<TARGET_ABI>/ /p:AndroidManifest=<PATH_TO_ANDROIDMANIFEST.XML>
/p:OutputPath=bin.<TARGET_ABI> /p:Configuration=Release <CSPROJ FILE>
```

The following list explains each command line parameter:

- `/t:Package` – Creates an Android APK that is signed using the debug keystore
- `/p:AndroidSupportedAbis=<TARGET_ABI>` – This the ABI to target. Must one of `armeabi`, `armeabi-v7a`, or `x86`
- `/p:IntermediateOutputPath=obj.<TARGET_ABI>/` – This is the directory that will hold the intermediate files that are created as a part of the build. If necessary, Xamarin.Android will create a directory named after the ABI, such as `obj.armeabi-v7a`. It is recommended to use one folder for each ABI as this will prevent issues that make result with files "leaking" from one build to another. Notice that this value is terminated with a directory separator (a `/` in the case of OS X).
- `/p:AndroidManifest` – This property specifies the path to the **AndroidManifest.XML** file that will be used during the build.
- `/p:OutputPath=bin.<TARGET_ABI>` – This is the directory that will house the final APK. Xamarin.Android will create a directory named after the ABI, for example `bin.armeabi-v7a`.
- `/p:Configuration=Release` – Perform a Release build of the APK. Debug builds may not be uploaded to Google Play.
- `<CS_PROJ FILE>` – This is the path to the `.csproj` file for the Xamarin.Android project.

Sign and Zipalign The APK

It is necessary to sign the APK before it can be distributed via Google Play. This can be performed by using the `jarsigner` application that is a part of the Java Developer's Kit. The following command line demonstrats how to use `jarsigner` at the command line:

```
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore <PATH/TO/KEYSTORE> -storepass <PASSWORD> -signedjar <PATH/FOR/SIGNED_JAR> <PATH/FOR/JAR/TO/SIGN> <NAME_OF_KEY_IN_KEYSTORE>
```

All Xamarin.Android applications must be zip-aligned before they can be run on a device. This is the format of the command line to use:

```
zipalign -f -v 4 <SIGNED_APK_TO_ZIPALIGN> <PATH/TO/ZIP_ALIGNED.APK>
```

Automating APK Creation With Rake

The sample project OneABIPerAPK is a simple Android project that will demonstrate how to calculate an ABI specific version number and build three separate APK's for each of the following ABI's:

- armeabi
- armeabi-v7a
- x86

The rakefile in the sample project performs each of the steps that were described in the previous sections:

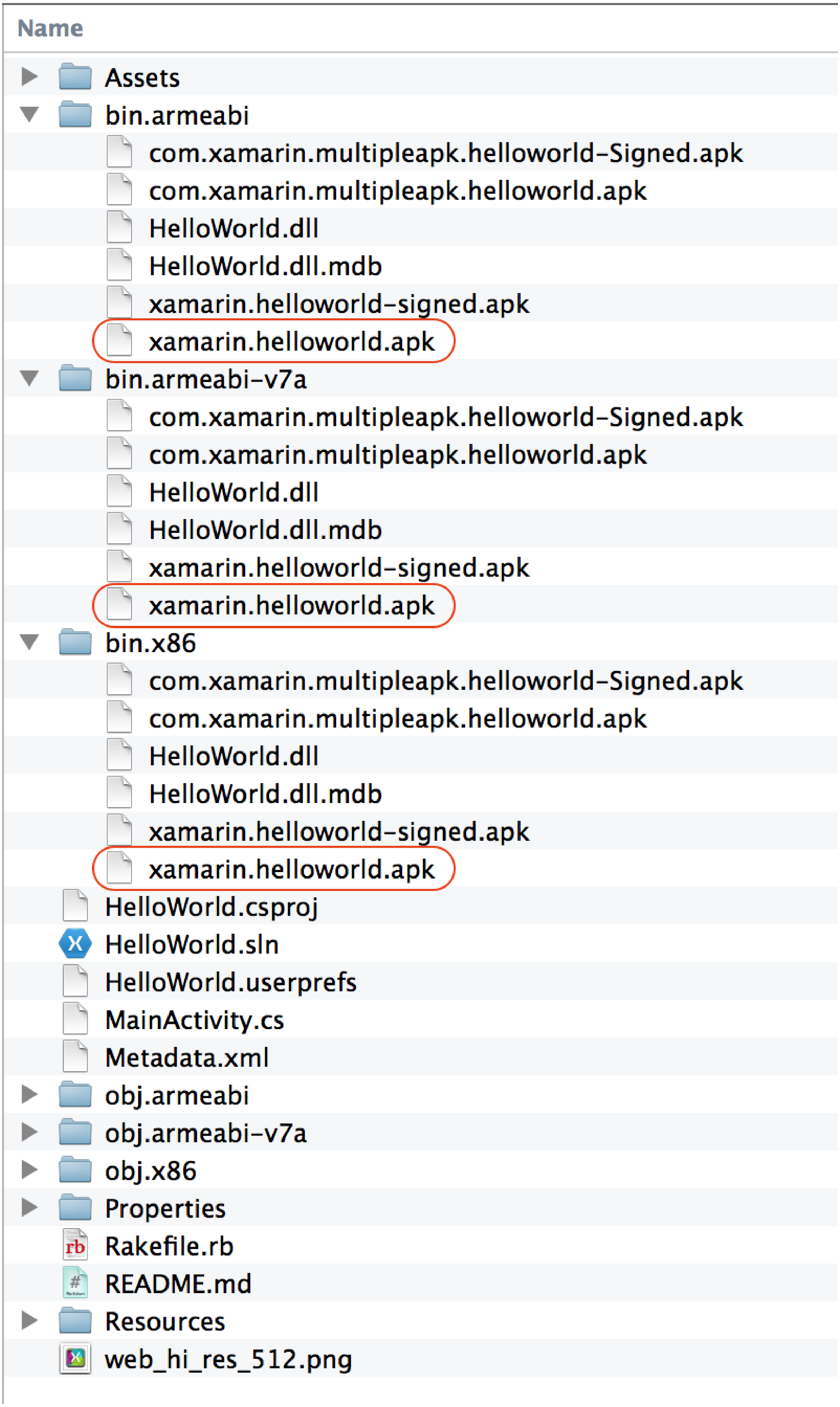
1. Create an android:versionCode for the APK.
2. Write the android:versionCode to a custom **AndroidManifest.XML** for that APK.
3. Compile a release build of the Xamarin.Android project that will singularly target the ABI and using the **AndroidManifest.XML** that was created in the previous step.
4. Sign the APK with a production keystore.

5. Zipalign the APK.

To build all of the APKs for the application, run the `build` Rake task from the command line:

```
$ rake build
==> Building an APK for ABI armeabi with ./Properties/AndroidManifest.xml.armeabi, android:versionCode =
10814120.
==> Building an APK for ABI x86 with ./Properties/AndroidManifest.xml.x86, android:versionCode = 60814120.
==> Building an APK for ABI armeabi-v7a with ./Properties/AndroidManifest.xml.armeabi-v7a,
android:versionCode = 20814120.
```

Once the rake task has completed, there will be three `bin` folders with the file `xamarin.helloworld.apk`. The next screenshot shows each of these folders with their contents:


Name
Assets
bin.armeabi
com.xamarin.multipleapk.helloworld-Signed.apk
com.xamarin.multipleapk.helloworld.apk
HelloWorld.dll
HelloWorld.dll.mdb
xamarin.helloworld-signed.apk
xamarin.helloworld.apk
bin.armeabi-v7a
com.xamarin.multipleapk.helloworld-Signed.apk
com.xamarin.multipleapk.helloworld.apk
HelloWorld.dll
HelloWorld.dll.mdb
xamarin.helloworld-signed.apk
xamarin.helloworld.apk
bin.x86
com.xamarin.multipleapk.helloworld-Signed.apk
com.xamarin.multipleapk.helloworld.apk
HelloWorld.dll
HelloWorld.dll.mdb
xamarin.helloworld-signed.apk
xamarin.helloworld.apk
HelloWorld.csproj
HelloWorld.sln
HelloWorld.userprefs
MainActivity.cs
Metadata.xml
obj.armeabi
obj.armeabi-v7a
obj.x86
Properties
Rakefile.rb
README.md
Resources
web_hi_res_512.png

> **NOTE**
>
> The build process outlined in this guide may be implemented in one of many different build systems. Although we don't have a pre-written example, it should also be possible with Powershell / psake or Fake.

Summary

This guide provided some suggestions with how to create Android APK's that target a specify ABI. It also discussed one possible scheme for creating `android:versionCodes` that will identify the CPU architecture that the APK is intended for. The walkthrough included a sample project that has it's build scripted using Rake.

Related Links

- OneABIPerAPK (sample)
- Publishing an Application
- Multiple APK Support for Google Play

Command Line Emulator

10/28/2019 • 2 minutes to read • Edit Online

Running the Android emulator from the command line

To enable running the Android emulator from the command line, you can use the "emulator" tool provided by the Android SDK. This tool can be used to run the emulator from Terminal on OS X or from Command Prompt on a Windows machine.

To launch a specific Android emulator, run the following command from the tools directory in the android SDK location (such as C:\android-sdk-windows\tools):

On Windows

```
emulator.exe -avd NameOfYourEmulator -partition-size 512
```

On macOS

```
./emulator -avd NameOfYourEmulator -partition-size 512
```

The reason for needing the partition size is to allow the emulator to have plenty of space to get the Xamarin.Android platform installed on the emulator as by default the size of the emulator is small.

You can find out more information on extra parameters on the Android site here - https://developer.android.com/studio/run/emulator-commandline

Debug Xamarin.Android apps

12/31/2019 • 2 minutes to read • Edit Online

This section discusses how to debug a Xamarin.Android app on devices or emulators.

Debugging Overview

Developing Android applications requires running the application, either on physical hardware or using an emulator. Using hardware is the best approach, but not always the most practical. In many cases, it can be simpler and more cost effective to simulate/emulate Android hardware using one of the emulators described below.

Debugging on the Android Emulator

This article explains how launch the Android emulator from Visual Studio and run your app in a virtual device.

Debugging on a Device

This article shows how to configure a physical Android device so that Xamarin.Android application can be deployed to it directly from either Visual Studio or Visual Studio for Mac.

Android Debug Log

One very common trick developers use to debug their applications is using `Console.WriteLine`. However, on a mobile platform like Android there is no console. Android devices provides a log that you will likely need to utilize while writing apps. This is sometimes referred to as **logcat** due to the command typed to retrieve it. This article describes how to use **logcat**.

Debug on the Android Emulator

7/8/2021 • 4 minutes to read • Edit Online

In this guide, you will learn how to launch a virtual device in the Android Emulator to debug and test your app.

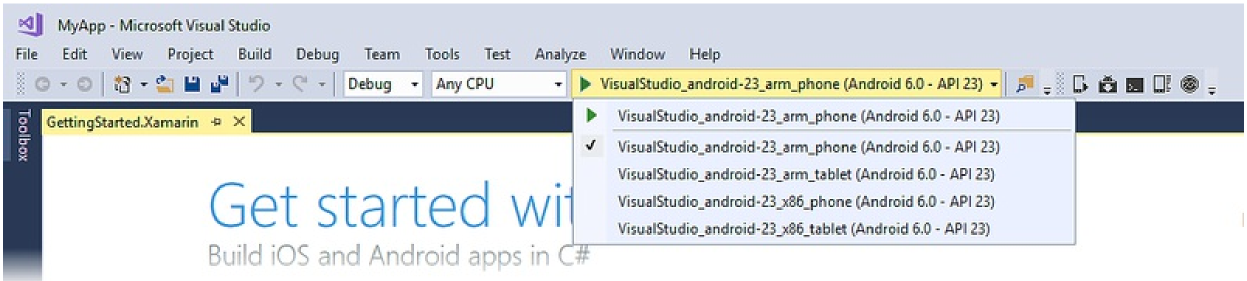
The Android Emulator (installed as part of the **Mobile development with .NET** workload), can be run in a variety of configurations to simulate different Android devices. Each one of these configurations is created as a *virtual device*. In this guide, you will learn how to launch the emulator from Visual Studio and run your app in a virtual device. For information about configuring the Android Emulator and creating new virtual devices, see Android Emulator Setup.

Using a Pre-Configured Virtual Device

- Visual Studio
- Visual Studio for Mac

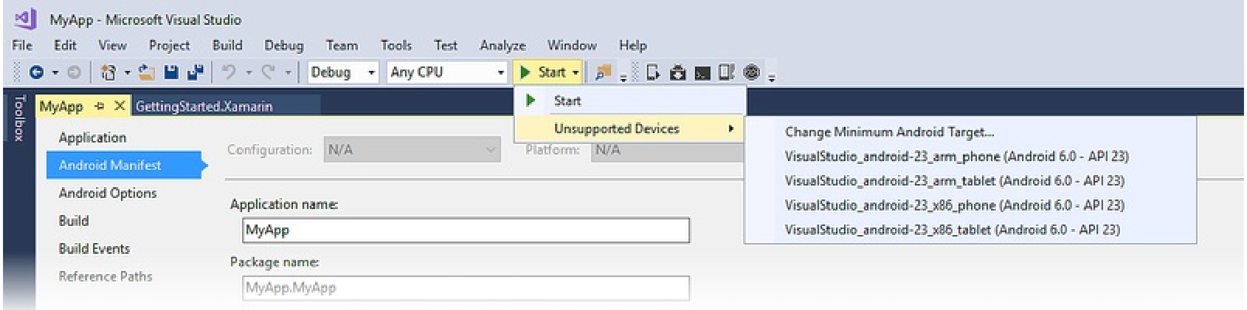
Visual Studio includes pre-configured virtual devices that appear in the device drop-down menu. For example, in the following Visual Studio 2017 screenshot, several pre-configured virtual devices are available:

- **VisualStudio_android-23_arm_phone**
- **VisualStudio_android-23_arm_tablet**
- **VisualStudio_android-23_x86_phone**
- **VisualStudio_android-23_x86_tablet**



Typically, you would select the **VisualStudio_android-23_x86_phone** virtual device to test and debug a phone app. If one of these pre-configured virtual devices meets your requirements (i.e., matches your app's target API level), skip to Launching the Emulator to begin running your app in the emulator. (If you are not yet familiar with Android API levels, see Understanding Android API Levels.)

If your Xamarin.Android project is using a Target Framework level that is incompatible with the available virtual devices, the drop-down menu lists the unusable virtual devices under **Unsupported Devices**. For example, the following project has a Target Framework set to **Android 7.1 Nougat (API 25)**, which is incompatible with the **Android 6.0** virtual devices that are listed in this example:



You can click **Change Minimum Android Target** to change the project's Minimum Android Version so that it matches the API level of the available virtual devices. Alternately, you can use the Android Device Manager to create new virtual devices that support your target API level. Before you can configure virtual devices for a new API level, you must first install the corresponding system images for that API level (see Setting up the Android SDK for Xamarin.Android).

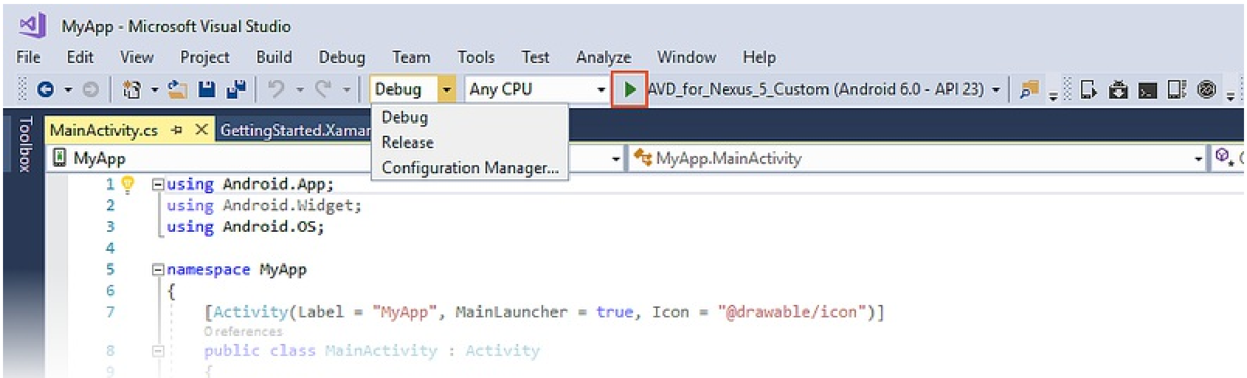
Editing Virtual Devices

To modify virtual devices (or to create new ones), you must use the Android Device Manager.

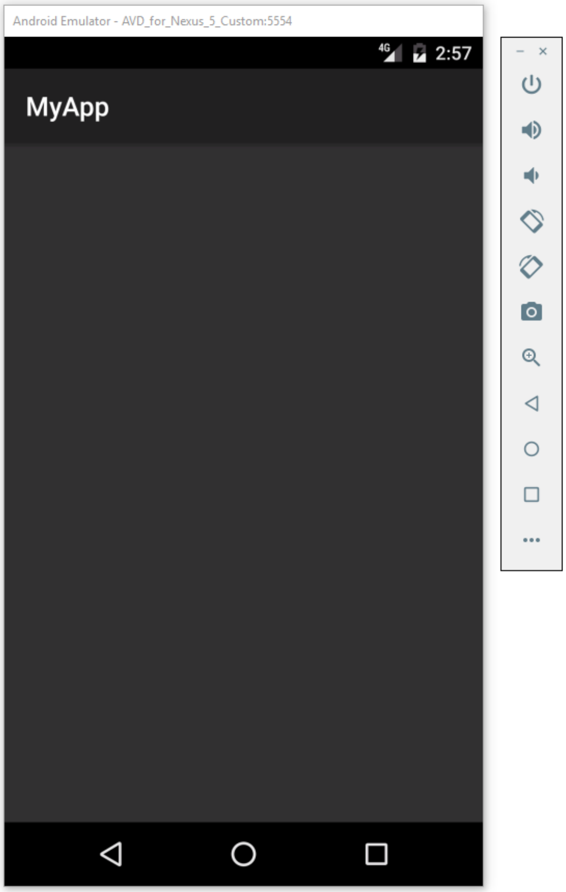
Launching the Emulator

Near the top of Visual Studio, there is a drop-down menu that can be used to select **Debug** or **Release** mode. Choosing **Debug** causes the debugger to attach to the application process running inside the emulator after the app starts. Choosing **Release** mode disables the debugger (however, you can still run the app and use log statements for debug). After you have chosen a virtual device from the device drop-down menu, select either **Debug** or **Release** mode, then click the Play button to run the application:

- Visual Studio
- Visual Studio for Mac



After the emulator starts, Xamarin.Android will deploy the app to the emulator. The emulator runs the app with the configured virtual device image. An example screenshot of the Android Emulator is displayed below. In this example, the emulator is running a blank app called **MyApp**:



The emulator may be left running: it is not necessary to shut it down and wait for it to restart each time the app is launched. The first time a Xamarin.Android app is run in the emulator, the Xamarin.Android shared runtime for the targeted API level is installed, followed by the application. The runtime installation may take a few moments, so please be patient. Installation of the runtime takes place only when the first Xamarin.Android app is deployed to the emulator – subsequent deployments are faster because only the app is copied to the emulator.

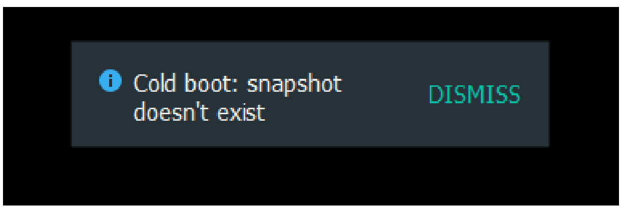
Quick Boot

Newer versions of the Android Emulator include a feature called *Quick Boot* that launches the emulator in only a few seconds. When you close the emulator, it takes a snapshot of the virtual device state so that it can be quickly restored from that state when it is restarted. To access this feature, you will need the following:

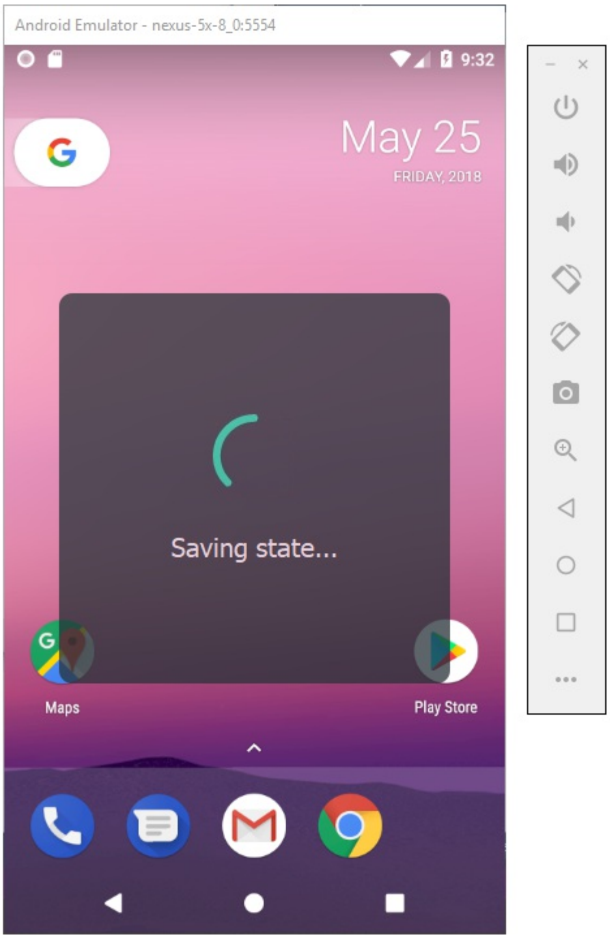
- Android Emulator version 27.0.2 or later
- Android SDK Tools version 26.1.1 or later

When the above-listed versions of the emulator and SDK tools are installed, the Quick Boot feature is enabled by default.

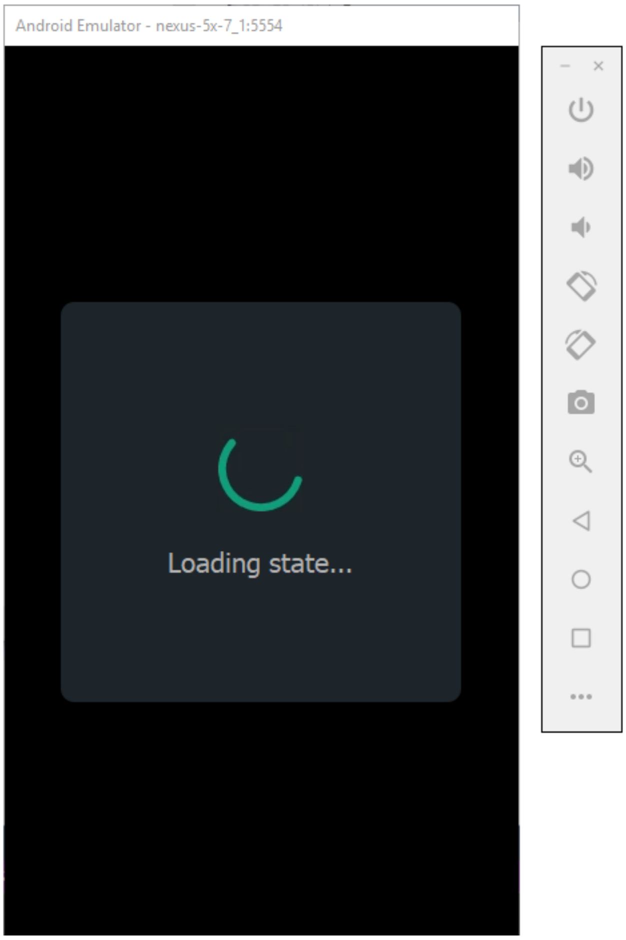
The first cold boot of the virtual device takes place without a speed improvement because a snapshot has not yet been created:



When you exit out of the emulator, Quick Boot saves the state of the emulator in a snapshot:



Subsequent virtual device starts are much faster because the emulator simply restores the state at which you closed the emulator.



Troubleshooting

For tips and workarounds for common emulator problems, see Android Emulator Troubleshooting.

Summary

This guide explained the process for configuring the Android Emulator to run and test Xamarin.Android apps. It described the steps for launching the emulator using pre-configured virtual devices, and it provided the steps for deploying an application to the emulator from Visual Studio.

For more information about using the Android Emulator, see the following Android Developer topics:

- Navigating on the Screen
- Performing Basic Tasks in the Emulator
- Working with Extended Controls, Settings, and Help
- Run the emulator with Quick Boot

Debug on an Android device

7/8/2021 • 2 minutes to read • Edit Online

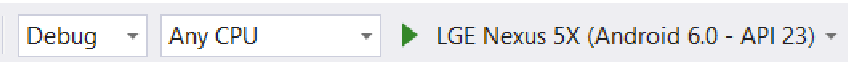
This article explains how to debug a Xamarin.Android application on a physical Android device.

It is possible to debug a Xamarin.Android app on an Android device using either Visual Studio for Mac or Visual Studio. Before debugging can occur on a device, it must be setup for development and connected to your PC or Mac.

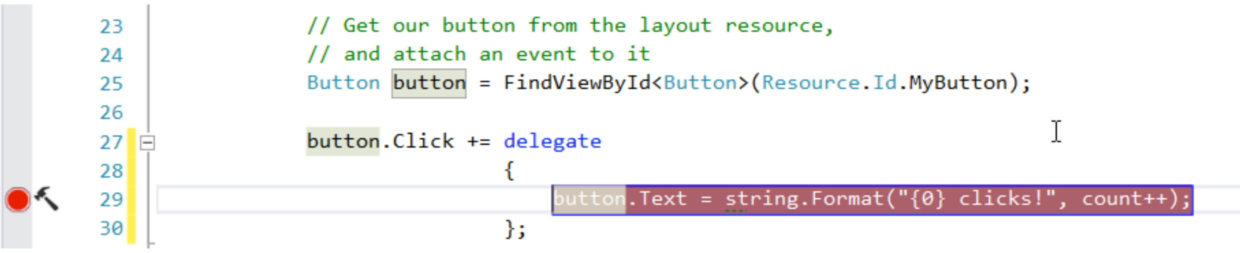
Debug Application

Once a device is connected to your computer, debugging a Xamarin.Android application is done in the same way as any other Xamarin product or .NET application. Ensure that the **Debug** configuration and the external device is selected in the IDE, this will ensure that the necessary debug symbols are available and that the IDE can connect to the running application:

- Visual Studio
- Visual Studio for Mac



Next, a breakpoint is set in the code:



Once the device has been selected, Xamarin.Android will connect to the device, deploy the application, and then run it. When the breakpoint is reached, the debugger will stop the application, allowing the application to be debugged in a fashion similar to any other C# application:

```
[Activity(Label = "App7", MainLauncher = true, Icon = "@drawable/icon")]
public class MainActivity : Activity
{
    int count = 1;

    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);

        // Set our view from the "main" layout resource
        SetContentView(Resource.Layout.Main);

        // Get our button from the layout resource,
        // and attach an event to it
        Button button = FindViewById<Button>(Resource.Id.MyButton);

        button.Click += delegate
                        {
                            button.Text = string.Format("{0} clicks!", count++);
                        };
    }
}
```

Summary

In this document discussed how to debug a Xamarin.Android application by setting a breakpoint and selecting the target device.

Related Links

- Set Up Device for Development
- Setting the Debuggable Attribute

Android Debug Log

7/8/2021 • 5 minutes to read • Edit Online

One very common trick developers use to debug their applications is to make calls to `Console.WriteLine`. However, on a mobile platform like Android there is no console. Android devices provides a log that you can use while writing apps. This is sometimes referred to as *logcat* due to the command that you type to retrieve it. Use the **Debug Log** tool to view the logged data.

Android Debug Log Overview

The **Debug Log** tool provides a way to view log output while debugging an app through Visual Studio. The debug log supports the following devices:

- Physical Android phones, tablets, and wearables.
- An Android Virtual device running on the Android Emulator.

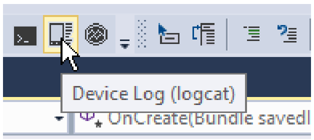
> **NOTE**
>
> The **Debug Log** tool does not work with Xamarin Live Player.

The **Debug Log** does not display log messages that are generated while the app is running standalone on the device (i.e., while it is disconnected from Visual Studio).

Accessing the Debug Log from Visual Studio

- Visual Studio
- Visual Studio for Mac

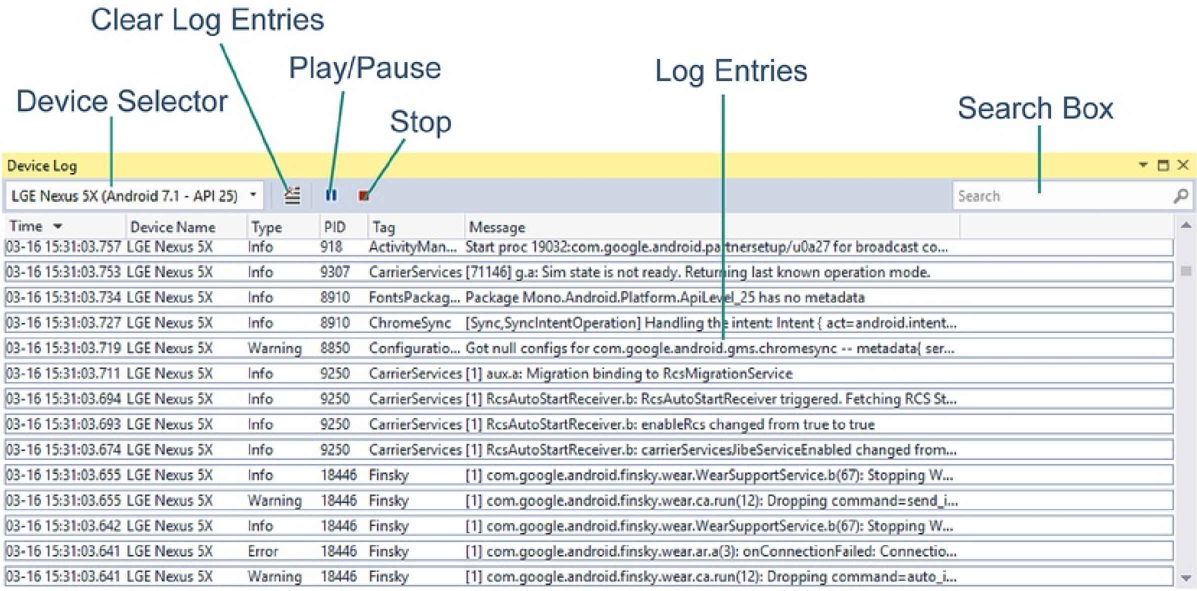
To open the **Device Log** tool, click **Device Log (logcat)** icon on the toolbar:



Alternately, launch the **Device Log** tool from one of the following menu selections:

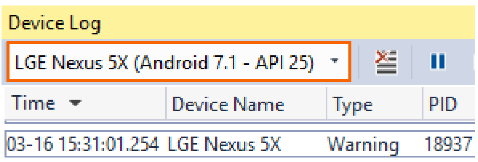
- **View > Other Windows > Device Log**
- **Tools > Android > Device Log**

The following screenshot illustrates the various parts of the **Debug Tool** window:



- **Device Selector** – Selects which physical device or running emulator to monitor.
- **Log Entries** – A table of log messages from logcat.
- **Clear Log Entries** – Clears all current log entries from the table.
- **Play/Pause** – Toggles between updating or pausing the display of new log entries.
- **Stop** – Halts the display of new log entries.
- **Search Box** – Enter search strings in this box to filter for a subset of log entries.

When the **Debug Log** tool window is displayed, use the device pull-down menu to choose the Android device to monitor:



After the device is selected, the **Device Log** tool automatically adds log entries from a running app – these log entries are shown in the table of log entries. Switching between devices stops and starts device logging. Note that an Android project must be loaded before any devices will appear in the device selector. If the device does not appear in the device selector, verify that it is available in the Visual Studio device drop-down menu next to the **Start** button.

Accessing from the Command Line

- Visual Studio
- Visual Studio for Mac

Another option is to view the debug log via the command line. Open a command prompt window and navigate to the Android SDK platform-tools folder (typically, the SDK platform-tools folder is located at **C:\Program Files (x86)\Android\android-sdk\platform-tools**).

If only a single device (physical device or emulator) is attached, the log can be viewed by entering the following command:

```
$ adb logcat
```

If more than one device is attached, the device must be explicitly identified. For example **adb -d logcat** displays the log of the only physical device connected, while **adb -e logcat** shows the log of the only emulator running.

More commands can be found by entering **adb** and reading the help messages.

Writing to the Debug Log

Messages can be written to the **Debug Log** by using the methods of the Android.Util.Log class. For example:

```
string tag = "myapp";

Log.Info (tag, "this is an info message");
Log.Warn (tag, "this is a warning message");
Log.Error (tag, "this is an error message");
```

This produces output similar to the following:

```
I/myapp   (11103): this is an info message
W/myapp   (11103): this is a warning message
E/myapp   (11103): this is an error message
```

It is also possible to use `Console.WriteLine` to write to the **Debug Log** – these messages appear in logcat with a slightly different output format (this technique is particularly useful when debugging Xamarin.Forms apps on Android):

```
System.Console.WriteLine ("DEBUG - Button Clicked!");
```

This produces output similar to the following in logcat:

```
Info (19543) / mono-stdout: DEBUG - Button Clicked!
```

Interesting Messages

When reading the log (and especially when providing log snippets to others), perusing the log file in its entirety is often too cumbersome. To make it easier to navigate through log messages, start by looking for a log entry that resembles the following:

```
I/ActivityManager(12944): Starting: Intent { act=android.intent.action.MAIN cat=
[android.intent.category.LAUNCHER] flg=0x10200000 cmp=GcTest.GcTest/gctest.Activity1 } from pid 24175
```

In particular, look for a line matching the regular expression that also contains the name of the application package:

```
^I.*ActivityManager.*Starting: Intent
```

This is the line which corresponds to the start of an activity, and *most* (but not all) of the following messages should relate to the application.

Notice that every message contains the process identifier (pid) of the process generating the message. In the above `ActivityManager` message, process `12944` generated the message. To determine which process is the process of the application being debugged, look for the **mono.MonoRuntimeProvider** message:

```
I/ActivityThread(  602): Pub TouchTest.TouchTest.__mono_init__: mono.MonoRuntimeProvider
```

This message comes from the process that was started. All subsequent messages that contain this pid come from the same process.

Debuggable Attribute

12/13/2019 • 2 minutes to read • Edit Online

To make debugging possible, Android supports the Java Debug Wire Protocol (JDWP). This is a technology that allows tools such as ADB to communicate with a JVM. While JDWP is important during development, it should be disabled prior to the application being published.

JDWP can be configured by the value of the `android:debuggable` attribute in an Android application. Choose *one* of the following three ways to set this attribute in Xamarin.Android:

AndroidManifest.xml

Create or open `AndroidManifest.xml` file, and set the `android:debuggable` attribute there. Take extra care not to ship your release build with debugging enabled.

Add an Application class attribute

If your Xamarin.Android app has a class with an `[Application]` attribute, update the attribute to `[Application(Debuggable = true)]`. Set it to `false` to disable.

Add an assembly attribute

If your Xamarin.Android app does NOT already have an `[Application]` class attribute, add an assembly-level attribute `[assembly: Application(Debuggable=true)]` in a c# file. Set it to `false` to disable.

Summary

If both the `AndroidManifest.xml` and the `ApplicationAttribute` are present, the contents of `AndroidManifest.xml` take priority over what is specified by the `ApplicationAttribute`.

If you add both a class attribute *and* an assembly attribute, there will be a compiler error:

```
"Error The "GenerateJavaStubs" task failed unexpectedly.
System.InvalidOperationException: Application cannot have both a type with an [Application] attribute and an
[assembly:Application] attribute."
```

By default – if neither the `AndroidManifest.xml` nor the `ApplicationAttribute` is present – the value of the `android:debuggable` attribute depends on whether or not debug symbols are generated. If debug symbols are present, then Xamarin.Android will set the `android:debuggable` attribute to `true` for you.

WARNING

The value of the `android:debuggable` attribute does NOT necessarily depend on the build configuration. It is possible for release builds to have the `android:debuggable` attribute set to true. If you use an attribute to set this value, you can choose to wrap the attribute in a compiler directive:

```
#if DEBUG
[Application(Debuggable = true)]
#else
[Application(Debuggable = false)]
#endif
```

Related Links

- Debuggable apps in the Android market

Xamarin.Android Environment

3/5/2021 • 4 minutes to read • Edit Online

Execution Environment

The *execution environment* is the set of environment variables and Android system properties that influence program execution. Android system properties can be set with the `adb shell setprop` command, while environment variables can be set by setting the `debug.mono.env` system property:

```
## Enable GREF logging
adb shell setprop debug.mono.log gref

## Set the MONO_LOG_LEVEL and MONO_LOG_MASK environment variables
## so that additional Mono messages will be written to `adb logcat`.
adb shell setprop debug.mono.env "'MONO_LOG_LEVEL=info|MONO_LOG_MASK=asm'"
```

Android system properties are set for all processes on the target device.

Starting with Xamarin.Android 4.6, both system properties and environment variables may be set or overridden on a per-app basis by adding an *environment file* to the project. An environment file is a Unix-formatted plain-text file with a **Build action of** `AndroidEnvironment`. The environment file contains lines with the format *key=value*. Comments are lines which start with `#`. Blank lines are ignored.

If *key* starts with an uppercase letter, then *key* is treated as an environment variable and **setenv**(3) is used to set the environment variable to the specified *value* during process startup.

If *key* starts with a lowercase letter, then *key* is treated as an Android system property and *value* is the *default value*. Android system properties which control Xamarin.Android execution behavior are looked up first from the Android system property store, and if no value is present then the value specified in the environment file is used. This is to permit `adb shell setprop` to be used to override values which come from the environment file for diagnostic purposes.

Xamarin.Android Environment Variables

Xamarin.Android supports the `XA_HTTP_CLIENT_HANDLER_TYPE` variable, which may be set either via `adb shell setprop debug.mono.env` or via the `$(AndroidEnvironment)` Build action.

`XA_HTTP_CLIENT_HANDLER_TYPE`

The assembly-qualified type which must inherit from HttpMessageHandler and is constructed from the `HttpClient()` default constructor.

In Xamarin.Android 6.1, this environment variable is not set by default, and HttpClientHandler will be used.

Alternatively, the value `Xamarin.Android.Net.AndroidClientHandler` may be specified to use `java.net.URLConnection` for network access, which *may* permit use of TLS 1.2 when Android supports it.

Added in Xamarin.Android 6.1.

Xamarin.Android System Properties

Xamarin.Android supports the following system properties, which may be set either via `adb shell setprop` or via the `$(AndroidEnvironment)` Build action.

- `debug.mono.debug`
- `debug.mono.env`
- `debug.mono.gc`
- `debug.mono.log`
- `debug.mono.max_grefc`
- `debug.mono.profile`
- `debug.mono.runtime_args`
- `debug.mono.trace`
- `debug.mono.wref`
- `XA_HTTP_CLIENT_HANDLER_TYPE`

`debug.mono.debug`

The value of the `debug.mono.debug` system property is an integer. If `1`, then behave "as if" the process were started with `mono --debug`. This generally shows file and line information in stack traces, etc., without requiring that the app be started from a debugger.

`debug.mono.env`

Contains a `|`-separated list of environment variables.

`debug.mono.gc`

The value of the `debug.mono.gc` system property is an integer. If `1`, then GC information should be logged.

This is equivalent to having the `debug.mono.log` system property contain `gc`.

`debug.mono.log`

Controls which additional information Xamarin.Android will log to `adb logcat`. It is a comma-separated string (`,`), containing one of the following values:

- `all`: Print out *all* messages. This is seldom a good idea, as it includes `lref` messages.
- `assembly`: Print out `.apk` and assembly parsing messages.
- `gc`: Print out GC-related messages.
- `gref`: Print out JNI Global Reference messages.
- `lref`: Print out JNI Local Reference messages.

> **NOTE**
>
> This will *really* spam `adb logcat`. In Xamarin.Android 5.1, this will also create a `.__override__/lrefs.txt` file, which can get *gigantic*. Avoid.

- `timing`: Print out some method timing information. This will also create the files `.__override__/methods.txt` and `.__override__/counters.txt`.

`debug.mono.max_grefc`

The value of the `debug.mono.max_grefc` system property is an integer. It's value *overrides* the default detected maximum GREF count for the target device.

Please note: This is only usable with `adb shell setprop debug.mono.max_grefc` as the value will not be available in time with an **environment.txt** file.

`debug.mono.profile`

The `debug.mono.profile` system property enables the profiler. It is equivalent to, and uses the same values as, the `mono --profile` option. (See the mono(1) man page for more information.)

`debug.mono.runtime_args`

The `debug.mono.runtime_args` system property contains additional options that should be parsed by **mono**.

`debug.mono.trace`

The `debug.mono.trace` system property enables tracing. It is equivalent to, and uses the same values as, the `mono --trace` option. (See the mono(1) man page for more information.)

In general, *do not use*. Use of tracing will spam `adb logcat` output, severaly slow down program behavior, and alter program behavior (up to and including adding additional error conditions).

Sometimes, however, it allows some additional investigation to be performed...

`debug.mono.wref`

The `debug.mono.wref` system property allows overriding the default detected JNI Weak Reference mechanism. There are two supported values:

- `jni` : Use JNI weak references, as created by `JNIEnv::NewWeakGlobalRef()` and destroyed by `JNIEnv::DeleteWeakGlobalREf()` .
- `java` : Use JNI Global references which reference `java.lang.WeakReference` instances.

`java` is used, by default, up through API-7 and on API-19 (Kit Kat) with ART enabled. (API-8 added `jni` references, and ART *broke* `jni` references.)

This system property is useful for testing and certain forms of investigation. *In general*, it should not be changed.

XA_HTTP_CLIENT_HANDLER_TYPE

First introduced in Xamarin.Android 6.1, this environment variable declares the default `HttpMessageHandler` implementation that will be used by the `HttpClient` . By default this variable is not set, and Xamarin.Android will use the `HttpClientHandler` .

```
XA_HTTP_CLIENT_HANDLER_TYPE=Xamarin.Android.Net.AndroidClientHandler
```

> **NOTE**
>
> The underlying Android device must support TLS 1.2. Android 5.0 and later support TLS 1.2

Example

```
## Comments are lines which start with '#'
## Blank lines are ignored.

## Enable GREF messages to `adb logcat`
debug.mono.log=gref

## Clear out a Mono environment variable to decrease logging
MONO_LOG_LEVEL=
```

Related Links

- Transport Layer Security

GDB

11/2/2020 • 3 minutes to read • Edit Online

Overview

Xamarin.Android 4.10 introduced partial support for using `gdb` by using the `_Gdb` MSBuild target.

> **NOTE**
>
> `gdb` support requires that the Android NDK be installed.

There are three ways to use `gdb` :

1. Debug builds with Fast Deployment enabled .
2. Debug builds with Fast Deployment disabled .
3. Release builds .

When things go wrong, please see the Troubleshooting section.

Debug Builds with Fast Deployment

When building and deploying a Debug build with Fast Deployment enabled, `gdb` can be attached by using the `_Gdb` MSBuild target.

First, install the app. This can be done via the IDE, or via the command line:

```
$ /Library/Frameworks/Mono.framework/Commands/xbuild /t:Install *.csproj
```

Secondly, run the `_Gdb` target. At the end of execution, a `gdb` command line will be printed:

```
$ /Library/Frameworks/Mono.framework/Commands/xbuild /t:_Gdb *.csproj
...
    Target _Gdb:
        "/opt/android/ndk/toolchains/arm-linux-androideabi-4.4.3/prebuilt/darwin-x86/bin/arm-linux-
androideabi-gdb" -x "/Users/jon/Development/Projects/Scratch.HelloXamarin20//gdb-symbols/gdb.env"
...
```

The `_Gdb` target will launch an arbitrary launcher Activity declared within your `AndroidManifest.xml` file. To explicitly specify which Activity to run, use the `RunActivity` MSBuild property. Starting Services and other Android constructs is not supported at this time.

The `_Gdb` target will create a `gdb-symbols` directory and copy the contents of your target's `/system/lib` and `$APPDIR/lib` directories there.

> **NOTE**
>
> The contents of the `gdb-symbols` directory are tied to the Android target you deployed to, and will not be automatically replaced should you change the target. (Consider this a bug.) If you change Android target devices, you will need to manually delete this directory.

Finally, copy the generated `gdb` command and execute it in your shell:

```
$ "/opt/android/ndk/toolchains/arm-linux-androideabi-4.4.3/prebuilt/darwin-x86/bin/arm-linux-androideabi-gdb" -x "/Users/jon/Development/Projects/Scratch.HelloXamarin20//gdb-symbols/gdb.env"
GNU gdb (GDB) 7.3.1-gg2
...
(gdb) bt
#0  0x40082e84 in nanosleep () from /Users/jon/Development/Projects/Scratch.HelloXamarin20/gdb-symbols/libc.so
#1  0x4008ffe6 in sleep () from /Users/jon/Development/Projects/Scratch.HelloXamarin20/gdb-symbols/libc.so
#2  0x74e46240 in ?? ()
#3  0x74e46240 in ?? ()
(gdb) c
```

Debug Builds without Fast Deployment

Debug builds *with* Fast Deployment operate by copying the Android NDK's `gdbserver` program into the Fast Deployment `.__override__` directory. When Fast Deployment is disabled, this directory may not exist.

There are two workarounds:

- Set the `debug.mono.log` system property so that the `.__override__` directory is created.
- Include `gdbserver` within your `.apk`.

Setting the `debug.mono.log` System Property

To set the `debug.mono.log` system property, use the `adb` command:

```
$ adb shell setprop debug.mono.log gc
```

Once the system property has been set, execute the `_Gdb` target and the printed `gdb` command, as with the Debug Builds with Fast Deployment configuration:

```
$ /Library/Frameworks/Mono.framework/Commands/xbuild /t:_Gdb *.csproj
  ...
    Target _Gdb:
        "/opt/android/ndk/toolchains/arm-linux-androideabi-4.4.3/prebuilt/darwin-x86/bin/arm-linux-androideabi-gdb" -x "/Users/jon/Development/Projects/Scratch.HelloXamarin20//gdb-symbols/gdb.env"
  ...
$ "/opt/android/ndk/toolchains/arm-linux-androideabi-4.4.3/prebuilt/darwin-x86/bin/arm-linux-androideabi-gdb" -x "/Users/jon/Development/Projects/Scratch.HelloXamarin20//gdb-symbols/gdb.env"
GNU gdb (GDB) 7.3.1-gg2
...
(gdb) c
```

Including `gdbserver` in your app

To include `gdbserver` within your app:

1. Find `gdbserver` within your Android NDK (it should be in **$ANDROID_NDK_PATH/prebuilt/android-arm/gdbserver/gdbserver**), and copy it into your Project directory.
2. Rename `gdbserver` to **libs/armeabi-v7a/libgdbserver.so**.
3. Add **libs/armeabi-v7a/libgdbserver.so** to your Project with a **Build action** of `AndroidNativeLibrary`.
4. Rebuild and reinstall your application.

Once the app has been reinstalled, execute the `_Gdb` target and the printed `gdb` command, as with the Debug

Builds with Fast Deployment configuration:

```
$ /Library/Frameworks/Mono.framework/Commands/xbuild /t:_Gdb *.csproj
  ...
    Target _Gdb:
        "/opt/android/ndk/toolchains/arm-linux-androideabi-4.4.3/prebuilt/darwin-x86/bin/arm-linux-
androideabi-gdb" -x "/Users/jon/Development/Projects/Scratch.HelloXamarin20//gdb-symbols/gdb.env"
  ...
$ "/opt/android/ndk/toolchains/arm-linux-androideabi-4.4.3/prebuilt/darwin-x86/bin/arm-linux-androideabi-
gdb" -x "/Users/jon/Development/Projects/Scratch.HelloXamarin20//gdb-symbols/gdb.env"
GNU gdb (GDB) 7.3.1-gg2
...
(gdb) c
```

Release Builds

`gdb` support requires three things:

1. The `INTERNET` permission.
2. App Debugging enabled.
3. An accessible `gdbserver`.

The `INTERNET` permission is enabled by default in Debug apps. If it is not already present in your application, you may add it either by editing **Properties/AndroidManifest.xml** or by editing the Project Properties.

App debugging can be enabled by either setting the ApplicationAttribute.Debugging custom attribute property to `true`, or by editing **Properties/AndroidManifest.xml** and setting the `//application/@android:debuggable` attribute to `true`:

```
<application android:label="Example.Name.Here" android:debuggable="true">
```

An accessible `gdbserver` may be provided by following the Debug Builds without Fast Deployment section.

One wrinkle: The `_Gdb` MSBuild target will kill any previously running app instances. This will not work on pre-Android v4.0 targets.

Troubleshooting

`mono_pmip` doesn't work

The `mono_pmip` function (useful for obtaining managed stack frames) is exported from `libmonosgen-2.0.so`, which the `_Gdb` target does not currently pull down. (This will be fixed in a future release.)

To enable calling functions located in `libmonosgen-2.0.so`, copy it from the target device into the `gdb-symbols` directory:

```
$ adb pull /data/data/Mono.Android.DebugRuntime/lib/libmonosgen-2.0.so Project/gdb-symbols
```

Then restart your debugging session.

Bus error: 10 when running the `gdb` command

When the `gdb` command errors out with `"Bus error: 10"`, restart the Android device.

```
$ "/path/to/arm-linux-androideabi-gdb" -x "Project/gdb-symbols/gdb.env"
GNU gdb (GDB) 7.3.1-gg2
Copyright (C) 2011 Free Software Foundation, Inc.
...
Bus error: 10
$
```

No stack trace after attach

```
$ "/path/to/arm-linux-androideabi-gdb" -x "Project/gdb-symbols/gdb.env"
GNU gdb (GDB) 7.3.1-gg2
Copyright (C) 2011 Free Software Foundation, Inc.
...
(gdb) bt
No stack.
```

This is usually a sign that the contents of the `gdb-symbols` directory are not synchronized with your Android target. (Did you change your Android target?)

Please delete the `gdb-symbols` directory and try again.

Linking on Android

7/5/2021 • 5 minutes to read • Edit Online

Xamarin.Android applications use a *linker* to reduce the size of the application. The linker employs static analysis of your application to determine which assemblies are actually used, which types are actually used, and which members are actually used. The linker then behaves like a *garbage collector*, continually looking for the assemblies, types, and members that are referenced until the entire closure of referenced assemblies, types, and members is found. Then everything outside of this closure is *discarded*.

For example, the Hello, Android sample:

CONFIGURATION	1.2.0 SIZE	4.0.1 SIZE
Release without Linking:	14.0 MB	16.0 MB
Release with Linking:	4.2 MB	2.9 MB

Linking results in a package that is 30% the size of the original (unlinked) package in 1.2.0, and 18% of the unlinked package in 4.0.1.

Control

Linking is based on *static analysis*. Consequently, anything that depends upon the runtime environment won't be detected:

```
// To play along at home, Example must be in a different assembly from MyActivity.
public class Example {
    // Compiler provides default constructor...
}

[Activity (Label="Linker Example", MainLauncher=true)]
public class MyActivity {
    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);

        // Will this work?
        var o = Activator.CreateInstance (typeof (ExampleLibrary.Example));
    }
}
```

Linker Behavior

The primary mechanism for controlling the linker is the **Linker Behavior** (*Linking* in Visual Studio) drop-down within the **Project Options** dialog box. There are three options:

1. **Don't Link** (*None* in Visual Studio)
2. **Link SDK Assemblies** (*Sdk Assemblies Only*)
3. **Link All Assemblies** (*Sdk and User Assemblies*)

The **Don't Link** option turns off the linker; the above "Release without Linking" application size example used this behavior. This is useful for troubleshooting runtime failures, to see if the linker is responsible. This setting is not usually recommended for production builds.

The **Link SDK Assemblies** option only links assemblies that come with Xamarin.Android. All other assemblies (such as your code) are not linked.

The **Link All Assemblies** option links all assemblies, which means your code may also be removed if there are no static references.

The above example will work with the *Don't Link* and *Link SDK Assemblies* options, and will fail with the *Link All Assemblies* behavior, generating the following error:

```
E/mono    (17755): [0xafd4d440:] EXCEPTION handling: System.MissingMethodException: Default constructor not
found for type ExampleLibrary.Example.
I/MonoDroid(17755): UNHANDLED EXCEPTION: System.MissingMethodException: Default constructor not found for
type ExampleLibrary.Example.
I/MonoDroid(17755): at System.Activator.CreateInstance (System.Type,bool) <0x00180>
I/MonoDroid(17755): at System.Activator.CreateInstance (System.Type) <0x00017>
I/MonoDroid(17755): at LinkerScratch2.Activity1.OnCreate (Android.OS.Bundle) <0x00027>
I/MonoDroid(17755): at Android.App.Activity.n_OnCreate_Landroid_os_Bundle_ (intptr,intptr,intptr) <0x00057>
I/MonoDroid(17755): at (wrapper dynamic-method) object.95bb4fbe-bef8-4e5b-8e99-ca83a5d7a124
(intptr,intptr,intptr) <0x00033>
E/mono    (17755): [0xafd4d440:] EXCEPTION handling: System.MissingMethodException: Default constructor not
found for type ExampleLibrary.Example.
E/mono    (17755):
E/mono    (17755): Unhandled Exception: System.MissingMethodException: Default constructor not found for
type ExampleLibrary.Example.
E/mono    (17755):   at System.Activator.CreateInstance (System.Type type, Boolean nonPublic) [0x00000] in
<filename unknown>:0
E/mono    (17755):   at System.Activator.CreateInstance (System.Type type) [0x00000] in <filename unknown>:0
E/mono    (17755):   at LinkerScratch2.Activity1.OnCreate (Android.OS.Bundle bundle) [0x00000] in <filename
unknown>:0
E/mono    (17755):   at Android.App.Activity.n_OnCreate_Landroid_os_Bundle_ (IntPtr jnienv, IntPtr
native__this, IntPtr native_savedInstanceState) [0x00000] in <filename unknown>:0
E/mono    (17755):   at (wrapper dynamic-method) object:95bb4fbe-bef8-4e5b-8e99-ca83a5d7a124
(intptr,intptr,intptr)
```

Preserving Code

The linker will sometimes remove code that you want to preserve. For example:

- You might have code that you call dynamically via `System.Reflection.MemberInfo.Invoke`.
- If you instantiate types dynamically, you may want to preserve the default constructor of your types.
- If you use XML serialization, you may want to preserve the properties of your types.

In these cases, you can use the Android.Runtime.Preserve attribute. Every member that is not statically linked by the application is subject to be removed, so this attribute can be used to mark members that are not statically referenced but are still needed by your application. You can apply this attribute to every member of a type, or to the type itself.

In the following example, this attribute is used to preserve the constructor of the `Example` class:

```
public class Example
{
    [Android.Runtime.Preserve]
    public Example ()
    {
    }
}
```

If you want to preserve the entire type, you can use the following attribute syntax:

```
[Android.Runtime.Preserve (AllMembers = true)]
```

For example, in the following code fragment the entire `Example` class is preserved for XML serialization:

```
[Android.Runtime.Preserve (AllMembers = true)]
class Example
{
    // Compiler provides default constructor...
}
```

Sometimes you want to preserve certain members, but only if the containing type was preserved. In those cases, use the following attribute syntax:

```
[Android.Runtime.Preserve (Conditional = true)]
```

If you do not want to take a dependency on the Xamarin libraries – for example, you are building a cross platform portable class library (PCL) – you can still use the `Android.Runtime.Preserve` attribute. To do this, declare a `PreserveAttribute` class within the `Android.Runtime` namespace like this:

```
namespace Android.Runtime
{
    public sealed class PreserveAttribute : System.Attribute
    {
        public bool AllMembers;
        public bool Conditional;
    }
}
```

In the above examples, the `Preserve` attribute is declared in the `Android.Runtime` namespace; however, you can use the `Preserve` attribute in any namespace because the linker looks up this attribute by type name.

falseflag

If the [Preserve] attribute can't be used, it is often useful to provide a block of code so that the linker believes that the type is used, while preventing the block of code from being executed at runtime. To make use of this technique, we could do:

```
[Activity (Label="Linker Example", MainLauncher=true)]
class MyActivity {

#pragma warning disable 0219, 0649
    static bool falseflag = false;
    static MyActivity ()
    {
        if (falseflag) {
            var ignore = new Example ();
        }
    }
#pragma warning restore 0219, 0649

    // ...
}
```

linkskip

It is possible to specify that a set of user-provided assemblies should not be linked at all, while allowing other user assemblies to be skipped with the *Link SDK Assemblies* behavior by using the AndroidLinkSkip MSBuild property:

```
<PropertyGroup>
    <AndroidLinkSkip>Assembly1;Assembly2</AndroidLinkSkip>
</PropertyGroup>
```

LinkDescription

The `@(LinkDescription)` **Build action** may be used on files which can contain a Custom linker configuration file. file. Custom linker configuration files may be required to preserve `internal` or `private` members that need to be preserved.

Custom Attributes

When an assembly is linked, the following custom attribute types will be removed from all members:

- System.ObsoleteAttribute
- System.MonoDocumentationNoteAttribute
- System.MonoExtensionAttribute
- System.MonoInternalNoteAttribute
- System.MonoLimitationAttribute
- System.MonoNotSupportedAttribute
- System.MonoTODOAttribute
- System.Xml.MonoFIXAttribute

When an assembly is linked, the following custom attribute types will be removed from all members in Release builds:

- System.Diagnostics.DebuggableAttribute
- System.Diagnostics.DebuggerBrowsableAttribute
- System.Diagnostics.DebuggerDisplayAttribute
- System.Diagnostics.DebuggerHiddenAttribute
- System.Diagnostics.DebuggerNonUserCodeAttribute
- System.Diagnostics.DebuggerStepperBoundaryAttribute
- System.Diagnostics.DebuggerStepThroughAttribute
- System.Diagnostics.DebuggerTypeProxyAttribute
- System.Diagnostics.DebuggerVisualizerAttribute

Related Links

- Custom Linker Configuration
- Linking on iOS

Multi-Core Devices & Xamarin.Android

7/8/2021 • 9 minutes to read • Edit Online

Android can run on several different computer architectures. This document discusses the different CPU architectures that may be employed for a Xamarin.Android application. This document will also explain how Android applications are packaged to support different CPU architectures. The Application Binary Interface (ABI) will be introduced, and guidance will be provided regarding which ABIs to use in a Xamarin.Android application.

Overview

Android allows for the creation of "fat binaries," a single `.apk` file that contains machine code that will support multiple, different CPU architectures. This is accomplished by associating each piece of machine code with an *Application Binary Interface*. The ABI is used to control which machine code will run on a given hardware device. For example, for an Android application to run on an x86 device, it is necessary to include x86 ABI support when compiling the application.

Specifically, each Android application will support at least one *embedded-application binary interface* (EABI). EABI are conventions specific to embedded software programs. A typical EABI will describe things such as:

- The CPU instruction set.
- The endianness of memory stores and loads at run time.
- The binary format of object files and program libraries, as well as which type of content is allowed or supported in these files and libraries.
- The various conventions used to pass data between application code and the system (for example: how registers and/or the stack are used when functions are called, alignment constraints, etc.)
- Alignment and size constraints for enum types, structures, fields, and arrays.
- The list of function symbols available to your machine code at run time, generally from a very specific selected set of libraries.

armeabi and Thread Safety

The Application Binary Interface will be discussed in detail below, but it is important to remember that the `armeabi` runtime used by Xamarin.Android is *not thread safe*. If an application that has `armeabi` support is deployed to an `armeabi-v7a` device, many strange and unexplainable exceptions will occur.

Due to a bug in Android 4.0.0, 4.0.1, 4.0.2, and 4.0.3, the native libraries will be picked up from the `armeabi` directory even though there is an `armeabi-v7a` directory present and the device is an `armeabi-v7a` device.

> **NOTE**
>
> Xamarin.Android will ensure that `.so` are added to the APK in the correct order. This bug should not be an issue for users of Xamarin.Android.

ABI Descriptions

Each ABI supported by Android is identified by a unique name.

armeabi

This is the name of an EABI for ARM-based CPUs that support at least the ARMv5TE instruction set. Android follows the little-endian ARM GNU/Linux ABI. This ABI does not support hardware-assisted floating-point computations. All FP operations are performed by software helper functions that come from the compiler's `libgcc.a` static library. SMP devices are not supported by `armeabi`.

> **IMPORTANT**
>
> Xamarin.Android's `armeabi` code is not thread safe and should not be used on multi-CPU `armeabi-v7a` devices (described below). Using `armeabi` code on a single-core `armeabi-v7a` device is safe.

armeabi-v7a

This is another ARM-based CPU instruction set that extends the `armeabi` EABI described above. The `armeabi-v7a` EABI has support for hardware floating-point operations and multiple CPU (SMP) devices. An application that uses the `armeabi-v7a` EABI can expect substantial performance improvements over an application that uses `armeabi`.

> **NOTE**
>
> `armeabi-v7a` machine code will not run on ARMv5 devices.

arm64-v8a

This is a 64-bit instruction set that is based on the ARMv8 CPU architecture. This architecture is used in the *Nexus 9*. Xamarin.Android 5.1 introduced support for this architecture (for more information, see 64-bit runtime support).

x86

This is the name of an ABI for CPUs that support the instruction set commonly named *x86* or *IA-32*. This ABI corresponds to instructions for the Pentium Pro instruction set, including the MMX, SSE, SSE2, and SSE3 instruction sets. It does not include any other optional IA-32 instruction set extensions such as:

- the MOVBE instruction.
- Supplemental SSE3 extension (SSSE3).
- any variant of SSE4.

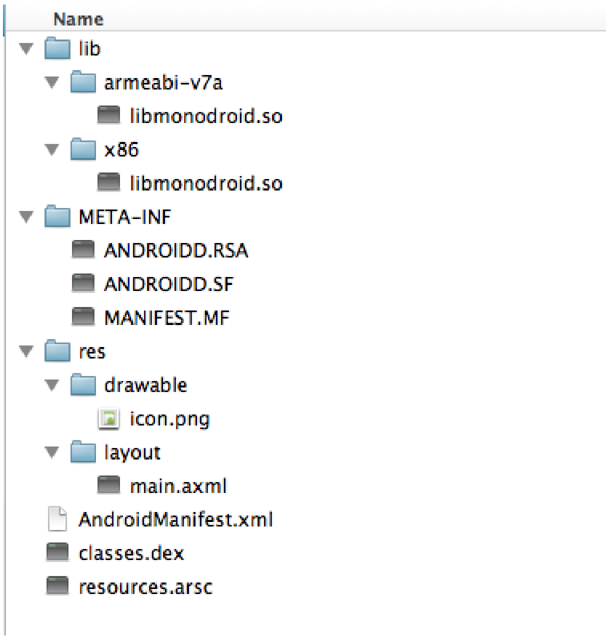
> **NOTE**
>
> Google TV, although it runs on x86, is not supported by Android's NDK.

x86_64

This is the name of an ABI for CPUs that support the 64-bit x86 instruction set (also referred to as *x64* or *AMD64*). Xamarin.Android 5.1 introduced support for this architecture (for more information, see 64-bit runtime support).

APK File Format

The Android Application Package is the file format that holds all of the code, assets, resources, and certificates necessary for an Android application. It is a `.zip` file, but uses the `.apk` file name extension. When expanded, the contents of an `.apk` created by Xamarin.Android can be seen in the screenshot below:



A quick description of the contents of the `.apk` file:

- **AndroidManifest.xml** – This is the `AndroidManifest.xml` file, in binary XML format.
- **classes.dex** – This contains the application code, compiled into the `dex` file format that is used by the Android runtime VM.
- **resources.arsc** – This file contains all of the precompiled resources for the application.
- **lib** – This directory holds the compiled code for each ABI. It will contain one subfolder for each ABI that was described in the previous section. In the screenshot above, the `.apk` in question has native libraries for both `armeabi-v7a` and for `x86` .
- **META-INF** – This directory (if present) is used to store signing information, package, and extension configuration data.
- **res** – This directory holds the resources that were not compiled into `resources.arsc` .

> **NOTE**
> The file `libmonodroid.so` is the native library required by all Xamarin.Android applications.

Android Device ABI Support

Each Android device supports executing native code in up to two ABIs:

- **The "primary" ABI** – This corresponds to the machine code used in the system image.
- **A "secondary" ABI** – This is an optional ABI that is also supported by the system image.

For example, a typical ARMv5TE device will only have a primary ABI of `armeabi` , while an ARMv7 device would specify a primary ABI of `armeabi-v7a` and a secondary ABI of `armeabi` . A typical x86 device would only specify a primary ABI of `x86` .

Android Native Library Installation

At package installation time, native libraries within the `.apk` are extracted into the app's native library directory, typically `/data/data/<package-name>/lib` , and are thereafter referred to as `$APP/lib` .

Android's native library installation behavior varies dramatically between Android versions.

Installing Native Libraries: Pre-Android 4.0

Android prior to 4.0 Ice Cream Sandwich will only extract native libraries from a *single ABI* within the `.apk`. Android apps of this vintage will first try to extract all native libraries for the primary ABI, and if no such libraries exist, Android will then extract all native libraries for the secondary ABI. No "merging" is done.

For example, consider a situation where an application is installed on an `armeabi-v7a` device. The `.apk,` which supports both `armeabi` and `armeabi-v7a`, has the following ABI `lib` directories and files in it:

```
lib/armeabi/libone.so
lib/armeabi/libtwo.so
lib/armeabi-v7a/libtwo.so
```

After installation, the native library directory will contain:

```
$APP/lib/libtwo.so # from the armeabi-v7a directory in the apk
```

In other words, no `libone.so` is installed. This will cause problems, as `libone.so` is not present for the application to load at run time. This behavior, while unexpected, has been logged as a bug and reclassified as "working as intended."

Consequently, when targeting Android versions prior to 4.0, it is necessary to provide *all* native libraries for *each* ABI that the application will support, that is, the `.apk` should contain:

```
lib/armeabi/libone.so
lib/armeabi/libtwo.so
lib/armeabi-v7a/libone.so
lib/armeabi-v7a/libtwo.so
```

Installing Native Libraries: Android 4.0 – Android 4.0.3

Android 4.0 Ice Cream Sandwich changes the extraction logic. It will enumerate all native libraries, see if the file's basename has already been extracted, and if both of the following conditions are met, then the library will be extracted:

- It hasn't already been extracted.
- The native library's ABI matches the target's primary or secondary ABI.

Meeting these conditions allows "merging" behavior; that is, if we have an `.apk` with the following contents:

```
lib/armeabi/libone.so
lib/armeabi/libtwo.so
lib/armeabi-v7a/libtwo.so
```

Then after installation, the native library directory will contain:

```
$APP/lib/libone.so
$APP/lib/libtwo.so
```

Unfortunately, this behavior is order dependent, as described in the following document - Issue 24321: Galaxy Nexus 4.0.2 uses armeabi native code when both armeabi and armeabi-v7a is included in apk.

The native libraries are processed "in order" (as listed by, for example, unzip), and the *first match* is extracted. Since the `.apk` contains `armeabi` and `armeabi-v7a` versions of `libtwo.so`, and the `armeabi` is listed first, it's the `armeabi` version that is extracted, *not* the `armeabi-v7a` version:

```
$APP/lib/libone.so # armeabi
$APP/lib/libtwo.so # armeabi, NOT armeabi-v7a!
```

Furthermore, even if both `armeabi` and `armeabi-v7a` ABIs are specified (as described below in the section *Declaring Supported ABIs*), Xamarin.Android will create the following element in the . `csproj`:

```
<AndroidSupportedAbis>armeabi,armeabi-v7a</AndroidSupportedAbis>
```

Consequently, the `armeabi` `libmonodroid.so` will be found first within the `.apk`, and the `armeabi` `libmonodroid.so` will be the one that is extracted, even though the `armeabi-v7a` `libmonodroid.so` is present and optimized for the target. This can also result in obscure run-time errors, as `armeabi` is not SMP safe.

Installing Native Libraries: Android 4.0.4 and later

Android 4.0.4 changes the extraction logic: it will enumerate all native libraries, read the file's basename, then extract the primary ABI version (if present), or the secondary ABI (if present). This allows "merging" behavior; that is, if we have an `.apk` with the following contents:

```
lib/armeabi/libone.so
lib/armeabi/libtwo.so
lib/armeabi-v7a/libtwo.so
```

Then after installation, the native library directory will contain:

```
$APP/lib/libone.so # from armeabi
$APP/lib/libtwo.so # from armeabi-v7a
```

Xamarin.Android and ABIs

Xamarin.Android supports the following *64-bit* architectures:

- `arm64-v8a`
- `x86_64`

> **NOTE**
>
> From August 2018 new apps will be required to target API level 26, and from August 2019 apps will be required to provide 64-bit versions in addition to the 32-bit version.

Xamarin.Android supports these 32-bit architectures:

- `armeabi` ^
- `armeabi-v7a`
- `x86`

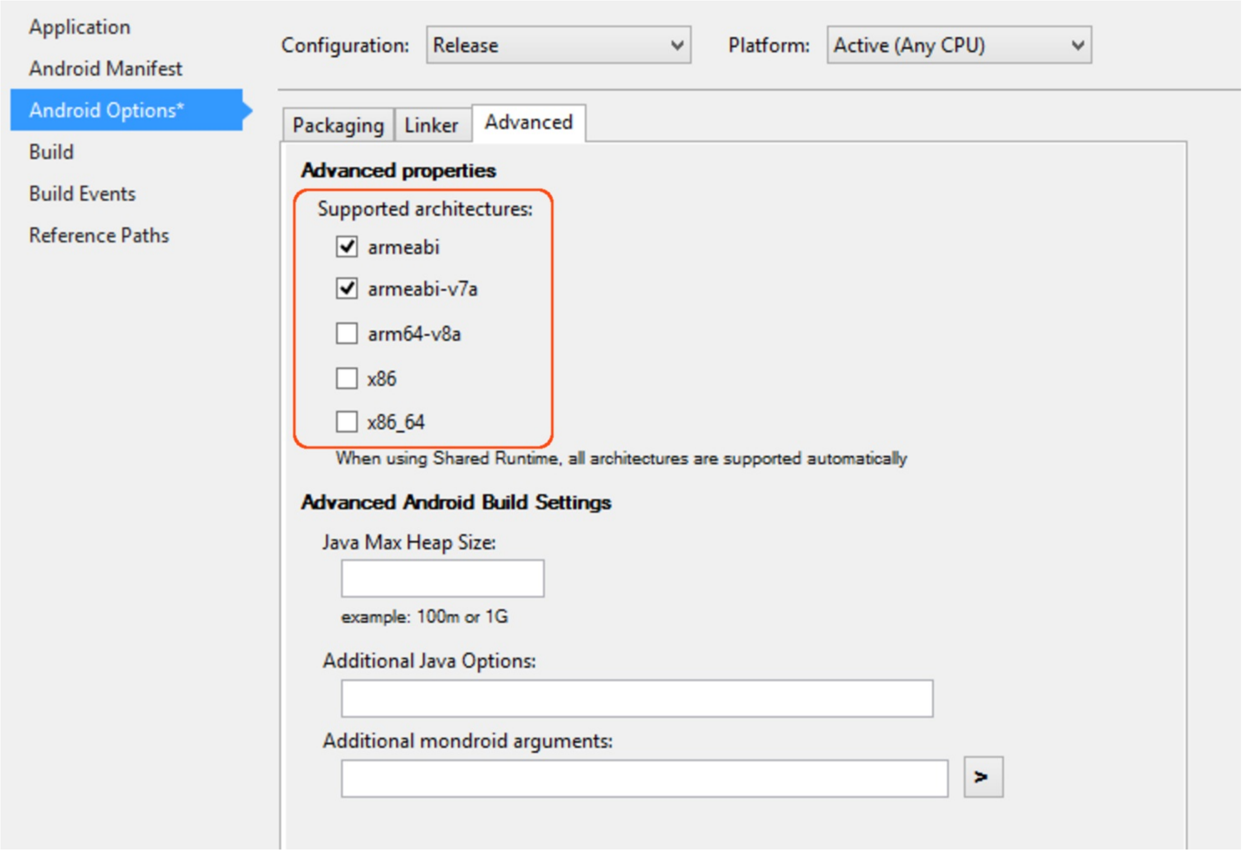
> **NOTE**
>
> ^ As of Xamarin.Android 9.2, `armeabi` is no longer supported.

Xamarin.Android does not currently provide support for `mips`.

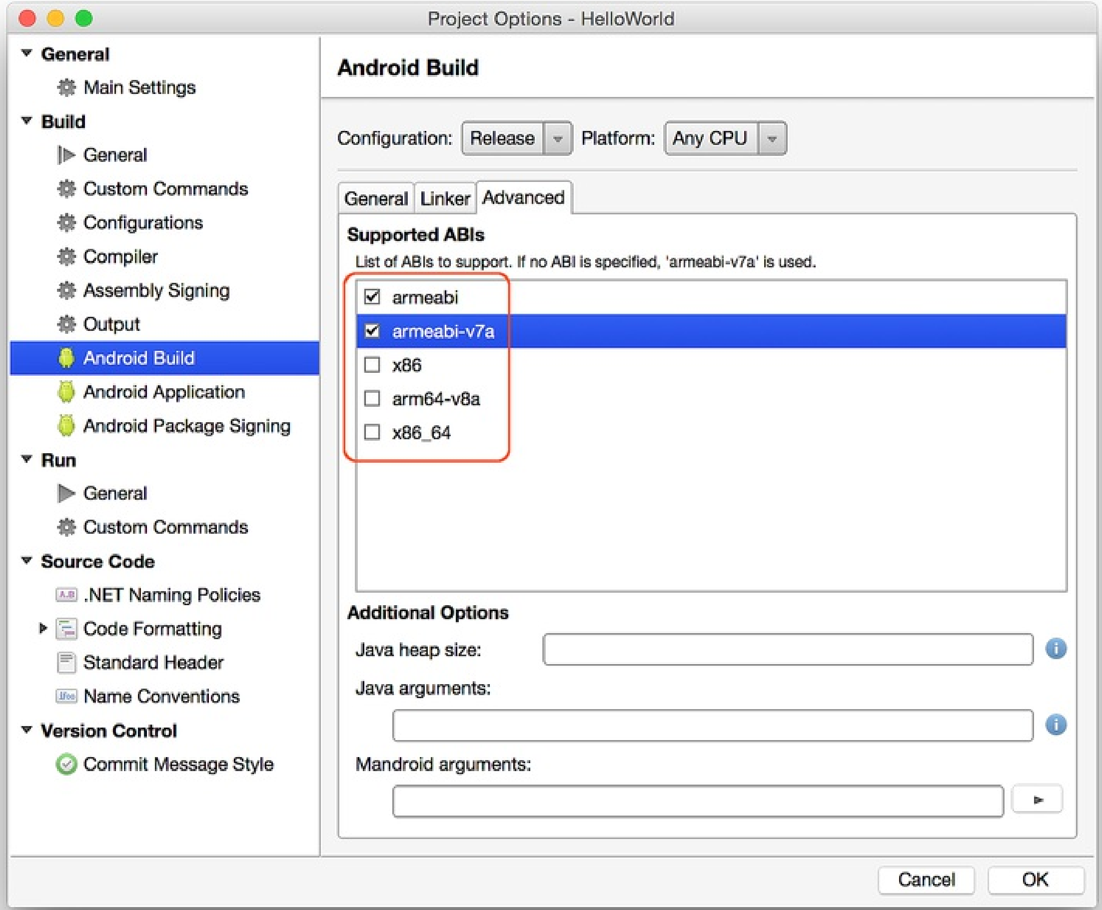
Declaring Supported ABI's

By default, Xamarin.Android will default to `armeabi-v7a` for **Release** builds, and to `armeabi-v7a` and `x86` for

Debug builds. Support for different ABIs can be set through the Project Options for a Xamarin.Android project. In Visual Studio, this can be set in the **Android Options** page of project **Properties**, under the **Advanced** tab, as shown in the following screenshot:



In Visual Studio for Mac, the supported architectures may be selected on the **Android Build** page of **Project Options**, under the **Advanced** tab, as shown in the following screenshot:



There are some situations when it may be necessary to declare additional ABI support such as when:

- Deploying the application to an `x86` device.
- Deploying the application to an `armeabi-v7a` device to ensure thread safety.

Summary

This document discussed the different CPU architectures that an Android application may run on. It introduced the Application Binary Interface and how it is used by Android to support disparate CPU architectures. It then went on to discuss how to specify ABI support in a Xamarin.Android application and highlighted the issues that arise when using Xamarin.Android applications on an `armeabi-v7a` device that are intended only for `armeabi`.

Related Links

- Android NDK
- Issue 9089:Nexus One - Won't load ANY native libraries from armeabi if there's at least one library at armeabi-v7a
- Issue 24321: Galaxy Nexus 4.0.2 uses armeabi native code when both armeabi and armeabi-v7a is included in apk

Xamarin.Android Performance

11/2/2020 • 7 minutes to read • Edit Online

There are many techniques for increasing the performance of applications built with Xamarin.Android. Collectively these techniques can greatly reduce the amount of work being performed by a CPU, and the amount of memory consumed by an application. This article describes and discusses these techniques.

Performance Overview

Poor application performance presents itself in many ways. It can make an application seem unresponsive, can cause slow scrolling, and can reduce battery life. However, optimizing performance involves more than just implementing efficient code. The user's experience of application performance must also be considered. For example, ensuring that operations execute without blocking the user from performing other activities can help to improve the user's experience.

There are a number of techniques for increasing the performance, and perceived performance, of applications built with Xamarin.Android. They include:

- Optimize Layout Hierarchies
- Optimize List Views
- Remove Event Handlers in Activities
- Limit the Lifespan of Services
- Release Resources when Notified
- Release Resources when the User Interface is Hidden
- Optimize Image Resources
- Dispose of Unused Image Resources
- Avoid Floating-Point Arithmetic
- Dismiss Dialogs

NOTE

Before reading this article you should first read Cross-Platform Performance, which discusses non-platform specific techniques to improve the memory usage and performance of applications built using the Xamarin platform.

Optimize Layout Hierarchies

Each layout added to an application requires initialization, layout, and drawing. The layout pass can be expensive when nesting `LinearLayout` instances that use the `weight` parameter, because each child will be measured twice. Using nested instances of `LinearLayout` can lead to a deep view hierarchy, which can result in poor performance for layouts that are inflated multiple times, such as in a `ListView`. Therefore, it's important that such layouts are optimized, as the performance benefits will then be multiplied.

For example, consider the `LinearLayout` for a list view row that has an icon, a title, and a description. The `LinearLayout` will contain an `ImageView` and a vertical `LinearLayout` that contains two `TextView` instances:

```xml
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="?android:attr/listPreferredItemHeight"
    android:padding="5dip">
    <ImageView
        android:id="@+id/icon"
        android:layout_width="wrap_content"
        android:layout_height="fill_parent"
        android:layout_marginRight="5dip"
        android:src="@drawable/icon" />
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="0dip"
        android:layout_weight="1"
        android:layout_height="fill_parent">
        <TextView
            android:layout_width="fill_parent"
            android:layout_height="0dip"
            android:layout_weight="1"
            android:gravity="center_vertical"
            android:text="Mei tempor iuvaret ad." />
        <TextView
            android:layout_width="fill_parent"
            android:layout_height="0dip"
            android:layout_weight="1"
            android:singleLine="true"
            android:ellipsize="marquee"
            android:text="Lorem ipsum dolor sit amet." />
    </LinearLayout>
</LinearLayout>
```

This layout is 3-levels deep, and is wasteful when inflated for each `ListView` row. However, it can be improved by flattening the layout, as shown in the following code example:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="?android:attr/listPreferredItemHeight"
    android:padding="5dip">
    <ImageView
        android:id="@+id/icon"
        android:layout_width="wrap_content"
        android:layout_height="fill_parent"
        android:layout_alignParentTop="true"
        android:layout_alignParentBottom="true"
        android:layout_marginRight="5dip"
        android:src="@drawable/icon" />
    <TextView
        android:id="@+id/secondLine"
        android:layout_width="fill_parent"
        android:layout_height="25dip"
        android:layout_toRightOf="@id/icon"
        android:layout_alignParentBottom="true"
        android:layout_alignParentRight="true"
        android:singleLine="true"
        android:ellipsize="marquee"
        android:text="Lorem ipsum dolor sit amet." />
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_toRightOf="@id/icon"
        android:layout_alignParentRight="true"
        android:layout_alignParentTop="true"
        android:layout_above="@id/secondLine"
        android:layout_alignWithParentIfMissing="true"
        android:gravity="center_vertical"
        android:text="Mei tempor iuvaret ad." />
</RelativeLayout>
```

The previous 3-level hierarchy has been reduced to a 2-level hierarchy, and a single `RelativeLayout` has replaced two `LinearLayout` instances. A significant performance increase will be gained when inflating the layout for each `ListView` row.

Optimize List Views

Users expect smooth scrolling and fast load times for `ListView` instances. However, scrolling performance can suffer when each list view row contains deeply nested view hierarchies, or when list view rows contain complex layouts. However, there are techniques that can be used to avoid poor `ListView` performance:

- Reuse row views For more information, see Reuse Row Views.
- Flatten layouts, where possible.
- Cache row content that is retrieved from a web service.
- Avoid image scaling.

Collectively these techniques can help to keep `ListView` instances scrolling smoothly.

Reuse Row Views

When displaying hundreds of rows in a `ListView`, it would be a waste of memory to create hundreds of `View` objects when only a small number of them are displayed on screen at once. Instead, only the `View` objects visible in the rows on screen can be loaded into memory, with the **content** being loaded into these reused objects. This prevents the instantiation of hundreds of additional objects, saving time and memory.

Therefore, when a row disappears from the screen its view can be placed in a queue for reuse, as shown in the following code example:

```
public override View GetView(int position, View convertView, ViewGroup parent)
{
   View view = convertView; // re-use an existing view, if one is supplied
   if (view == null) // otherwise create a new one
       view = context.LayoutInflater.Inflate(Android.Resource.Layout.SimpleListItem1, null);
   // set view properties to reflect data for the given row
   view.FindViewById<TextView>(Android.Resource.Id.Text1).Text = items[position];
   // return the view, populated with data, for display
   return view;
}
```

As the user scrolls, the `ListView` calls the `GetView` override to request new views to display – if available it passes an unused view in the `convertView` parameter. If this value is `null` then the code creates a new `View` instance, otherwise the `convertView` properties can be reset and reused.

For more information, see Row View Re-Use in Populating a ListView with Data.

Remove Event Handlers in Activities

When an activity is destroyed in the Android runtime, it could still be alive in the Mono runtime. Therefore, remove event handlers to external objects in `Activity.OnPause` to prevent the runtime from keeping a reference to an activity that has been destroyed.

In an activity, declare event handler(s) at class level:

```
EventHandler<UpdatingEventArgs> service1UpdateHandler;
```

Then implement the handlers in the activity, such as in `OnResume`:

```
service1UpdateHandler = (object s, UpdatingEventArgs args) => {
    this.RunOnUiThread (() => {
        this.updateStatusText1.Text = args.Message;
    });
};
App.Current.Service1.Updated += service1UpdateHandler;
```

When the activity exits the running state, `OnPause` is called. In the `OnPause` implementation, remove the handlers as follows:

```
App.Current.Service1.Updated -= service1UpdateHandler;
```

Limit the Lifespan of Services

When a service starts, Android keeps the service process running. This makes the process expensive because its memory can't be paged, or used elsewhere. Leaving a service running when it's not required therefore increases the risk of an application exhibiting poor performance due to memory constraints. It can also make application switching less efficient as it reduces the number of processes Android can cache.

The lifespan of a service can be limited by using an `IntentService`, which terminates itself once it's handled the intent that started it.

Release Resources when Notified

During the application lifecycle, the `OnTrimMemory` callback provides a notification when the device memory is low. This callback should be implemented to listen for the following memory level notifications:

- `TrimMemoryRunningModerate` – the application *may* want to release some unneeded resources.
- `TrimMemoryRunningLow` – the application *should* release unneeded resources.
- `TrimMemoryRunningCritical` – the application *should* release as many non-critical processes as it can.

In addition, when the application process is cached, the following memory level notifications may be received by the `OnTrimMemory` callback:

- `TrimMemoryBackground` – release resources that can be quickly and efficiently rebuilt if the user returns to the app.
- `TrimMemoryModerate` – releasing resources can help the system keep other processes cached for better overall performance.
- `TrimMemoryComplete` – the application process will soon be terminated if more memory isn't soon recovered.

Notifications should be responded to by releasing resources based on the received level.

Release Resources when the User Interface is Hidden

Release any resources used by the app's user interface when the user navigates to another app, as it can significantly increase Android's capacity for cached processes, which in turn can have an impact on the user experience quality.

To receive a notification when the user exits the UI, implement the `OnTrimMemory` callback in `Activity` classes and listen for the `TrimMemoryUiHidden` level, which indicates that the UI is hidden from view. This notification will be received only when *all* the UI components of the application become hidden from the user. Releasing UI resources when this notification is received ensures that if the user navigates back from another activity in the app, the UI resources are still available to quickly resume the activity.

Optimize Image Resources

Images are some of the most expensive resources that applications use, and are often captured at high resolutions. Therefore, when displaying an image, display it at the resolution required for the device's screen. If the image is of a higher resolution than the screen, it should be scaled down.

For more information, see Optimize Image Resources in the Cross-Platform Performance guide.

Dispose of Unused Image Resources

To save on memory usage, it is a good idea to dispose of large image resources that are no longer needed. However, it is important to ensure that images are disposed of correctly. Instead of using an explicit `.Dispose()` invocation, you can take advantage of using statements to ensure correct use of `IDisposable` objects.

For example, the Bitmap class implements `IDisposable`. Wrapping the instantiation of a `BitMap` object in a `using` block ensures that it will be disposed of correctly on exit from the block:

```
using (Bitmap smallPic = BitmapFactory.DecodeByteArray(smallImageByte, 0, smallImageByte.Length))
{
    // Use the smallPic bit map here
}
```

For more information about releasing disposable resources, see Release IDisposable Resources.

Avoid Floating-Point Arithmetic

On Android devices, floating-point arithmetic is about 2x slower than integer arithmetic. Therefore, replace floating-point arithmetic with integer arithmetic if possible. However, there's no execution time difference between `float` and `double` arithmetic on recent hardware.

> **NOTE**
>
> Even for integer arithmetic, some CPUs lack hardware divide capabilities. Therefore, integer division and modulus operations are often performed in software.

Dismiss Dialogs

When using the `ProgressDialog` class (or any dialog or alert), instead of calling the `Hide` method when the dialog's purpose is complete, call the `Dismiss` method. Otherwise, the dialog will still be alive and will leak the activity by holding a reference to it.

Summary

This article described and discussed techniques for increasing the performance of applications built with Xamarin.Android. Collectively these techniques can greatly reduce the amount of work being performed by a CPU, and the amount of memory consumed by an application.

Related Links

- Cross-Platform Performance

Profiling Android Apps

7/8/2021 • 3 minutes to read • Edit Online

Before deploying your app to an app store, it's important to identify and fix any performance bottlenecks, excessive memory usage issues, or inefficient use of network resources. Two profiler tools are available to serve this purpose:

- Xamarin Profiler
- Android Profiler in Android Studio

This guide introduces the Xamarin Profiler and provides detailed information for getting started with using the Android Profiler.

Xamarin Profiler

The Xamarin Profiler is a standalone application that is integrated with Visual Studio and Visual Studio for Mac for profiling Xamarin apps from within the IDE. For more information about using the Xamarin Profiler, see Xamarin Profiler.

> **NOTE**
> You must be a Visual Studio Enterprise subscriber to unlock the Xamarin Profiler feature in either Visual Studio Enterprise on Windows or Visual Studio for Mac.

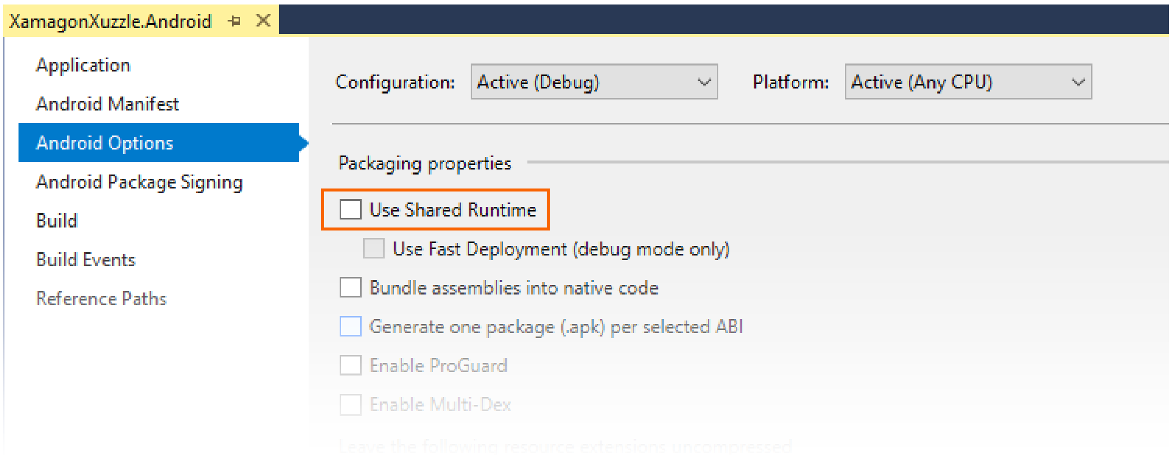
Android Studio Profiler

Android Studio 3.0 and later includes an Android Profiler tool. You can use the Android Profiler to measure the performance of a Xamarin Android app built with Visual Studio – without the need for a Visual Studio Enterprise license. However, unlike the Xamarin Profiler, the Android Profiler is not integrated with Visual Studio and can only be used to profile an Android application package (APK) that has been built in advance and imported into the Android Profiler.

Launching a Xamarin Android app in Android Profiler

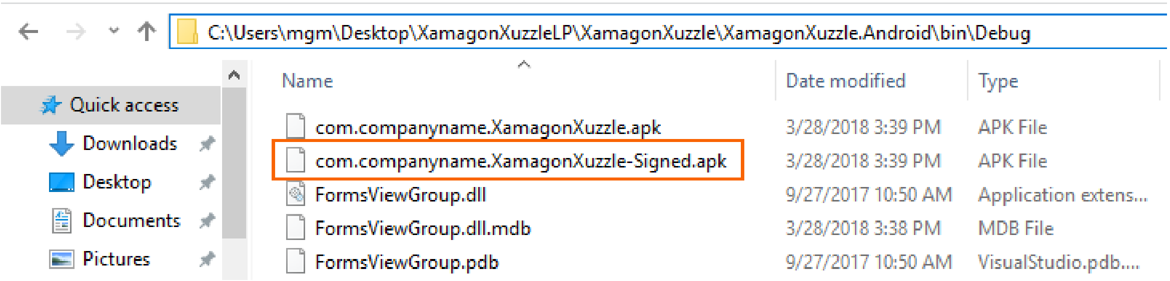
The following steps explain how to launch an Xamarin Android application in Android Studio's Android Profiler tool. In the example screenshots below, the Xamarin Forms XamagonXuzzle app is built and profiled using Android Profiler:

1. In the Android project build options, disable **Use Shared Runtime**. This ensures that the Android application package (APK) is built without a dependency on the shared development-time Mono runtime.

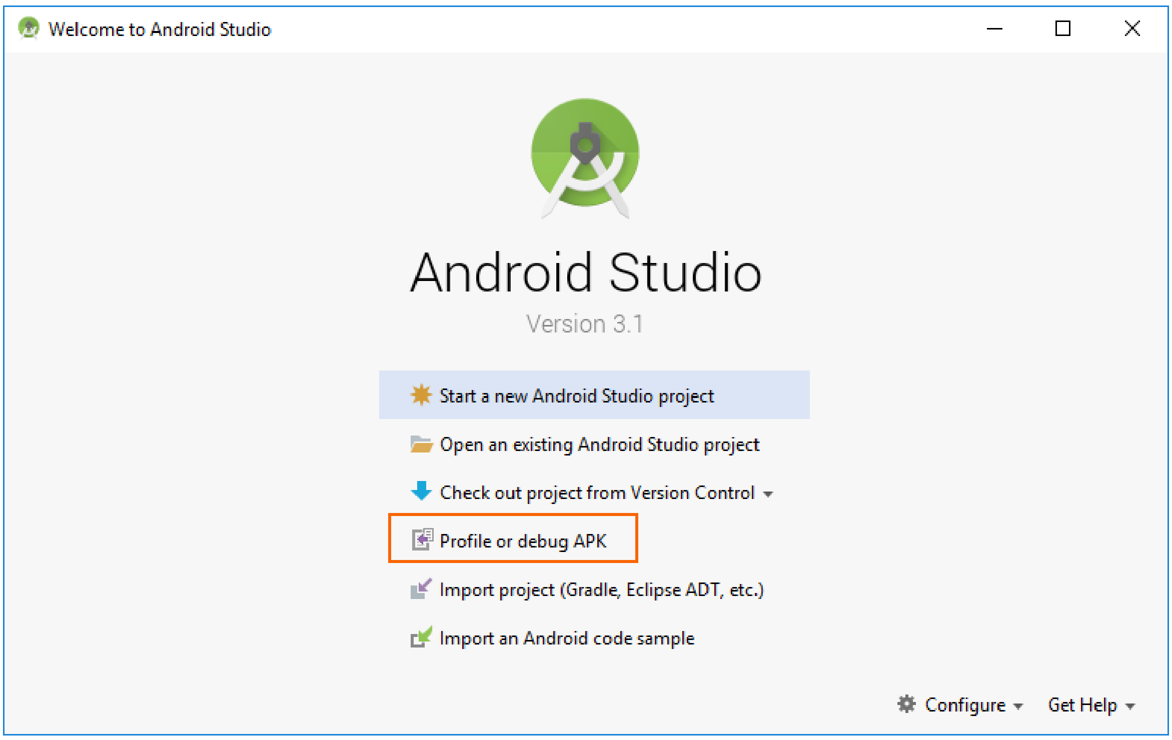


2. Build the app for **Debug** and deploy it to a physical device or emulator. This causes a signed **Debug** version of the APK to be built. For the **XamagonXuzzle** example, the resulting APK is named **com.companyname.XamagonXuzzle-Signed.apk**.

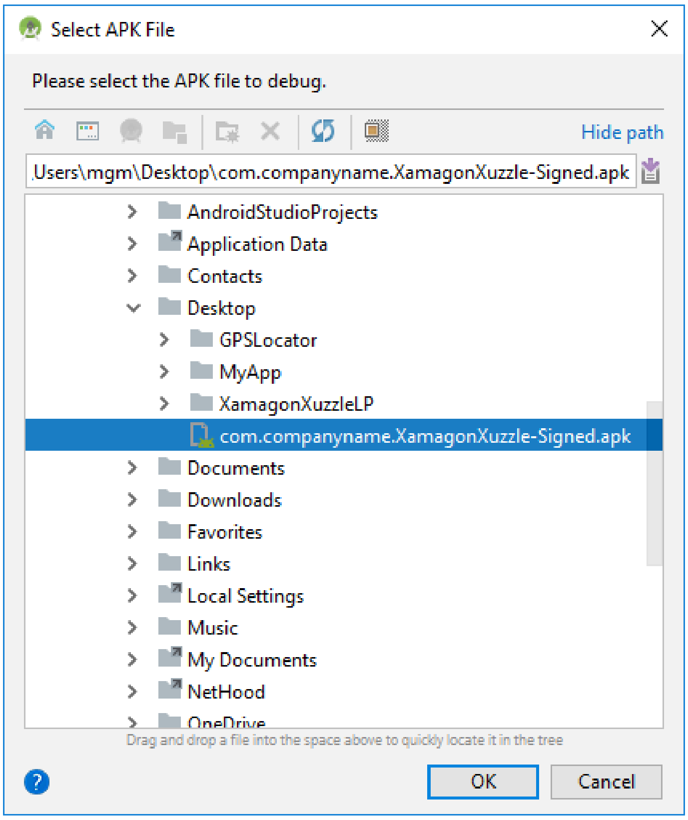
3. Open the project folder and navigate to **bin/Debug**. In this folder, locate the **Signed.apk** version of the app and copy it to a conveniently-accessible place (such as the desktop). In the following screenshot, the APK **com.companyname.XamagonXuzzle-Signed.apk** is located and copied to the desktop:



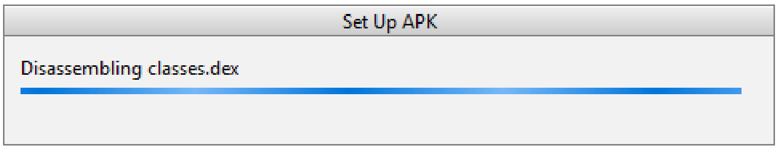
4. Launch Android Studio and select **Profile or debug APK**:



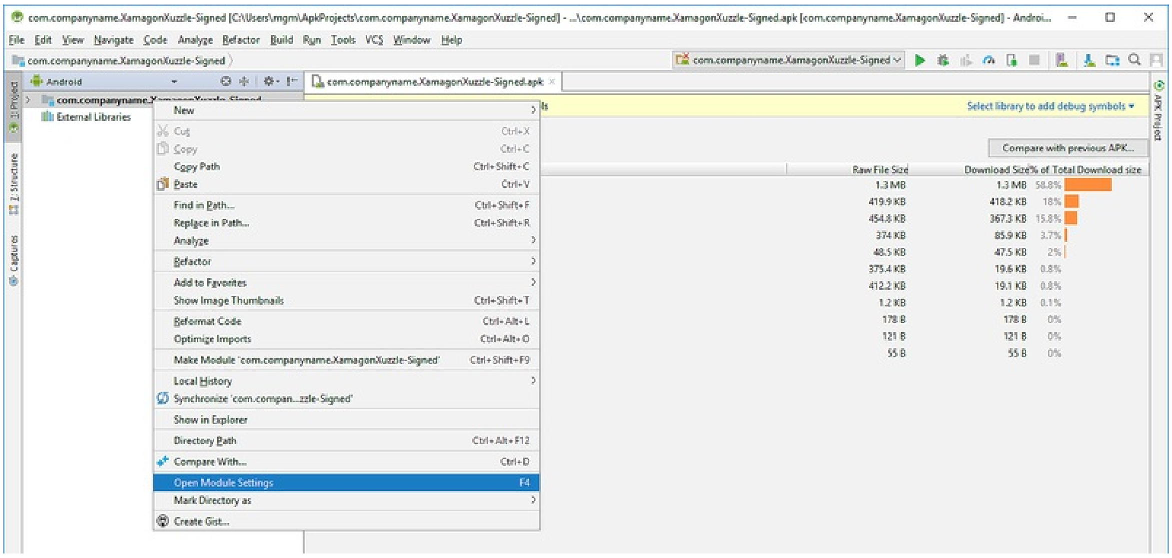
5. In the **Select APK File** dialog, navigate to the APK that you built and copied earlier. Select the APK and click **OK**:



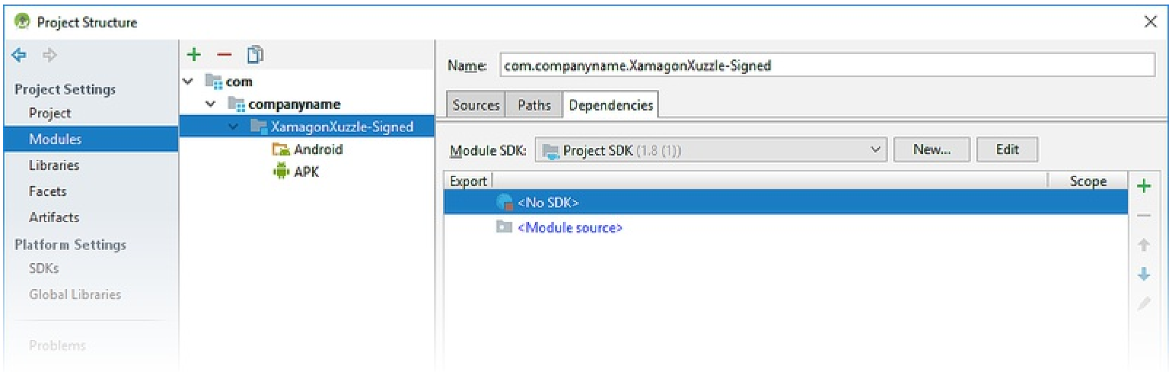
6. Android Studio will load the APK and dissassembles **classes.dex**:



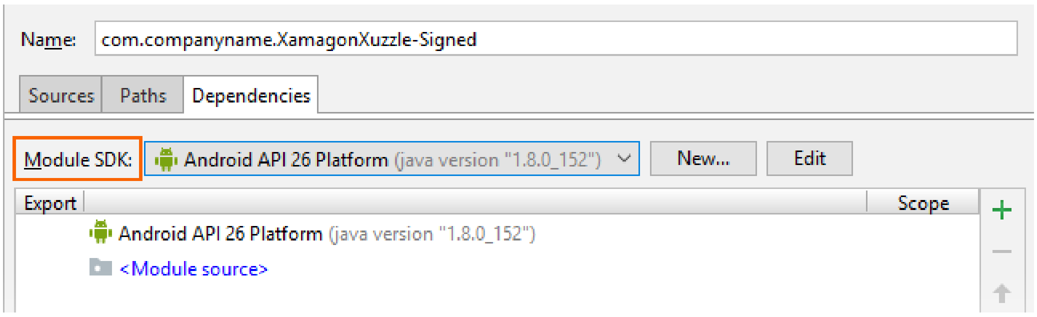
7. After the APK is loaded, Android Studio displays the following project screen for the APK. Right-click the app name in the tree view on the left and select **Open Module Settings**:



8. Navigate to **Project Settings > Modules**, select the **-Signed** node of the app, then click **<No SDK>**:

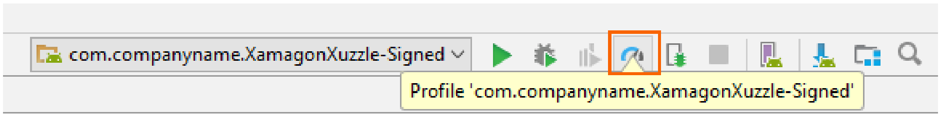


9. In the **Module SDK** pull-down menu, select the Android SDK level that was used to build the app (in this example, API level 26 was used to build **XamagonXuzzle**):

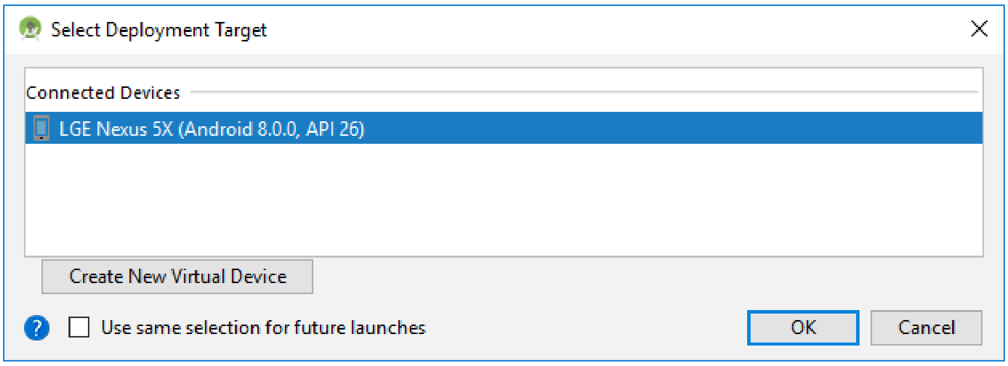


Click **Apply** and **OK** to save this setting.

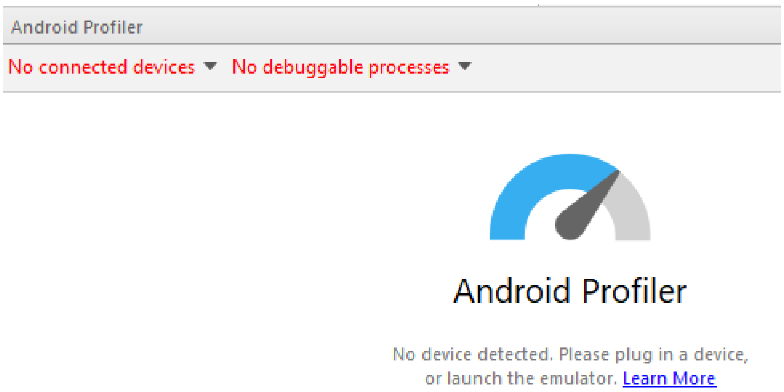
10. Launch the profiler from the toolbar icon:



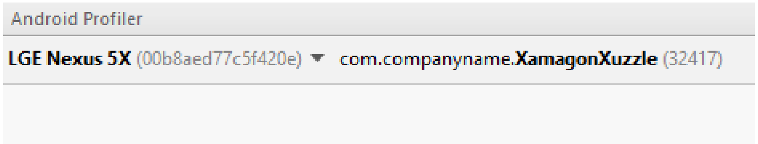
11. Select the deployment target for running/profiling the app and click **OK**. The deployment target can be a physical device or a virtual device running in an emulator. In this example, a Nexus 5X device is used:



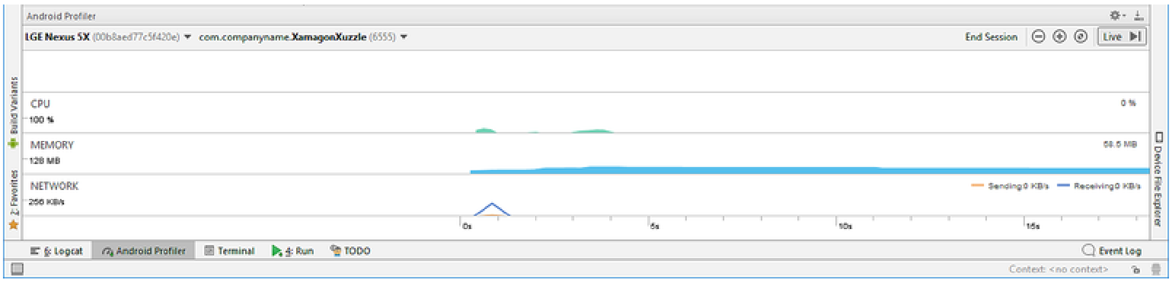
12. After the profiler starts, it will take a few seconds for it to connect to the deployment device and the app process. While it is installing the APK, Android Profiler will report **No connected devices** and **No debuggable processes**.



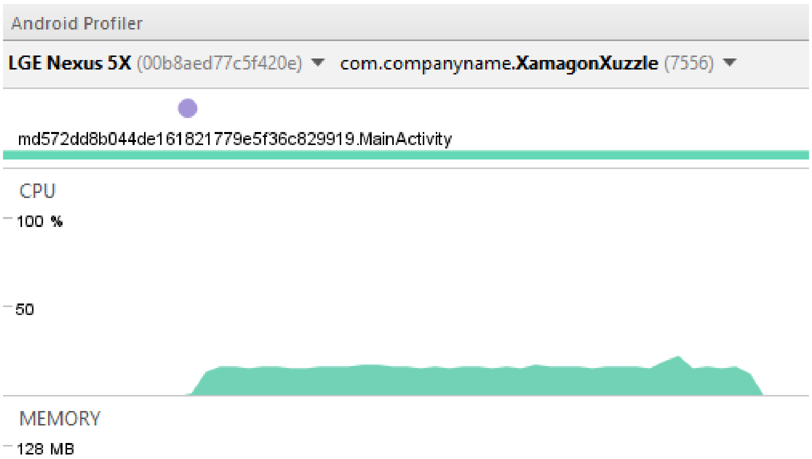
13. After several seconds, Android Profiler will complete APK installation and launch the APK, reporting the device name and the name of the app process being profiled (in this example, **LGE Nexus 5X** and **com.companyname.XamagonXuzzle**, respectively):



14. After the device and debuggable process are identified, Android Profiler begins profiling the app:



15. If you tap the **RANDOMIZE** button on **XamagonXuzzle** (which causes it to shift and randomize tiles), you will see the CPU usage increase during the app's randomization interval:



Using the Android Profiler

Detailed information for using the Android Profiler is included in the Android Studio documentation. The following topics will be of interest to Xamarin Android developers:

- CPU Profiler – Explains how to inspect the app's CPU usage and thread activity in real-time.
- Memory Profiler – Displays a real-time graph of the app's memory usage, and includes a button to record memory allocations for analysis.

- Network Profiler – Displays real-time network activity of data sent and received by the app.

Preparing an Application for Release

7/8/2021 • 17 minutes to read • Edit Online

After an application has been coded and tested, it is necessary to prepare a package for distribution. The first task in preparing this package is to build the application for release, which mainly entails setting some application attributes.

Use the following steps to build the app for release:

- **Specify the Application Icon** – Each Xamarin.Android application should have an application icon specified. Although not technically necessary, some markets, such as Google Play, require it.
- **Version the Application** – This step involves initializing or updating the versioning information. This is important for future application updates and to ensure that the users are aware of which version of the application they have installed.
- **Shrink the APK** – The size of the final APK can be substantially reduced by using the Xamarin.Android linker on the managed code and ProGuard on the Java bytecode.
- **Protect the Application** – Prevent users or attackers from debugging, tampering, or reverse engineering the application by disabling debugging, obfuscating the managed code, adding anti-debug and anti-tamper, and using native compilation.
- **Set Packaging Properties** – Packaging properties control the creation of the Android application package (APK). This step optimizes the APK, protects its assets, and modularizes the packaging as needed. Additionally, you can provide your users with an Android App Bundle that's optimized for their devices.
- **Compile** – This step compiles the code and assets to verify that it builds in Release mode.
- **Archive for Publishing** – This step builds the app and places it in an archive for signing and publishing.

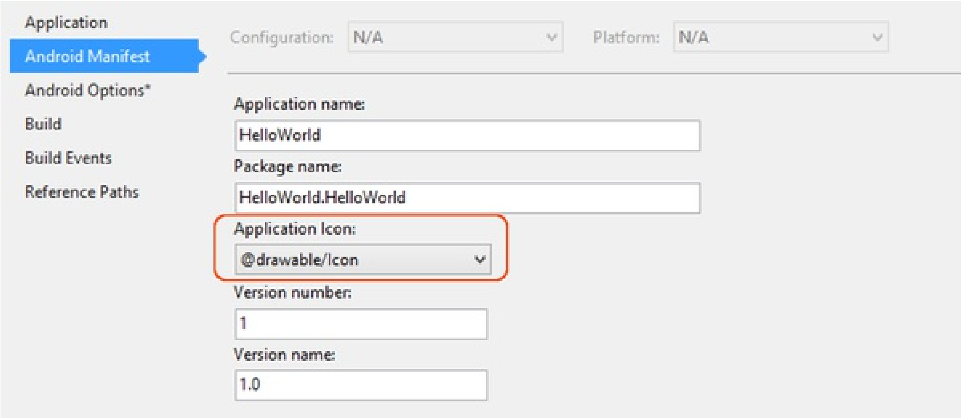
Each of these steps is described below in more detail.

Specify the Application Icon

It is strongly recommended that each Xamarin.Android application specifies an application icon. Some application marketplaces will not allow an Android application to be published without one. The `Icon` property of the `Application` attribute is used to specify the application icon for a Xamarin.Android project.

- Visual Studio
- Visual Studio for Mac

In Visual Studio 2017 and later, specify the application icon through the **Android Manifest** section of project **Properties**, as shown in the following screenshot:



In these examples, `@drawable/icon` refers to an icon file that is located at **Resources/drawable/icon.png** (note that the **.png** extension is not included in the resource name). This attribute can also be declared in the file **Properties\AssemblyInfo.cs**, as shown in this sample snippet:

```
[assembly: Application(Icon = "@drawable/icon")]
```

Normally, `using Android.App` is declared at the top of **AssemblyInfo.cs** (the namespace of the `Application` attribute is `Android.App`); however, you may need to add this `using` statement if it is not already present.

Version the Application

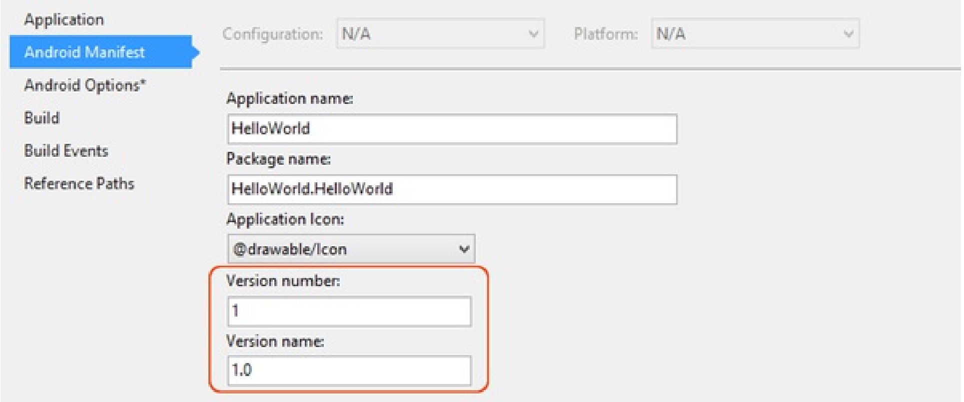
Versioning is important for Android application maintenance and distribution. Without some sort of versioning in place, it is difficult to determine if or how an application should be updated. To assist with versioning, Android recognizes two different types of information:

- **Version Number** – An integer value (used internally by Android and the application) that represents the version of the application. Most applications start out with this value set to 1, and then it is incremented with each build. This value has no relationship or affinity with the version name attribute (see below). Applications and publishing services should not display this value to users. This value is stored in the **AndroidManifest.xml** file as `android:versionCode`.

- **Version Name** – A string that is used only for communicating information to the user about the version of the application (as installed on a specific device). The version name is intended to be displayed to users or in Google Play. This string is not used internally by Android. The version name can be any string value that would help a user identify the build that is installed on their device. This value is stored in the **AndroidManifest.xml** file as `android:versionName`.

- Visual Studio
- Visual Studio for Mac

In Visual Studio, these values can be set in the **Android Manifest** section of project **Properties**, as shown in the following screenshot:



Shrink the APK

Xamarin.Android APKs can be made smaller through a combination of the Xamarin.Android linker, which removes unnecessary *managed* code, and the *ProGuard* tool from the Android SDK, which removes unused *Java bytecode*. The build process first uses the Xamarin.Android linker to optimize the app at the managed code (C#) level, and then it later uses ProGuard (if enabled) to optimize the APK at the Java bytecode level.

Configure the Linker

Release mode turns off the shared runtime and turns on linking so that the application only ships the pieces of Xamarin.Android required at runtime. The *linker* in Xamarin.Android uses static analysis to determine which assemblies, types, and type members are used or referenced by a Xamarin.Android application. The linker then discards all the unused assemblies, types, and members that are not used (or referenced). This can result in a significant reduction in the package size. For example, consider the HelloWorld sample, which experiences an 83% reduction in the final size of its APK:

- Configuration: None – Xamarin.Android 4.2.5 Size = 17.4 MB.
- Configuration: SDK Assemblies Only – Xamarin.Android 4.2.5 Size = 3.0 MB.

- Visual Studio
- Visual Studio for Mac

Set linker options through the **Android Options** section of the project **Properties**:

The **Linking** pull-down menu provides the following options for controlling the linker:

- **None** – This turns off the linker; no linking will be performed.
- **SDK Assemblies Only** – This will only link the assemblies that are required by Xamarin.Android. Other assemblies will not be linked.
- **Sdk and User Assemblies** – This will link all assemblies that are required by the application, and not just the ones required by Xamarin.Android.

Linking can produce some unintended side effects, so it is important that an application be re-tested in Release mode on a physical device.

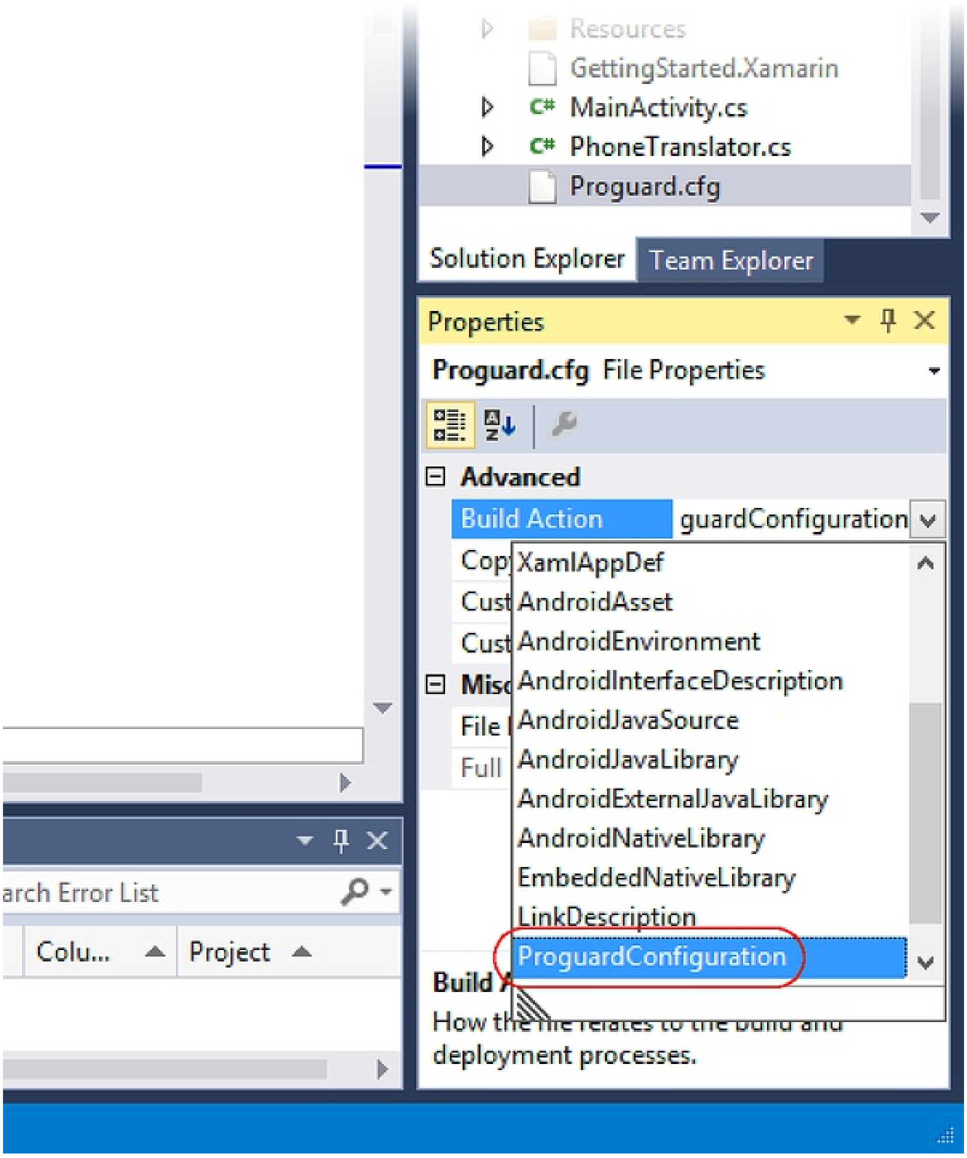
ProGuard

ProGuard is an Android SDK tool that links and obfuscates Java code. ProGuard is normally used to create smaller applications by reducing the footprint of large included libraries (such as Google Play Services) in your APK. ProGuard removes unused Java bytecode, which makes the resulting app smaller. For example, using ProGuard on small Xamarin.Android apps usually achieves about a 24% reduction in size – using ProGuard on larger apps with multiple library dependencies typically achieves an even greater size reduction.

ProGuard is not an alternative to the Xamarin.Android linker. The Xamarin.Android linker links *managed* code, while ProGuard links Java bytecode. The build process first uses the Xamarin.Android linker to optimize the managed (C#) code in the app, and then it later uses ProGuard (if enabled) to optimize the APK at the Java bytecode level.

When **Enable ProGuard** is checked, Xamarin.Android runs the ProGuard tool on the resulting APK. A ProGuard configuration file is generated and used by ProGuard at build time. Xamarin.Android also supports custom *ProguardConfiguration* build actions. You can add a custom ProGuard configuration file to your project, right-click it, and select it as a build action as shown in this example:

- Visual Studio
- Visual Studio for Mac



ProGuard is disabled by default. The **Enable ProGuard** option is available only when the project is set to **Release** mode. All ProGuard build actions are ignored unless **Enable ProGuard** is checked. The Xamarin.Android ProGuard configuration does not obfuscate the APK, and it is not possible to enable obfuscation, even with custom configuration files. If you wish to use obfuscation, please see Application Protection with Dotfuscator.

For more detailed information about using the ProGuard tool, see ProGuard.

Protect the Application

Disable Debugging

During development of an Android application, debugging is performed with the use of the *Java Debug Wire Protocol* (JDWP). This is a technology that allows tools such as **adb** to communicate with a JVM for the purposes of debugging. JDWP is turned on by default for Debug builds of a Xamarin.Android application. While JDWP is important during development, it can pose a security issue for released applications.

> **IMPORTANT**
> Always disable the debug state in a released application as it is possible (via JDWP) to gain full access to the Java process and execute arbitrary code in the context of the application if this debug state is not disabled.

The Android Manifest contains the `android:debuggable` attribute, which controls whether or not the application may be debugged. It is considered a good practice to set the `android:debuggable` attribute to `false`. The simplest way to do this is by adding a conditional compile statement in **AssemblyInfo.cs**:

```
#if DEBUG
[assembly: Application(Debuggable=true)]
#else
[assembly: Application(Debuggable=false)]
#endif
```

Note that Debug builds automatically set some permissions to make debug easier (such as **Internet** and **ReadExternalStorage**). Release builds, however, use only the permissions that you explicitly configure. If you find that switching to the Release build causes your app to lose a permission that was available in the Debug build, verify that you have explicitly enabled this permission in the **Required permissions** list as described in Permissions.

Application Protection with Dotfuscator

- Visual Studio
- Visual Studio for Mac

Even with debugging disabled, it is still possible for attackers to re-package an application, adding or removing configuration options or permissions. This allows them to reverse-engineer, debug, or tamper with the application. Dotfuscator Community Edition (CE) can be used to obfuscate managed code and inject runtime security state detection code into a Xamarin.Android app at build time to detect and respond if the app is running on a rooted device.

Dotfuscator CE is included with Visual Studio 2017. To use Dotfuscator, click **Tools > PreEmptive Protection - Dotfuscator**.

To configure Dotfuscator CE, please see Using Dotfuscator Community Edition with Xamarin. Once it is configured, Dotfuscator CE will automatically protect each build that is created.

Bundle Assemblies into Native Code

When this option is enabled, assemblies are bundled into a native shared library. This allows assemblies to be compressed, permitting smaller `.apk` files. Assembly compression also confers a *minimal* form of obfuscation; such obfuscation should not be relied upon.

This option requires an Enterprise license and is only available when **Use Fast Deployment** is disabled. **Bundle assemblies into native code** is disabled by default.

Note that the **Bundle into Native Code** option does *not* mean that the assemblies are compiled into native code. It is not possible to use AOT Compilation to compile assemblies into native code.

AOT Compilation

The **AOT Compilation** option (on the Packaging Properties page) enables Ahead-of-Time (AOT) compilation of assemblies. When this option is enabled, Just In Time (JIT) startup overhead is minimized by precompiling assemblies before runtime. The resulting native code is included in the APK along with the uncompiled assemblies. This results in shorter application startup time, but at the expense of slightly larger APK sizes.

The **AOT Compilation** option requires an Enterprise license or higher. **AOT compilation** is available only when the project is configured for Release mode, and it is disabled by default. For more information about AOT Compilation, see AOT.

LLVM Optimizing Compiler

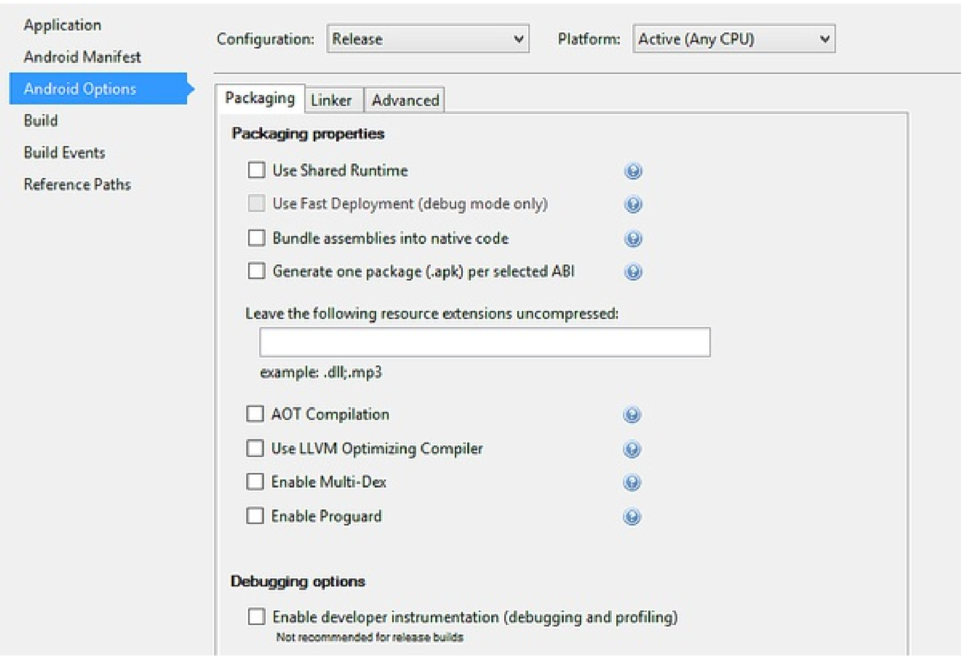
The *LLVM Optimizing Compiler* will create smaller and faster compiled code and convert AOT-compiled assemblies into native code, but at the expense of slower build times. The LLVM compiler is disabled by default. To use the LLVM compiler, the **AOT Compilation** option must first be enabled (on the Packaging Properties page).

> **NOTE**
> The **LLVM Optimizing Compiler** option requires an Enterprise license.

Set Packaging Properties

- Visual Studio
- Visual Studio for Mac

Packaging properties can be set in the **Android Options** section of project **Properties**, as shown in the following screenshot:



Many of these properties, such as **Use Shared Runtime**, and **Use Fast Deployment** are intended for Debug mode. However, when the application is configured for Release mode, there are other settings that determine how the app is optimized for size and execution speed, how it is protected from tampering, and how it can be packaged to support different architectures and size restrictions.

Specify Supported Architectures

When preparing a Xamarin.Android app for release, it is necessary to specify the CPU architectures that are supported. A single APK can contain machine code to support multiple, different architectures. See CPU Architectures for details about supporting multiple CPU architectures.

Generate One Package (.APK) per Selected ABI

When this option is enabled, one APK will be created for each of the supported ABI's (selected on the **Advanced** tab, as described in CPU Architectures) rather than a single, large APK for all supported ABI's. This option is available only when the project is configured for Release mode, and it is disabled by default.

Multi-Dex

When the **Enable Multi-Dex** option is enabled, Android SDK tools are used to bypass the 65K method limit of the **.dex** file format. The 65K method limitation is based on the number of Java methods that an app *references* (including those in any libraries that the app depends on) – it is not based on the number of methods that are *written in the source code*. If an application only defines a few methods but uses many (or large libraries), it is possible that the 65K limit will be exceeded.

It is possible that an app is not using every method in every library that is referenced; therefore, it is possible that a tool such as ProGuard (see above) can remove the unused methods from code. The best practice is to enable **Enable Multi-Dex** only if absolutely necessary, i.e.the app still references more than 65K Java methods even after using ProGuard.

For more information about Multi-Dex, see Configure Apps with Over 64K Methods.

Android App Bundles

App bundles differ from APKs as they cannot be deployed directly to a device. Rather, it's a format that is intended to be uploaded with all of your compiled code and resources. After you upload your signed app bundle, Google Play will have everything it needs to build and sign your application's APKs and serve them to your users using Dynamic Delivery.

To enable support for Android App Bundles, you'll need to opt-in to the `bundle` value of the **Android Package Format** property within your Android project options. Before you do this, ensure you change your project to a `Release` configuration as app bundles are intended for release packages only.

You can now generate an app bundle by following the Archive Flow. This will generate an app bundle for your application.

For more information about Android App Bundles, see Android App Bundles.

Compile

- Visual Studio
- Visual Studio for Mac

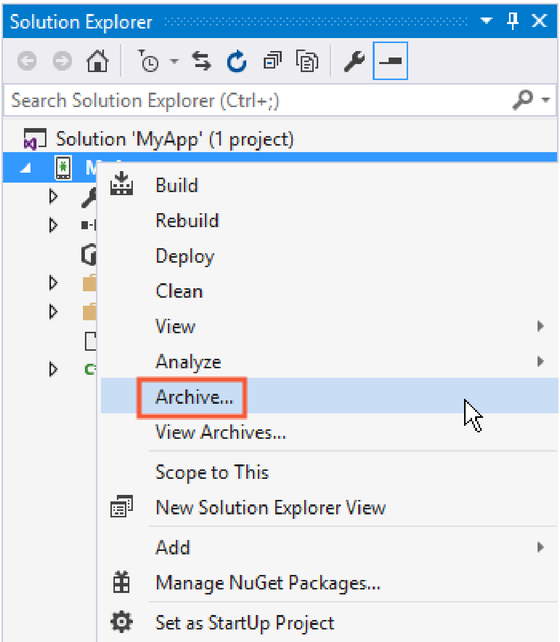
After all of the above steps are completed, the app is ready for compilation. Select **Build > Rebuild Solution** to verify that it builds successfully in Release mode. Note that this step does not yet produce an APK.

Signing the App Package discusses packaging and signing in more detail.

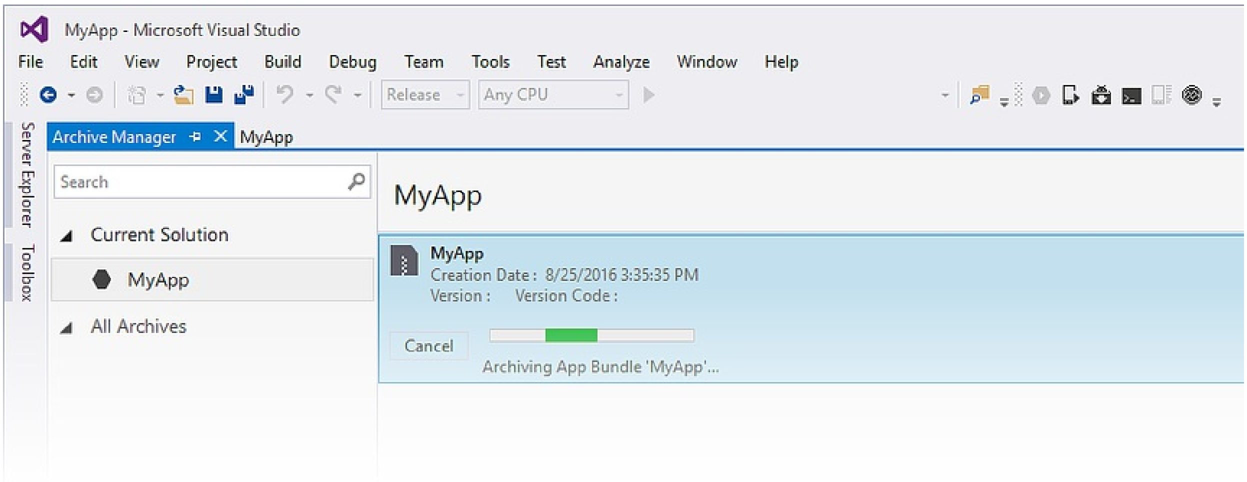
Archive for Publishing

- [Visual Studio]
- [Visual Studio for Mac]

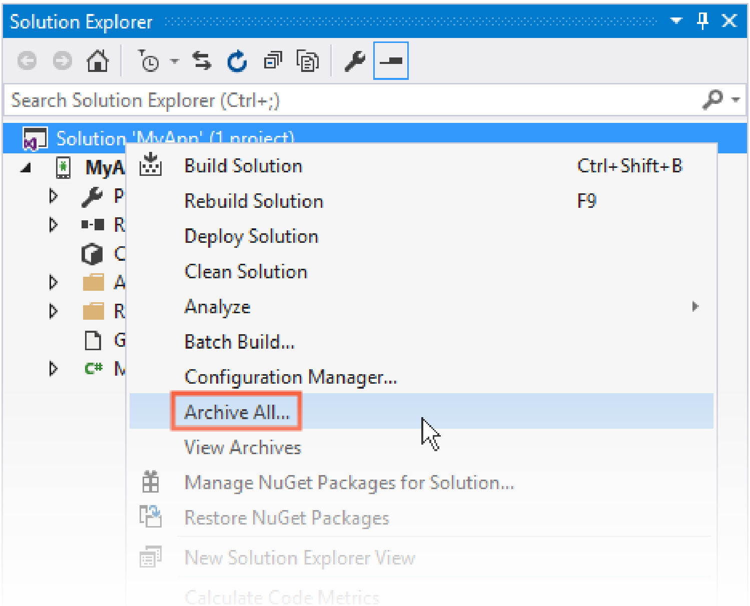
To begin the publishing process, right-click the project in **Solution Explorer** and select the **Archive...** context menu item:



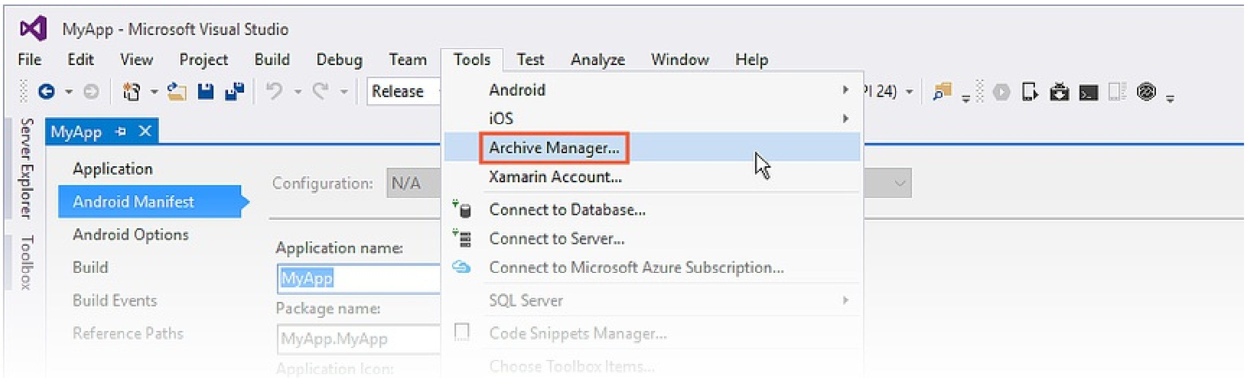
Archive... launches the **Archive Manager** and begins the process of archiving the App bundle as shown in this screenshot:



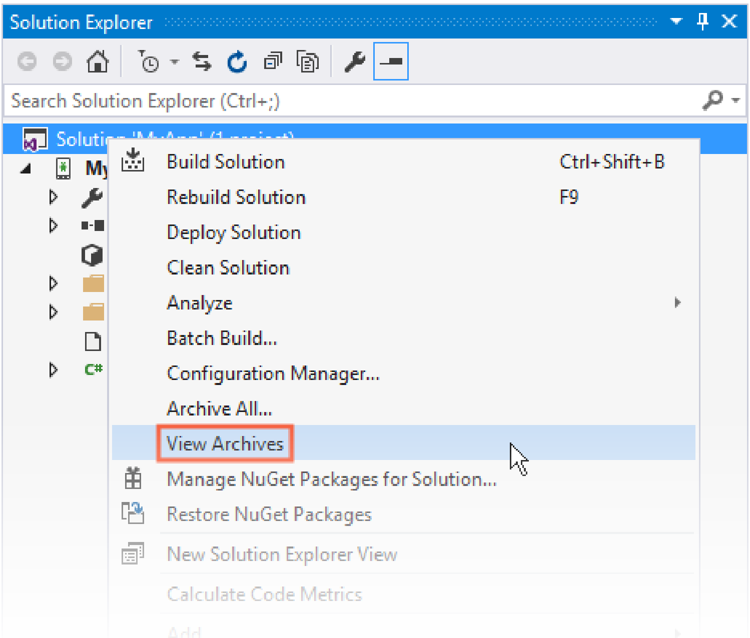
Another way to create an archive is to right-click the Solution in the **Solution Explorer** and select **Archive All...**, which builds the solution and archives all Xamarin projects that can generate an archive:



Both **Archive** and **Archive All** automatically launch the **Archive Manager**. To launch the **Archive Manager** directly, click the **Tools > Archive Manager...** menu item:

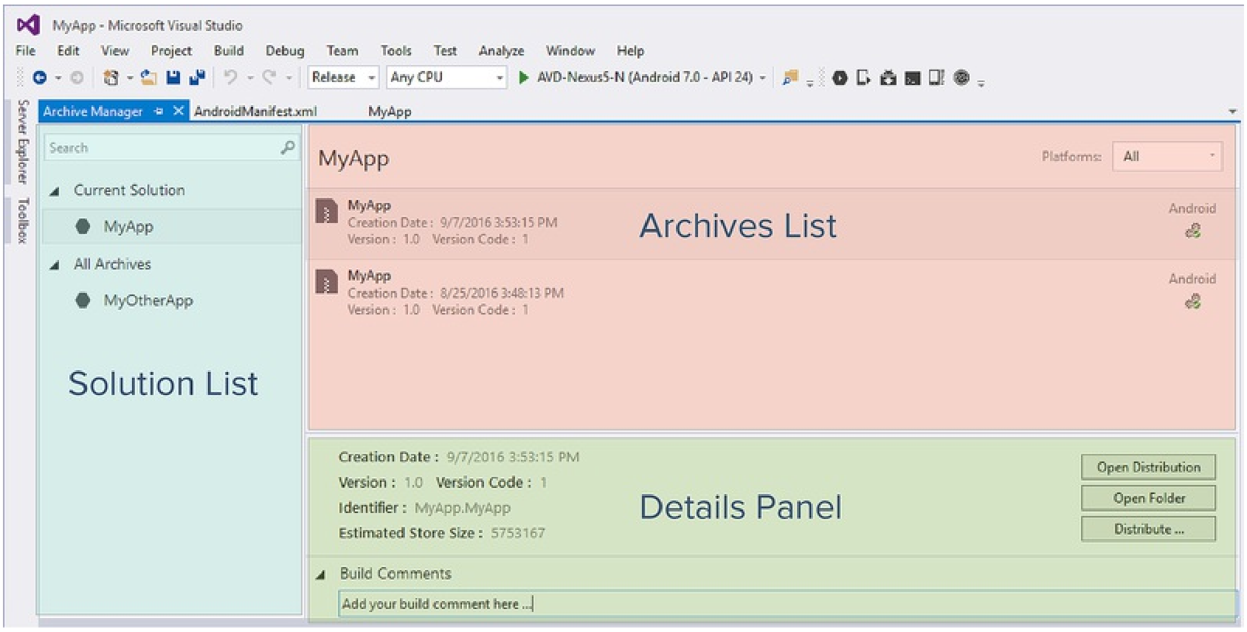


The solution's archives at any time by right clicking the **Solution** node and selecting **View Archives**:



The Archive Manager

The **Archive Manager** is comprised of a **Solution List** pane, an **Archives List**, and a **Details Panel**:



The **Solution List** displays all solutions having at least one archived project. The **Solution List** includes the following sections:

- **Current Solution** – Displays the current solution. Note that this area may be empty if the current solution does not have an existing archive.
- **All Archives** – Displays all solutions that have an archive.
- **Search** text box (at the top) – Filters the solutions listed in the **All Archives** list according to the search string entered in the text box.

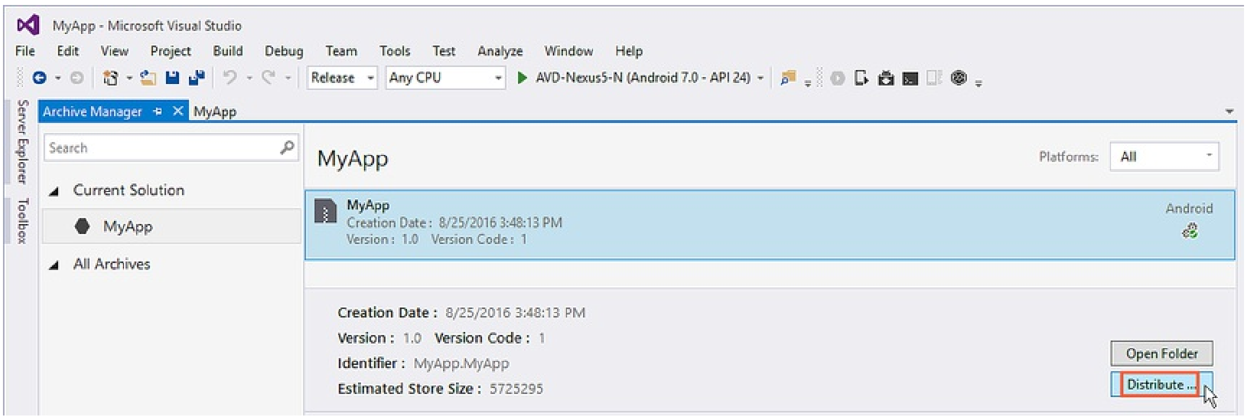
The **Archives List** displays the list of all archives for the selected solution. The **Archives List** includes the following sections:

- **Selected solution name** – Displays the name of the solution selected in the **Solution List**. All information shown in the **Archives List** refers to this selected solution.
- **Platforms Filter** – This field makes it possible to filter archives by platform type (such as iOS or Android).
- **Archive Items** – List of archives for the selected solution. Each item in this list includes the project name, creation date, and platform. It can also show additional information such as the progress when an item is being archived or published.

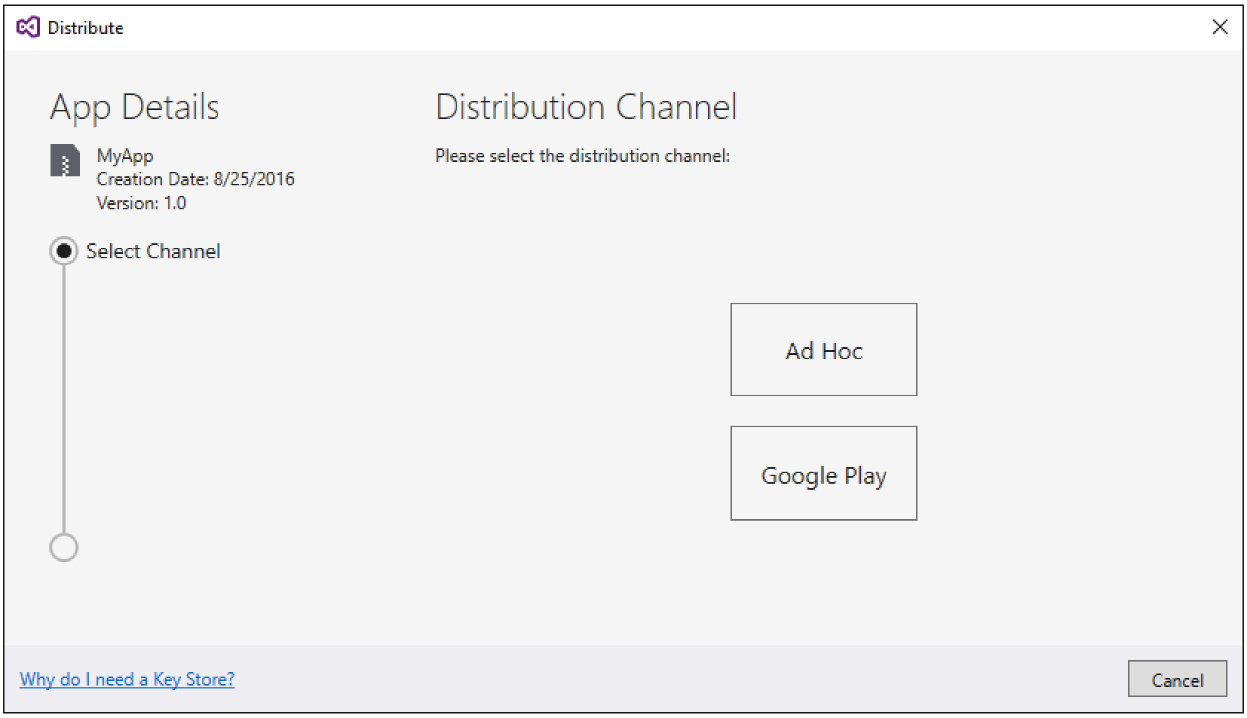
The **Details Panel** displays additional information about each archive. It also allows the user to start the Distribution workflow or open the folder where the distribution has been created. The **Build Comments** section makes it possible to include build comments in the archive.

Distribution

When an archived version of the application is ready to publish, select the archive in the **Archive Manager** and click the **Distribute...** button:



The **Distribution Channel** dialog shows information about the app, an indication of distribution workflow progress, and a choice of distribution channels. On the first run, two choices are presented:



It is possible to choose one of the following distribution channels:

- **Ad-Hoc** – Saves a signed APK to disk that can be sideloaded to Android devices. Continue to Signing the App Package to learn how to create an Android signing identity, create a new signing certificate for Android applications, and publish an *ad hoc* version of the app to disk. This is a good way to create an APK for testing.
- **Google Play** – Publishes a signed APK to Google Play. Continue to Publishing to Google Play to learn how to sign and publish an APK in the Google Play store.

Related Links

- Multi-Core Devices and Xamarin.Android
- CPU Architectures
- AOT
- Shrink Your Code and Resources
- Configure Apps with Over 64K Methods

ProGuard

7/8/2021 • 8 minutes to read • Edit Online

Xamarin.Android ProGuard is a Java class file shrinker, optimizer, and pre-verifier. It detects and removes unused code, analyzes and optimizes bytecode. This guide explains how ProGuard works, how to enable it in your project, and how to configure it. It also provides several examples of ProGuard configurations.

Overview

ProGuard detects and removes unused classes, fields, methods, and attributes from your packaged application. It can even do the same for referenced libraries (this can help you avoid the 64k reference limit). The ProGuard tool from the Android SDK will also optimize bytecode and remove unused code instructions. ProGuard reads **input jars** and then shrinks, optimizes, and pre-verifies them; it writes the results to one or more **output jars**.

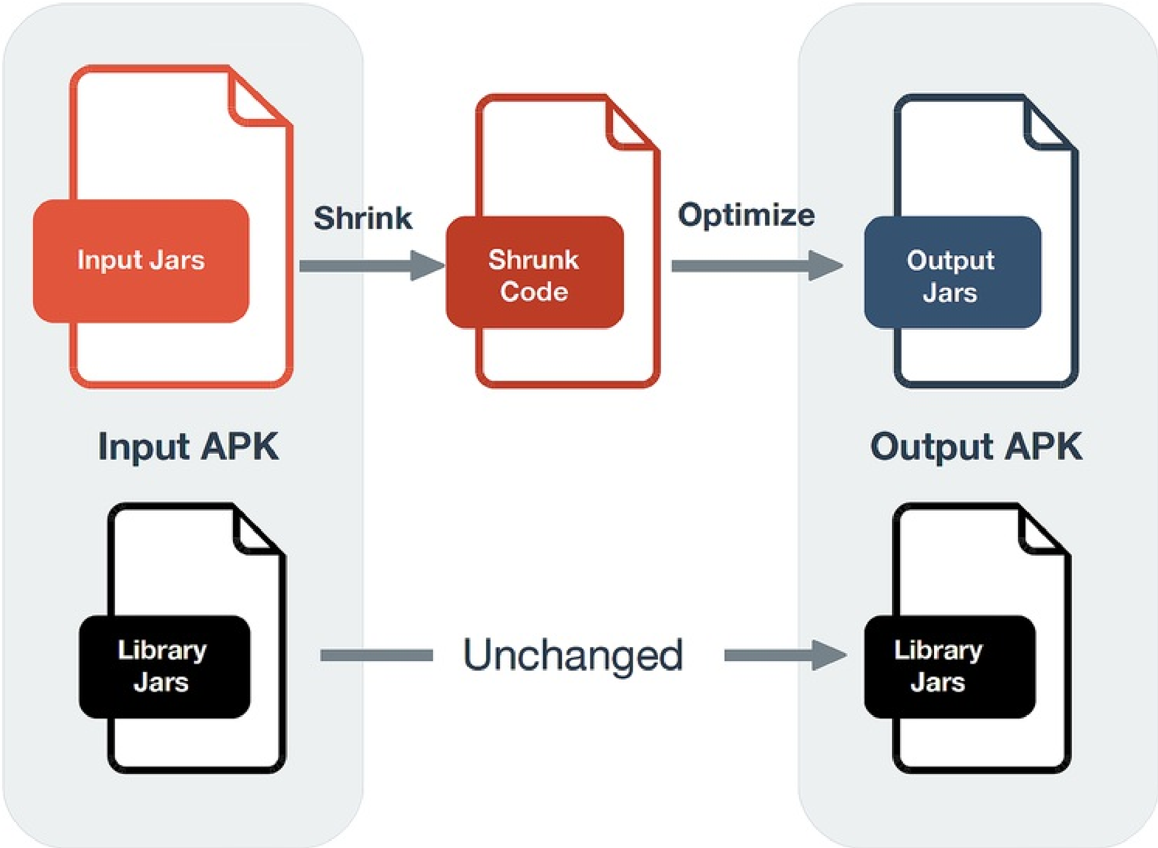
ProGuard processes input APK's using the following steps:

1. **Shrinking step** – ProGuard recursively determines which classes and class members are used. All other classes and class members are discarded.
2. **Optimization step** – ProGuard further optimizes the code. Among other optimizations, classes and methods that are not entry points can be made private, static, or final, unused parameters can be removed, and some methods may be inlined.
3. **Obfuscation step** – In native Android development, ProGuard renames classes and class members that are not entry points. Retaining entry points ensures that they can still be accessed by their original names. However, this step is not supported by Xamarin.Android because the app is compiled down to Intermediate Language (IL).
4. **Preverification step** – Performs checks on Java bytecodes ahead of runtime and annotates class files for the benefit of the Java VM. This is the only step that doesn't have to know the entry points.

Each of these steps is *optional*. As will be explained in the next section, Xamarin.Android ProGuard uses only a subset of these steps.

ProGuard in Xamarin.Android

The Xamarin.Android ProGuard configuration does not obfuscate the APK. In fact, it is not possible to enable obfuscation through ProGuard (even through the use of custom configuration files). Thus, Xamarin.Android's ProGuard performs only the **shrinking** and **optimization** steps:



One important item to know in advance before using ProGuard is how it works within the `Xamarin.Android` build process. This process uses two separate steps:

1. Xamarin Android Linker
2. ProGuard

Each of these steps is described next.

Linker Step

The Xamarin.Android linker employs static analysis of your application to determine the following:

- Which assemblies are actually used.
- Which types are actually used.
- Which members are actually used.

The linker will always run before the ProGuard step. Because of this, the linker can strip an assembly/type/member that you might expect ProGuard to run on. (For more information about linking in Xamarin.Android, see Linking on Android.)

ProGuard Step

After the linker step completes successfully, ProGuard is run to remove unused Java bytecode. This is the step that optimizes the APK.

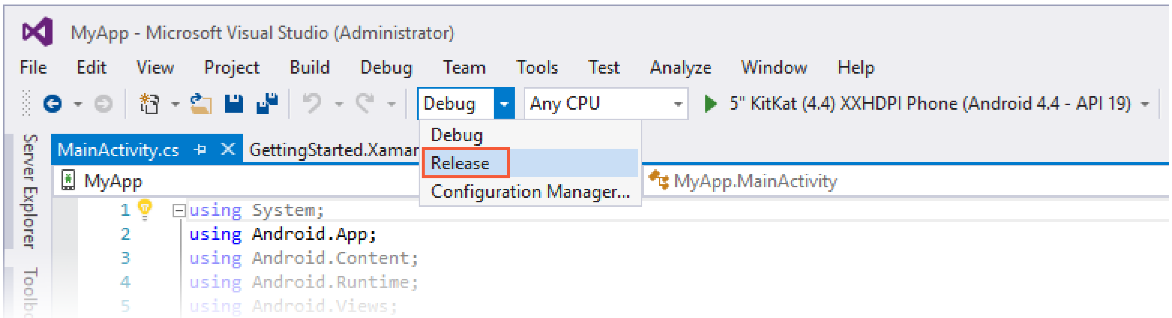
Using ProGuard

To use ProGuard in your app project, you must first enable ProGuard. Next, you can either let the Xamarin.Android build process use a default ProGuard configuration file, or you can create your own custom configuration file for ProGuard to use.

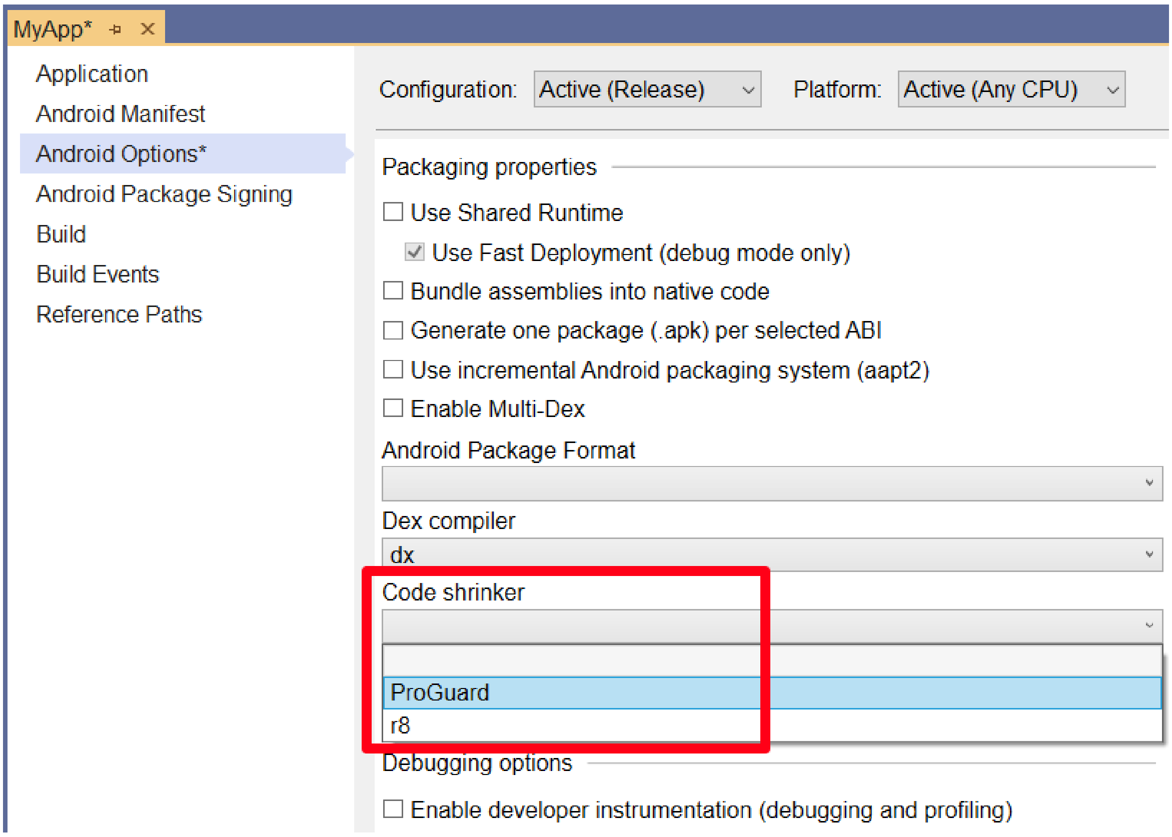
Enabling ProGuard

Use the following steps to enable ProGuard in your app project:

1. Ensure that your project is set to the **Release** configuration (this is important because the linker must run in order for ProGuard to run):



2. Choose **ProGuard** from the **Code shrinker** drop-down list on the **Properties > Android Options** window:



For most Xamarin.Android apps, the default ProGuard configuration file supplied by Xamarin.Android will be sufficient to remove all (and only) unused code. To view the default ProGuard configuration, open the file at **obj\Release\proguard\proguard_xamarin.cfg**.

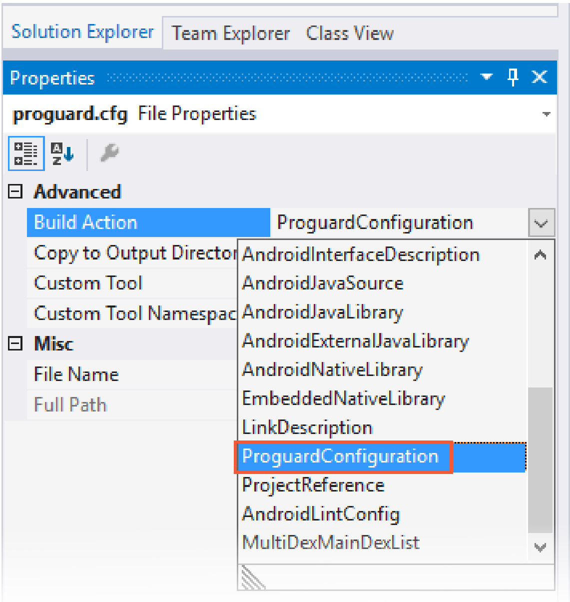
The following example illustrates a typical generated **proguard_xamarin.cfg** file:

```
# This is Xamarin-specific (and enhanced) configuration.

-dontobfuscate

-keep class mono.MonoRuntimeProvider { *; <init>(...); }
-keep class mono.MonoPackageManager { *; <init>(...); }
-keep class mono.MonoPackageManager_Resources { *; <init>(...); }
-keep class mono.android.** { *; <init>(...); }
-keep class mono.java.** { *; <init>(...); }
-keep class mono.javax.** { *; <init>(...); }
-keep class opentk.platform.android.AndroidGameView { *; <init>(...); }
-keep class opentk.GameViewBase { *; <init>(...); }
-keep class opentk_1_0.platform.android.AndroidGameView { *; <init>(...); }
-keep class opentk_1_0.GameViewBase { *; <init>(...); }

-keep class android.runtime.** { <init>(***); }
-keep class assembly_mono_android.android.runtime.** { <init>(***); }
# hash for android.runtime and assembly_mono_android.android.runtime.
-keep class md52ce486a14f4bcd95899665e9d932190b.** { *; <init>(...); }
-keepclassmembers class md52ce486a14f4bcd95899665e9d932190b.** { *; <init>(...); }

# Android's template misses fluent setters...
-keepclassmembers class * extends android.view.View {
   *** set*(***);
}

# also misses those inflated custom layout stuff from xml...
-keepclassmembers class * extends android.view.View {
   <init>(android.content.Context,android.util.AttributeSet);
   <init>(android.content.Context,android.util.AttributeSet,int);
}
```

The next section describes how to create a customized ProGuard configuration file.

Customizing ProGuard

Optionally, you can add a custom ProGuard Configuration file to exert more control over the ProGuard tooling. For example, you may want to explicitly tell ProGuard which classes to keep. To do this, create a new **.cfg** file and apply the `ProGuardConfiguration` build action in the **Properties** pane of the **Solution Explorer**:



Keep in mind that this configuration file does not replace the Xamarin.Android **proguard_xamarin.cfg** file since both are used by ProGuard.

There might be cases where ProGuard is unable to properly analyze your application; it could potentially remove code that your application actually needs. If this happens, you can add a `-keep` line to your custom ProGuard configuration file:

```
-keep public class MyClass
```

In this example, `MyClass` is set to the actual name of the class that you want ProGuard to skip.

You can also register your own names with `[Register]` annotations and use these names to customize ProGuard rules. You can register names for Adapters, Views, BroadcastReceivers, Services, ContentProviders, Activities, and Fragments. For more information about using the `[Register]` custom attribute, see Working with JNI.

ProGuard Options

ProGuard offers a number of options that you can configure to provide finer control over its operation. The ProGuard Manual provides complete reference documentation for the use of ProGuard.

Xamarin.Android supports the following ProGuard options:

- Input/Output Options
- Keep Options
- Shrinking Options
- General Options
- Class Paths
- File Names
- File Filters
- Filters
- Overview of `Keep` Options
- Keep Option Modifiers
- Class Specifications

The following options are *ignored* by Xamarin.Android:

- Optimization Options
- Obfuscation Options
- Preverification Options

ProGuard and Android Nougat

If you are trying to use ProGuard against Android 7.0 or later, you must download a newer version of ProGuard because the Android SDK does not ship a new version that is compatible with JDK 1.8.

You can use this NuGet package to install a newer version of `proguard.jar`. For more information about updating the default Android SDK `proguard.jar`, see this Stack Overflow discussion.

You can find all versions of ProGuard at the SourceForge page.

Example ProGuard Configurations

Two example ProGuard configuration files are listed below. Please note that, in these cases, the Xamarin.Android build process will supply the **input**, **output**, and **library** jars. Thus, you can focus on other options like `-keep`.

A simple Android activity

The following example illustrates the configuration for a simple Android activity:

```
-injars  bin/classes
-outjars bin/classes-processed.jar
-libraryjars /usr/local/java/android-sdk/platforms/android-9/android.jar

-dontpreverify
-repackageclasses ''
-allowaccessmodification
-optimizations !code/simplification/arithmetic

-keep public class mypackage.MyActivity
```

A complete Android application

The following example illustrates the configuration for a complete Android app:

```
-injars  bin/classes
-injars  libs
-outjars bin/classes-processed.jar
-libraryjars /usr/local/java/android-sdk/platforms/android-9/android.jar

-dontpreverify
-repackageclasses ''
-allowaccessmodification
-optimizations !code/simplification/arithmetic
-keepattributes *Annotation*

-keep public class * extends android.app.Activity
-keep public class * extends android.app.Application
-keep public class * extends android.app.Service
-keep public class * extends android.content.BroadcastReceiver
-keep public class * extends android.content.ContentProvider

-keep public class * extends android.view.View {
public <init>(android.content.Context);
public <init>(android.content.Context, android.util.AttributeSet);
public <init>(android.content.Context, android.util.AttributeSet, int);
public void set*(...);
}

-keepclasseswithmembers class * {
public <init>(android.content.Context, android.util.AttributeSet);
}

-keepclasseswithmembers class * {
public <init>(android.content.Context, android.util.AttributeSet, int);
}

-keepclassmembers class * implements android.os.Parcelable {
static android.os.Parcelable$Creator CREATOR;
}

-keepclassmembers class **.R$* {
public static <fields>;
}
```

ProGuard and the Xamarin.Android Build Process

The following sections explain how ProGuard runs during a Xamarin.Android **Release** build.

What command is ProGuard running?

ProGuard is simply a `.jar` provided with the Android SDK. Thus, it is invoked in a command:

```
java -jar proguard.jar options ...
```

The ProGuard Task

The ProGuard task is found inside the **Xamarin.Android.Build.Tasks.dll** assembly. It is part of the `_CompileToDalvikWithDx` target, which is a part of the `_CompileDex` target.

The following listing provides an example of the default parameters that are generated after you a create a new project using **File > New Project**:

```
ProGuardJarPath = C:\Android\android-sdk\tools\proguard\lib\proguard.jar
AndroidSdkDirectory = C:\Android\android-sdk\
JavaToolPath = C:\Program Files (x86)\Java\jdk1.8.0_92\\bin
ProGuardToolPath = C:\Android\android-sdk\tools\proguard\
JavaPlatformJarPath = C:\Android\android-sdk\platforms\android-25\android.jar
ClassesOutputDirectory = obj\Release\android\bin\classes
AcwMapFile = obj\Release\acw-map.txt
ProGuardCommonXamarinConfiguration = obj\Release\proguard\proguard_xamarin.cfg
ProGuardGeneratedReferenceConfiguration = obj\Release\proguard\proguard_project_references.cfg
ProGuardGeneratedApplicationConfiguration = obj\Release\proguard\proguard_project_primary.cfg
ProGuardConfigurationFiles

    {sdk.dir}tools\proguard\proguard-android.txt;
    {intermediate.common.xamarin};
    {intermediate.references};
    {intermediate.application};
    ;

JavaLibrariesToEmbed = C:\Program Files (x86)\Reference
Assemblies\Microsoft\Framework\MonoAndroid\v7.0\mono.android.jar
ProGuardJarInput = obj\Release\proguard\__proguard_input__.jar
ProGuardJarOutput = obj\Release\proguard\__proguard_output__.jar
DumpOutput = obj\Release\proguard\dump.txt
PrintSeedsOutput = obj\Release\proguard\seeds.txt
PrintUsageOutput = obj\Release\proguard\usage.txt
PrintMappingOutput = obj\Release\proguard\mapping.txt
```

The next example illustrates a typical ProGuard command that is run from the IDE:

```
C:\Program Files (x86)\Java\jdk1.8.0_92\\bin\java.exe -jar C:\Android\android-
sdk\tools\proguard\lib\proguard.jar -include obj\Release\proguard\proguard_xamarin.cfg -include
obj\Release\proguard\proguard_project_references.cfg -include
obj\Release\proguard\proguard_project_primary.cfg "-injars
'obj\Release\proguard\__proguard_input__.jar';'C:\Program Files (x86)\Reference
Assemblies\Microsoft\Framework\MonoAndroid\v7.0\mono.android.jar'" "-libraryjars 'C:\Android\android-
sdk\platforms\android-25\android.jar'" -outjars "obj\Release\proguard\__proguard_output__.jar" -
optimizations !code/allocation/variable
```

Troubleshooting

File Issues

The following error message may be displayed when ProGuard reads its configuration file:

```
Unknown option '-keep' in line 1 of file 'proguard.cfg'
```

This issue typically happens on Windows because the `.cfg` file has the wrong encoding. ProGuard cannot handle *byte order mark* (BOM) which may be present in text files. If a BOM is present, then ProGuard will exit with the above error.

- Visual Studio
- Visual Studio for Mac

To prevent this problem, edit the custom configuration file from a text editor that will allow the file to be saved without a BOM. To solve this problem, ensure that your text editor has its encoding set to `UTF-8`. For example, the text editor Notepad++ can save files without the BOM by selecting the **Encoding > Encode in UTF-8 Without BOM** when saving the file.

Other Issues

The ProGuard Troubleshooting page discusses common issues you may encounter (and solutions) when using ProGuard.

Summary

This guide explained how ProGuard works in Xamarin.Android, how to enable it in your app project, and how to configure it. It provided example ProGuard configurations, and it described solutions to common problems. For more information about the ProGuard tool and Android, see Shrink Your Code and Resources.

Related Links

- Preparing an Application for Release

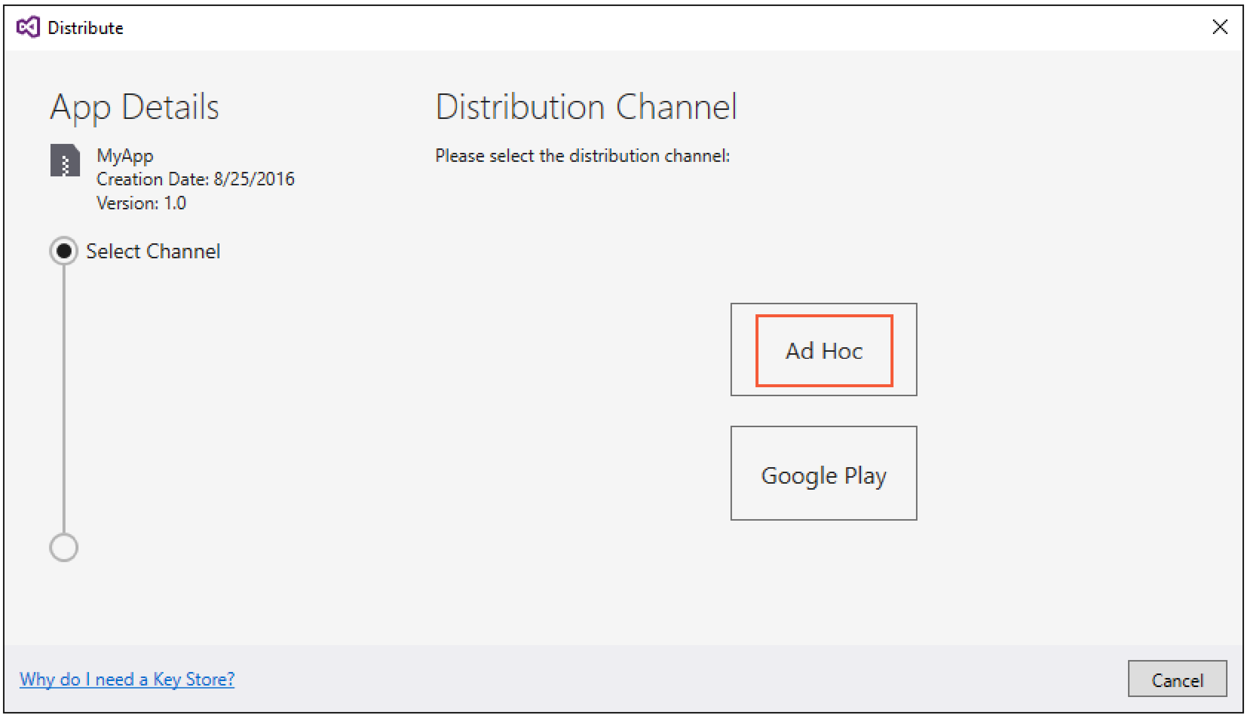
Signing the Android Application Package

7/8/2021 • 4 minutes to read • Edit Online

In Preparing an App for Release the **Archive Manager** was used to build the app and place it in an archive for signing and publishing. This section explains how to create an Android signing identity, create a new signing certificate for Android applications, and publish the archived app *ad hoc* to disk. The resulting APK can be sideloaded into Android devices without going through an app store.

- Visual Studio
- Visual Studio for Mac

In Archive for Publishing, the **Distribution Channel** dialog presents two choices for distribution. Select **Ad-Hoc**:

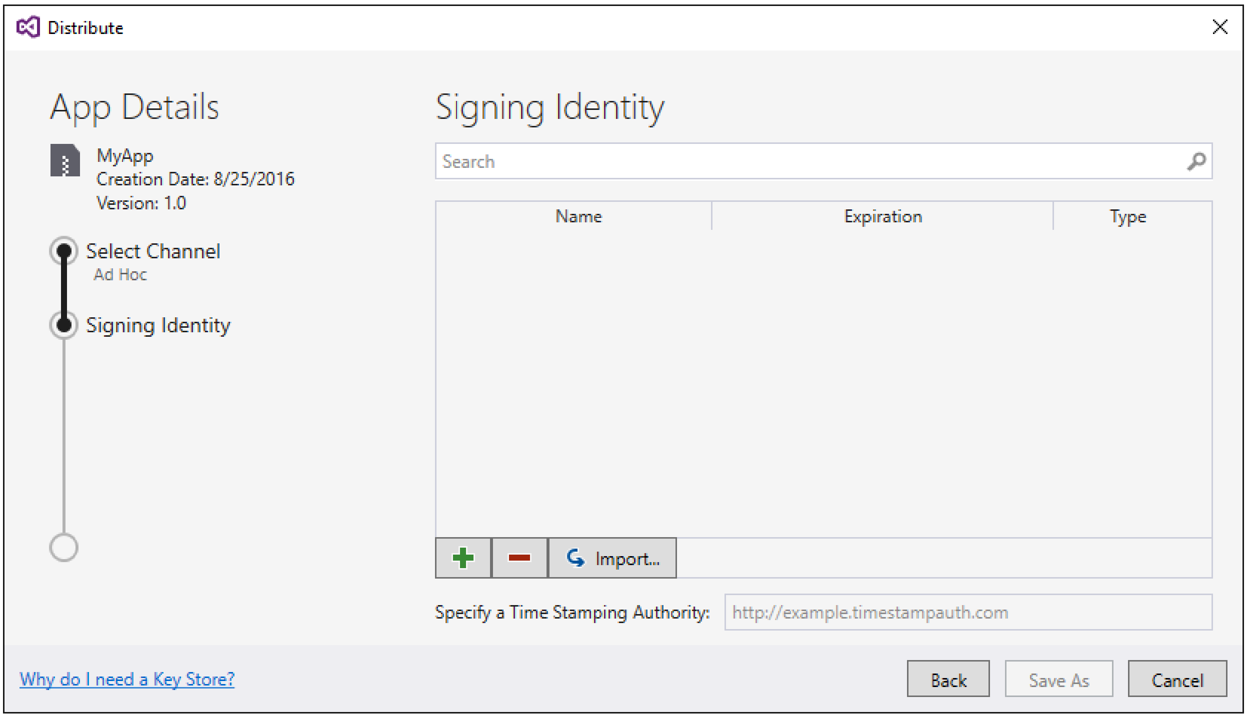


Create a New Certificate

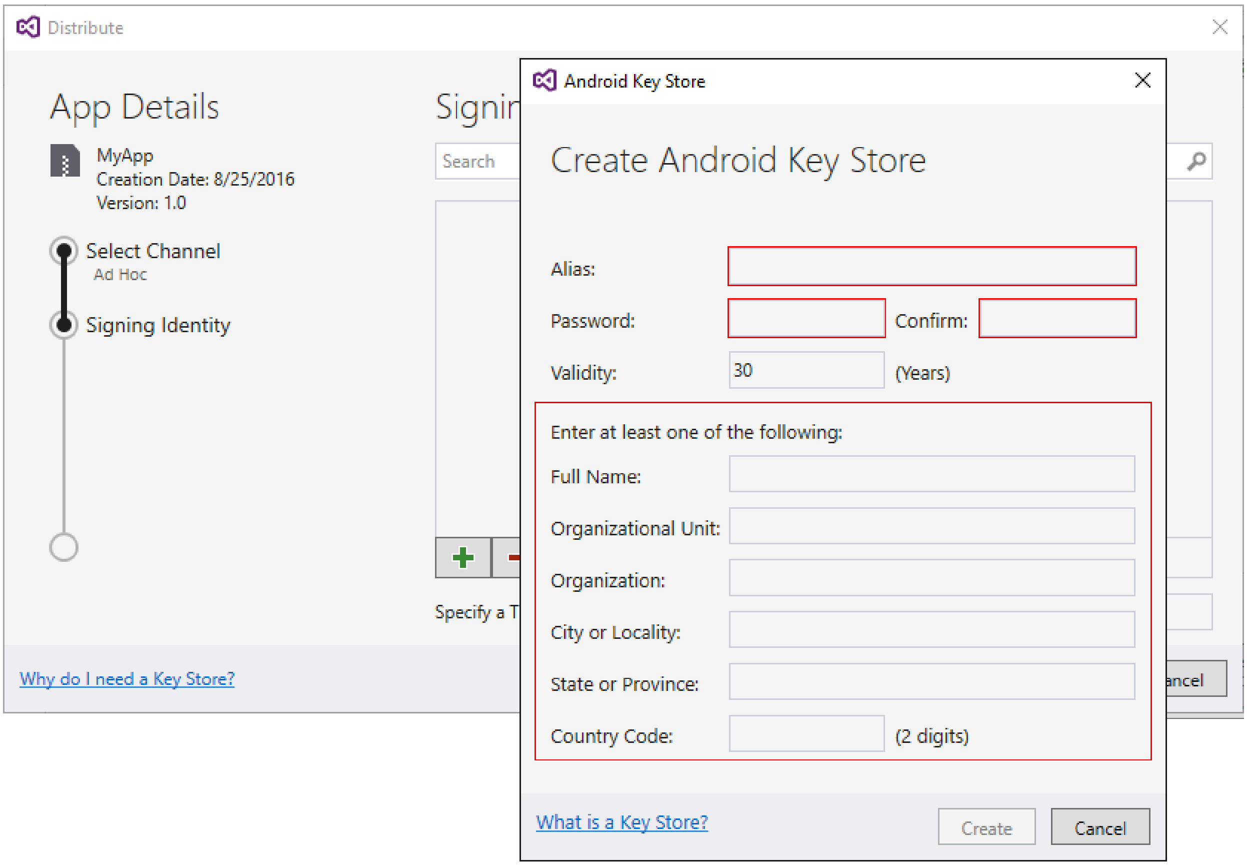
- Visual Studio
- Visual Studio for Mac

After **Ad-Hoc** is selected, Visual Studio opens the **Signing Identity** page of the dialog as shown in the next screenshot. To publish the .APK, it must first be signed with a signing key (also referred to as a certificate).

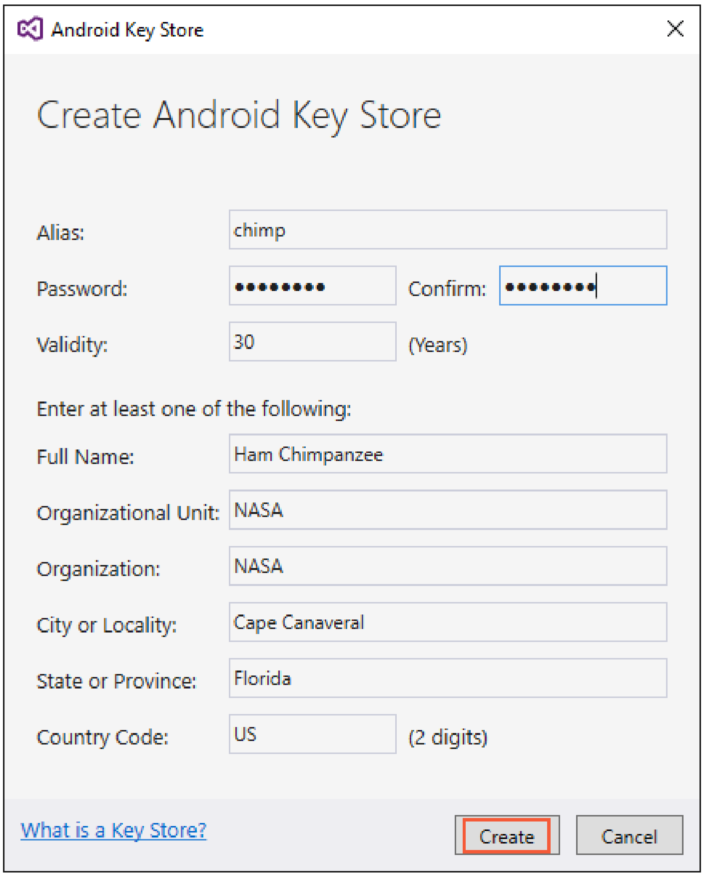
An existing certificate can be used by clicking the **Import** button and then proceeding to Sign the APK. Otherwise, click the click the + button to create a new certificate:



The **Create Android Key Store** dialog is displayed; use this dialog to create a new signing certificate that can be used for signing Android applications. Enter the required information (outlined in red) as shown in this dialog:



The following example illustrates the kind of information that must be provided. Click **Create** to create the new certificate:



The resulting keystore resides in the following location:

C:\Users*USERNAME*\AppData\Local\Xamarin\Mono for Android\Keystore*ALIASALIAS*.keystore**

For example, using **chimp** as the alias, the above steps would create a new signing key in the following location:

C:\Users*USERNAME*\AppData\Local\Xamarin\Mono for Android\Keystore\chimp\chimp.keystore

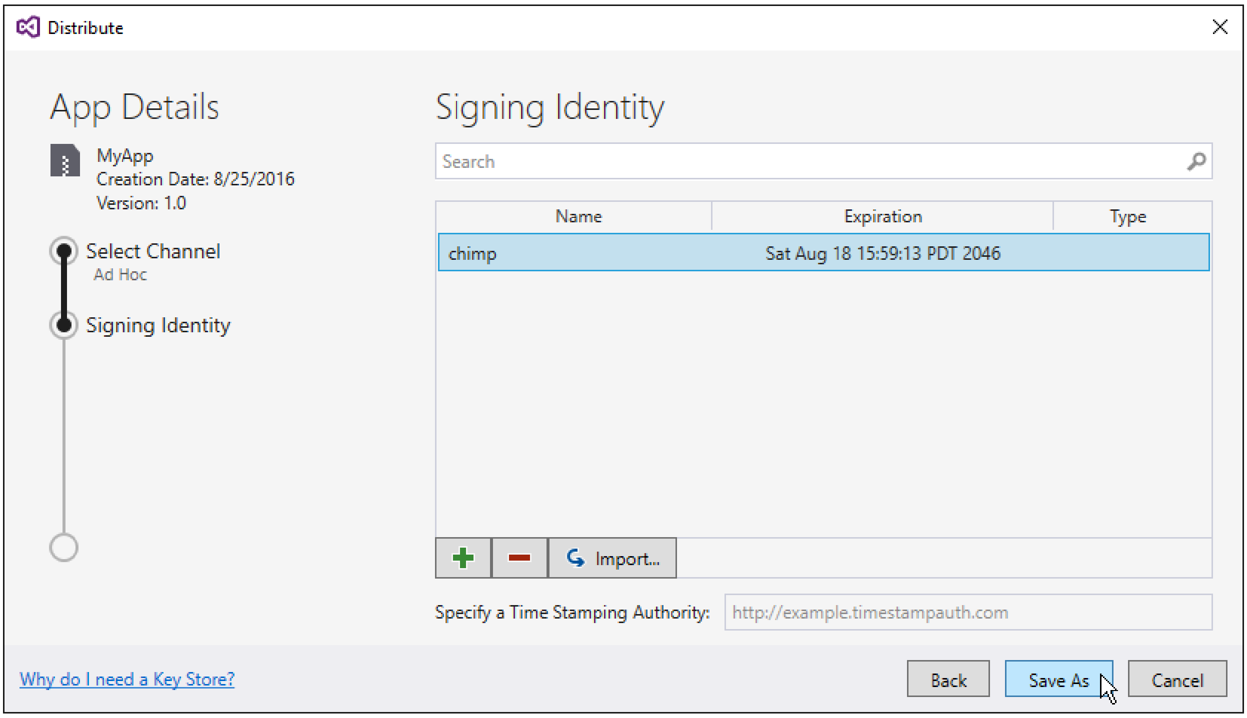
> **IMPORTANT**
>
> The AppData folder is hidden by default and you may need to unhide it to access it.
>
> In addition, be sure to back up the resulting keystore file and password in a safe place – it is not included in the Solution. If you lose your keystore file (for example, because you moved to another computer or reinstalled Windows), you will be unable to sign your app with the same certificate as previous versions.

For more information about the keystore, see Finding your Keystore's MD5 or SHA1 Signature.

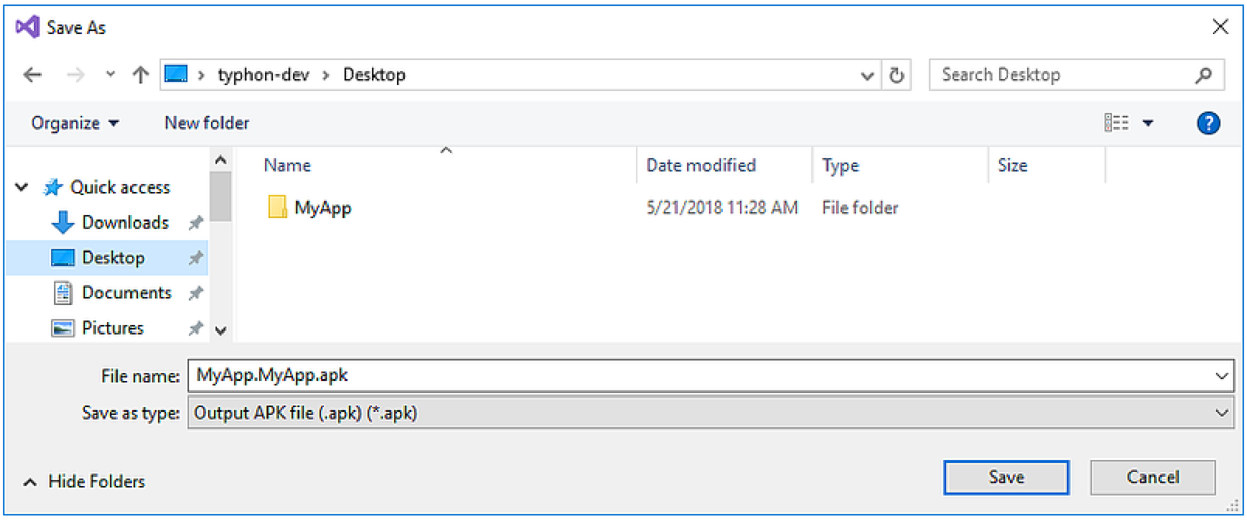
Sign the APK

- Visual Studio
- Visual Studio for Mac

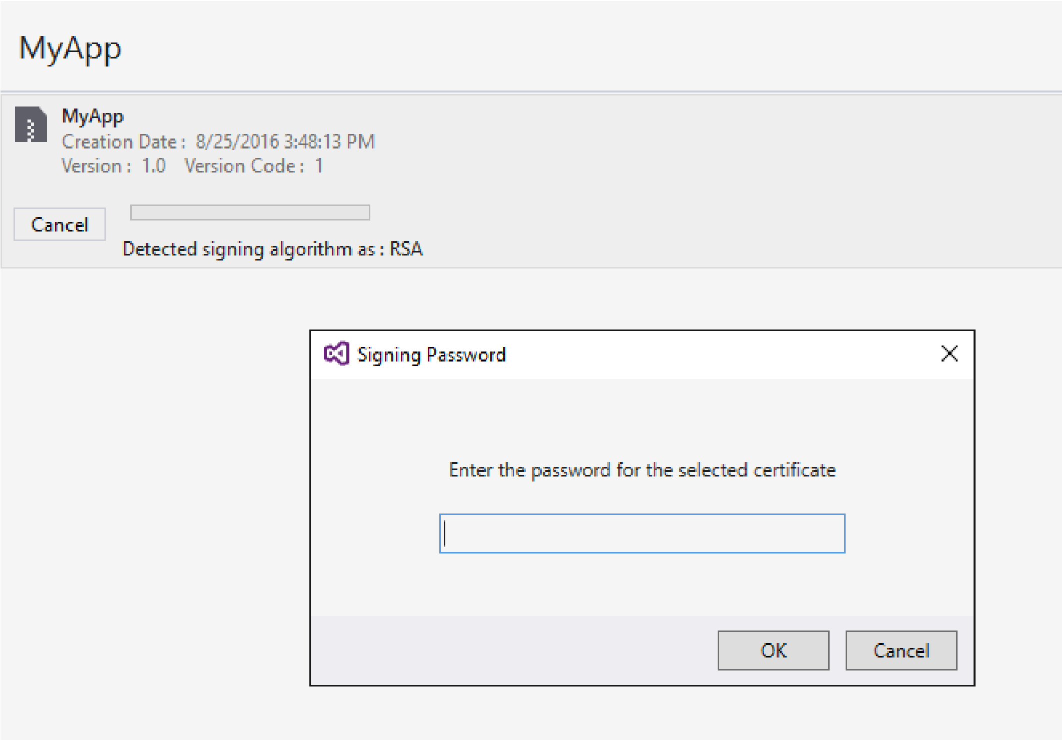
When **Create** is clicked, a new key store (containing a new certificate) will be saved and listed under **Signing Identity** as shown in the next screenshot. To publish an app on Google Play, click **Cancel** and go to Publishing to Google Play. To publish *ad-hoc*, select the signing identity to use for signing and click **Save As** to publish the app for independent distribution. For example, the **chimp** signing identity (created earlier) is selected in this screenshot:



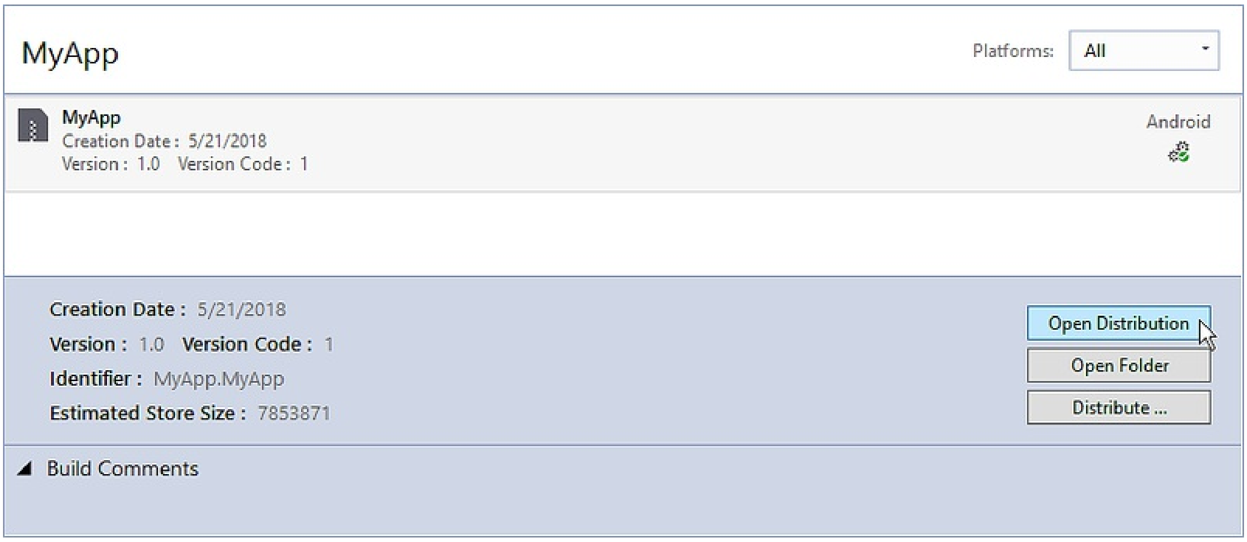
Next, the **Archive Manager** displays the publishing progress. When the publishing process completes, the **Save As** dialog opens to ask for a location where the generated .APK file is to be stored:



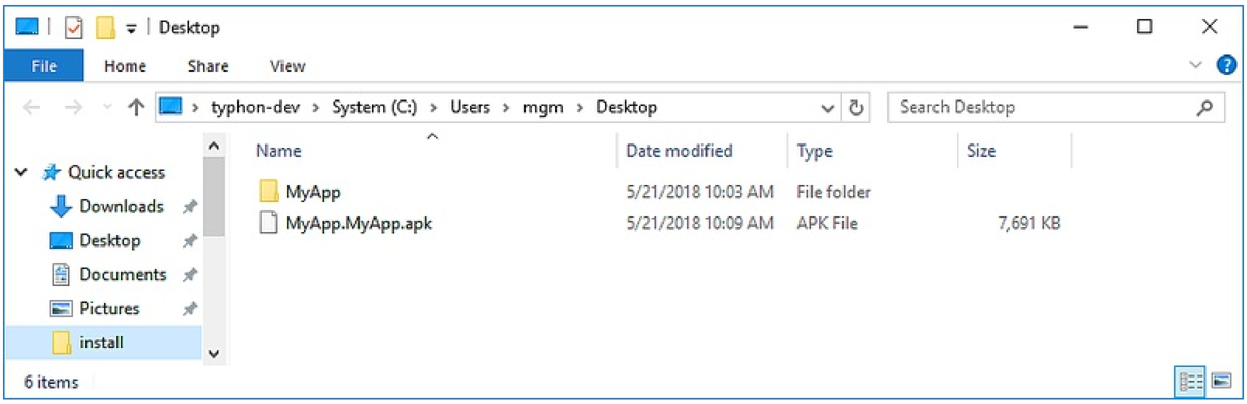
Navigate to the desired location and click **Save**. If the key password is unknown, the **Signing Password** dialog will appear to prompt for the password for the selected certificate:



After the signing process completes, click **Open Distribution**:



This causes Windows Explorer to open the folder containing the generated APK file. At this point, Visual Studio has compiled the Xamarin.Android application into an APK that is ready for distribution. The following screenshot displays an example of the ready-to-publish app, **MyApp.MyApp.apk**:



Next Steps

After the application package has been signed for release, it must be published. The following sections describe several ways to publish an application.

Related Links

- Android Generate Keystore

Manually Signing the APK

11/2/2020 • 6 minutes to read • Edit Online

After the application has been built for release, the APK must be signed prior to distribution so that it can be run on an Android device. This process is typically handled with the IDE, however there are some situations where it is necessary to sign the APK manually, at the command line. The following steps are involved with signing an APK:

1. **Create a Private Key** – This step needs to be performed only once. A private key is necessary to digitally sign the APK. After the private key has been prepared, this step can be skipped for future release builds.

2. **Zipalign the APK** – *Zipalign* is an optimization process that is performed on an application. It enables Android to interact more efficiently with the APK at runtime. Xamarin.Android conducts a check at runtime, and will not allow the application to run if the APK has not been zipaligned.

3. **Sign the APK** – This step involves using the **apksigner** utility from the Android SDK and signing the APK with the private key that was created in the previous step. Applications that are developed with older versions of the Android SDK build tools prior to v24.0.3 will use the **jarsigner** app from the JDK. Both of these tools will be discussed in more detail below.

The order of the steps is important and is dependent on which tool used to sign the APK. When using **apksigner**, it is important to first **zipalign** the application, and then to sign it with **apksigner**. If it is necessary to use **jarsigner** to sign the APK, then it is important to first sign the APK and then run **zipalign**.

Prerequisites

This guide will focus on using **apksigner** from the Android SDK build tools, v24.0.3 or higher. It assumes that an APK has already been built.

Applications that are built using an older version of the Android SDK Build Tools must use **jarsigner** as described in Sign the APK with jarsigner below.

Create a Private Keystore

A *keystore* is a database of security certificates that is created by using the program keytool from the Java SDK. A keystore is critical to publishing a Xamarin.Android application, as Android will not run applications that have not been digitally signed.

During development, Xamarin.Android uses a debug keystore to sign the application, which allows the application to be deployed directly to the emulator or to devices configured to use debuggable applications. However, this keystore is not recognized as a valid keystore for the purposes of distributing applications.

For this reason, a private keystore must be created and used for signing applications. This is a step that should only be performed once, as the same key will be used for publishing updates and can then be used to sign other applications.

It is important to protect this keystore. If it is lost, then it will not be possible to publish updates to the application with Google Play. The only solution to the problem caused by a lost keystore would be to create a new keystore, re-sign the APK with the new key, and then submit a new application. Then the old application would have to be removed from Google Play. Likewise, if this new keystore is compromised or publicly distributed, then it is possible for unofficial or malicious versions of an application to be distributed.

Create a New Keystore

Creating a new keystore requires the command line tool keytool from the Java SDK. The following snippet is an example of how to use **keytool** (replace `<my-filename>` with the file name for the keystore and `<key-name>` with the name of the key within the keystore):

```
$ keytool -genkeypair -v -keystore <filename>.keystore -alias <key-name> -keyalg RSA \
          -keysize 2048 -validity 10000
```

The first thing that **keytool** will ask for is the password for the keystore. Then it will ask for some information to help with creating the key. The following snippet is an example of creating a new key called `publishingdoc` that will be stored in the file `xample.keystore`:

```
$ keytool -genkeypair -v -keystore xample.keystore -alias publishingdoc -keyalg RSA -keysize 2048 -validity
10000
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  Ham Chimpanze
What is the name of your organizational unit?
  [Unknown]:  NASA
What is the name of your organization?
  [Unknown]:  NASA
What is the name of your City or Locality?
  [Unknown]:  Cape Canaveral
What is the name of your State or Province?
  [Unknown]:  Florida
What is the two-letter country code for this unit?
  [Unknown]:  US
Is CN=Ham Chimpanze, OU=NASA, O=NASA, L=Cape Canaveral, ST=Florida, C=US correct?
  [no]:  yes

Generating 2,048 bit RSA key pair and self-signed certificate (SHA1withRSA) with a validity of 10,000 days
        for: CN=Ham Chimpanze, OU=NASA, O=NASA, L=Cape Canaveral, ST=Florida, C=US
Enter key password for <publishingdoc>
        (RETURN if same as keystore password):
Re-enter new password:
[Storing xample.keystore]
```

To list the keys that are stored in a keystore, use the **keytool** with the – `list` option:

```
$ keytool -list -keystore xample.keystore
```

Zipalign the APK

Before signing an APK with **apksigner**, it is important to first optimize the file using the **zipalign** tool from the Android SDK. **zipalign** will restructure the resources in an APK along 4-byte boundaries. This alignment allows Android to quickly load the resources from the APK, increasing the performance of the application and potentially reducing memory use. Xamarin.Android will conduct a run-time check to determine if the APK has been zipaligned. If the APK is not zipaligned, then the application will not run.

The follow command will use the signed APK and produce a signed, zipaligned APK called **helloworld.apk** that is ready for distribution.

```
$ zipalign -f -v 4 mono.samples.helloworld-unsigned.apk helloworld.apk
```

Sign the APK

After zipaligning the APK, it is necessary to sign it using a keystore. This is done with the **apksigner** tool, found in the **build-tools** directory of the version of the SDK build tools. For example, if the Android SDK build tools v25.0.3 is installed, then **apksigner** can be found in the directory:

```
$ ls $ANDROID_HOME/build-tools/25.0.3/apksigner
/Users/tom/android-sdk-macosx/build-tools/25.0.3/apksigner*
```

The following snippet assumes that **apksigner** is accessible by the `PATH` environment variable. It will sign an APK using the key alias `publishingdoc` that is contained in the file **xample.keystore**:

```
$ apksigner sign --ks xample.keystore --ks-key-alias publishingdoc mono.samples.helloworld.apk
```

When this command is run, **apksigner** will ask for the password to the keystore if necessary.

See Google's documentation for more details on the use of **apksigner**.

> **NOTE**
>
> According to Google issue 62696222, **apksigner** is "missing" from the Android SDK. The workaround for this is to install the Android SDK build tools v25.0.3 and use that version of **apksigner**.

Sign the APK with jarsigner

> **WARNING**
>
> This section only applies if it is nececssary to sign the APK with the **jarsigner** utility. Developers are encouraged to use **apksigner** to sign the APK.

This technique involves signing the APK file using the **jarsigner** command from the Java SDK. The **jarsigner** tool is provided by the Java SDK.

The following shows how to sign an APK by using **jarsigner** and the key `publishingdoc` that is contained in a keystore file named **xample.keystore** :

```
$ jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore xample.keystore
mono.samples.helloworld.apk publishingdoc
```

> **NOTE**
>
> When using **jarsigner**, it is important to sign the APK *first*, and then to use **zipalign**.

Related Links

- Application Signing
- jarsigner
- keytool
- zipalign
- Build Tools 26.0.0 - where did apksigner go?

Finding your Keystore's Signature

4/12/2021 • 3 minutes to read • Edit Online

The MD5 or SHA1 signature of a Xamarin.Android app depends on the **.keystore** file that was used to sign the APK. Typically, a debug build will use a different **.keystore** file than a release build.

For Debug / Non-Custom Signed Builds

Xamarin.Android signs all debug builds with the same **debug.keystore** file. This file is generated when Xamarin.Android is first installed.The steps below detail the process for finding the MD5 or SHA1 signature of the default Xamarin.Android **debug.keystore** file.

- Visual Studio
- Visual Studio for Mac

Locate the Xamarin **debug.keystore** file that is used to sign the app. By default, the keystore that is used to sign debug versions of a Xamarin.Android application can be found at the following location:

C:\Users*USERNAME*\AppData\Local\Xamarin\Mono for Android\debug.keystore

Information about a keystore is obtained by running the `keytool.exe` command from the JDK. This tool is typically found in the following location:

C:\Program Files (x86)\Java\jdk*VERSION*\bin\keytool.exe

Add the directory containing **keytool.exe** to the `PATH` environment variable. Open a **Command Prompt** and run `keytool.exe` using the following command:

```
keytool.exe -list -v -keystore "%LocalAppData%\Xamarin\Mono for Android\debug.keystore" -alias
androiddebugkey -storepass android -keypass android
```

When run, **keytool.exe** should output the following text. The **MD5:** and **SHA1:** labels identify the respective signatures:

```
Alias name: androiddebugkey
Creation date: Aug 19, 2014
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=Android Debug, O=Android, C=US
Issuer: CN=Android Debug, O=Android, C=US
Serial number: 53f3b126
Valid from: Tue Aug 19 13:18:46 PDT 2014 until: Sun Nov 15 12:18:46 PST 2043
Certificate fingerprints:
         MD5:  27:78:7C:31:64:C2:79:C6:ED:E5:80:51:33:9C:03:57
         SHA1: 00:E5:8B:DA:29:49:9D:FC:1D:DA:E7:EE:EE:1A:8A:C7:85:E7:31:23
         SHA256: 21:0D:73:90:1D:D6:3D:AB:4C:80:4E:C4:A9:CB:97:FF:34:DD:B4:42:FC:
08:13:E0:49:51:65:A6:7C:7C:90:45
         Signature algorithm name: SHA1withRSA
         Version: 3
```

For Release / Custom Signed Builds

The process for release builds that are signed with a custom **.keystore** file are the same as above, with the release **.keystore** file replacing the **debug.keystore** file that is used by Xamarin.Android. Replace your own values for the keystore password, and alias name from when the release keystore file was created.

- Visual Studio
- Visual Studio for Mac

When the Visual Studio **Distribute** wizard is used to sign a Xamarin.Android app, the resulting keystore resides in the following location:

C:\Users*USERNAME*\AppData\Local\Xamarin\Mono for Android\Keystore*aliasalias*.keystore**

For example, if you followed the steps in Create a New Certificate to create a new signing key, the resulting example keystore resides in the following location:

C:\Users*USERNAME*\AppData\Local\Xamarin\Mono for Android\Keystore\chimp\chimp.keystore

For more information about signing a Xamarin.Android app, see Signing the Android Application Package.

Publishing an Application

7/8/2021 • 3 minutes to read • Edit Online

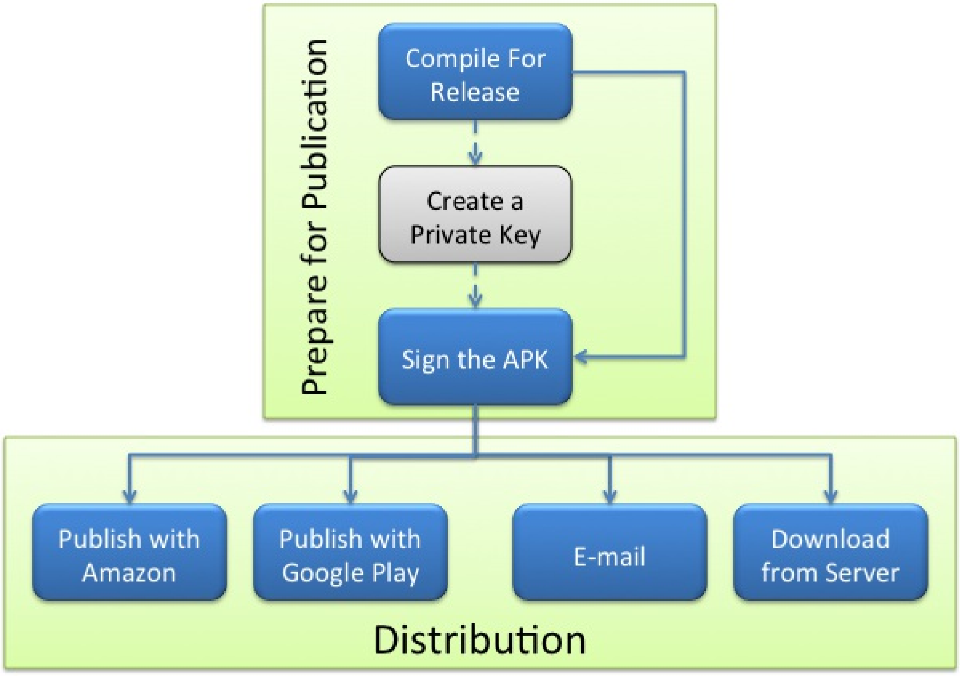
After a great application has been created, people will want to use it. This section covers the steps involved with the public distribution of an application created with Xamarin.Android via channels such as e-mail, a private web server, Google Play, or the Amazon App Store for Android.

Overview

The final step in the development of a Xamarin.Android application is to publish the application. Publishing is the process of compiling a Xamarin.Android application so that it is ready for users to install on their devices, and it involves two essential tasks:

- **Preparing for Publication** – A release version of the application is created that can be deployed to Android-powered devices (see Preparing an Application for Release for more information about release preparation).
- **Distribution** – The release version of an application is made available through one or more of the various distribution channels.

The following diagram illustrates the steps involved with publishing a Xamarin.Android application:



As can be seen by the diagram above, the preparation is the same regardless of the distribution method that is used. There are several ways that an Android application may be released to users:

- **Via a Website** – A Xamarin.Android application can be made available for download on a website, from which users may then install the application by clicking on a link.
- **By e-mail** – It is possible for users to install a Xamarin.Android application from their e-mail. The application will be installed when the attachment is opened with an Android-powered device.
- **Through a Market** – There are several application marketplaces that exist for distribution, such as Google Play or Amazon App Store for Android .

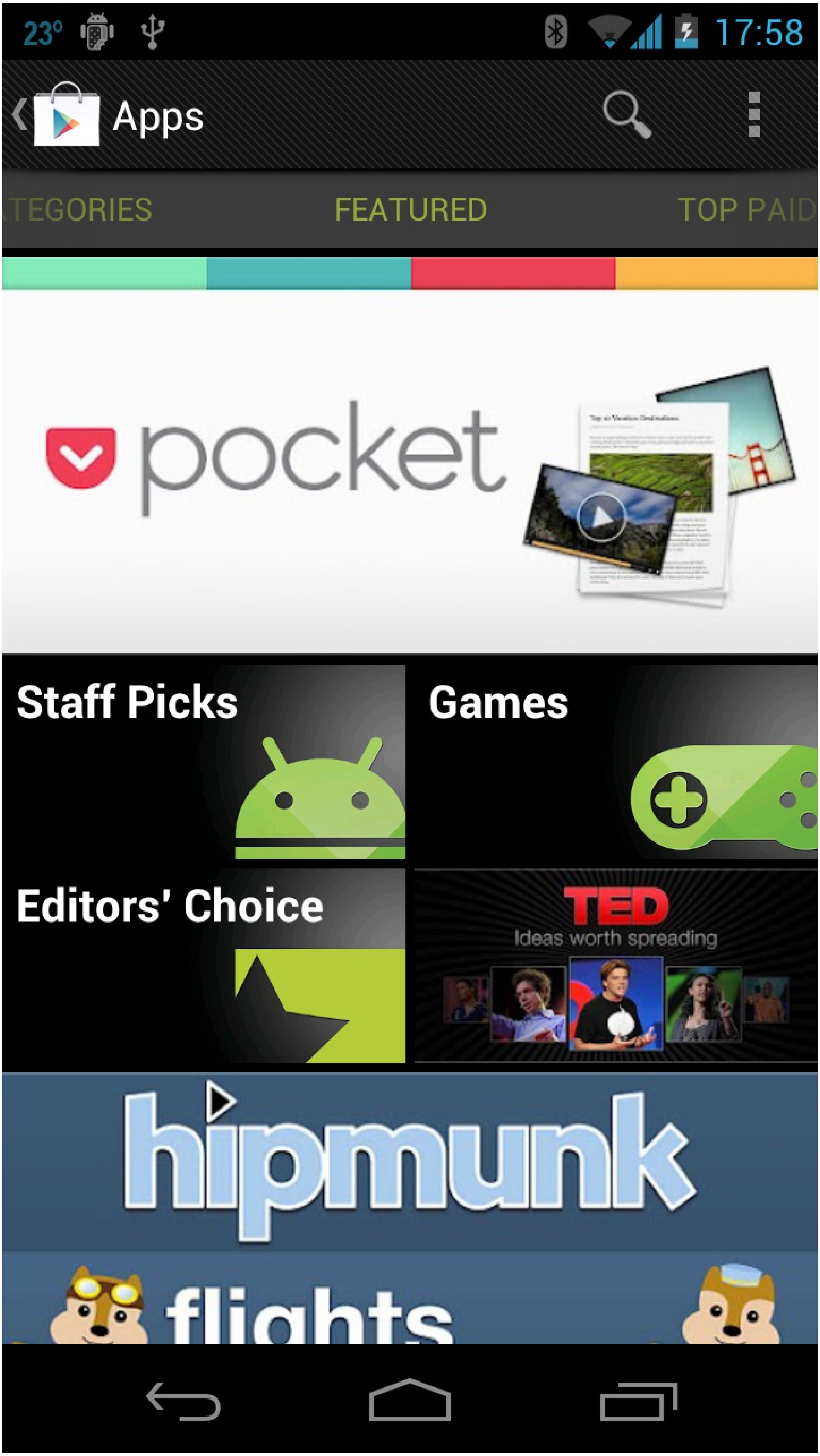
Using an established marketplace is the most common way to publish an application as it provides the broadest market reach and the greatest control over distribution. However, publishing an application through a marketplace requires additional effort.

Multiple channels can distribute a Xamarin.Android application simultaneously. For example, an application could be published on Google Play, the Amazon App Store for Android, and also be downloaded from a web server.

The other two methods of distribution (downloading or e-mail) are most useful for a controlled subset of users, such as an enterprise environment or an application that is only meant for a small or well-specified set of users. Server and e-mail distribution are also simpler publishing models, requiring less preparation to publish an application.

The Amazon Mobile App Distribution Program enables mobile app developers to distribute and sell their applications on Amazon. Users can discover and shop for apps on their Android-powered devices by using the Amazon App Store application. A screenshot of the Amazon App Store running on an Android device appears below:

Google Play is arguably the most comprehensive and popular marketplace for Android applications. Google Play allows users to discover, download, rate, and pay for applications by clicking a single icon either on their device or on their computer. Google Play also provides tools to assist in the analysis of sales and market trends and to control which devices and users may download an application. A screenshot of Google Play running on an Android device appears below:



23°
17:58
Apps
TEGORIES
FEATURED
TOP PAID
pocket
Staff Picks
Games
Editors' Choice
TED
Ideas worth spreading
hipmunk
flights

This section shows how to upload the application to a store such as Google Play, along with the appropriate promotional materials. APK expansion files are explained, providing a conceptual overview of what they are and how they work. Google Licensing services are also described. Finally, alternate means of distribution are introduced, including the use of an HTTP web server, simple e-mail distribution, and the Amazon App Store for Android.

Related Links

- HelloWorldPublishing (sample)
- Build Process
- Linking
- Obtaining A Google Maps API Key
- Deploy via Visual Studio App Center
- Application Signing
- Publishing on Google Play
- Google Application Licensing
- Android.Play.ExpansionLibrary
- Mobile App Distribution Portal
- Amazon Mobile App Distribution FAQ

Publishing to Google Play

7/8/2021 • 12 minutes to read • Edit Online

Although there are many app markets for distributing an application, Google Play is arguably the largest and most visited store in the world for Android apps. Google Play provides a single platform for distributing, advertising, selling, and analyzing the sales of an Android application.

This section will cover topics that are specific to Google Play, such as registering to become a publisher, gathering assets to help Google Play promote and advertise your application, guidelines for rating your application on Google Play, and using filters to restrict the deployment of an application to certain devices.

Requirements

To distribute an application through Google Play, a developer account must be created. This only needs to be performed once, and does involve a one time fee of $25 USD.

All applications need to be signed with a cryptographic key that expires after October 22, 2033.

The maximum size for an APK published on Google Play is 100MB. If an application exceeds that size, Google Play will allow extra assets to be delivered through *APK Expansion Files*. Android Expansion files permit the APK to have 2 additional files, each of them up to 2GB in size. Google Play will host and distribute these files at no cost. Expansion files will be discussed in another section.

Google Play is not globally available. Some locations may not be supported for the distribution of applications.

Becoming a Publisher

To publish applications on Google play, it is necessary to have a publisher account. To sign up for a publisher account follow these steps:

1. Visit the Google Play Developer Console.
2. Enter basic information about your developer identity.
3. Read and accept the Developer Distribution Agreement for your locale.
4. Pay the $25 USD registration fee.
5. Confirm verification by e-mail.
6. After the account has been created, it is possible to publish applications using Google Play.

Google Play does not support all countries in the world. The most up-to-date lists of countries can be found in the following links:

1. Supported Locations for Developer & Merchant Registration – This is a list of all countries where developers may register as merchants and sell paid applications.
2. Supported Locations for distribution to Google Play users – This is a list of all countries where applications may be distributed.

Preparing Promotional Assets

To effectively promote and advertise an application on Google Play, Google allows developers to submit promotional assets such as screenshots, graphics, and video to be submitted. Google Play will then use those assets to advertise and promote the application.

Launcher Icons

A *launcher icon* is a graphic that represents an application. Each launcher icon should be a 32-bit PNG with an alpha channel for transparency. An application should have icons for all of the generalized screen densities as outlined in the list below:

- **ldpi** (120dpi) – 36 x 36 px
- **mdpi** (160dpi) – 48 x 48 px
- **hdpi** (240dpi) – 72 x 72 px
- **xhdpi** (320dpi) – 96 x 96 px

Launcher icons are the first things that a user will see of applications on Google Play, so care should be taken to make the launcher icons visually appealing and meaningful.

Tips for Launcher Icons:

1. **Simple and uncluttered**– Launcher icons should be kept simple and uncluttered. This means excluding the name of the application from the icon. Simpler icons will be more memorable, and will be easier to distinguish at the smaller sizes.
2. **Icons should not be thin**– Overly thin icons will not stand out well on all backgrounds.
3. **Use the alpha channel**– Icons should make use of the alpha channel, and should not be full-framed images.

High Resolution Application Icons

Applications on Google Play require a high fidelity version of the application icon. It is only used by Google Play, and does not replace the application launcher icon. The specifications for the high-resolution icon are:

1. 32-bit PNG with an alpha channel
2. 512 x 512 pixels
3. Maximum size of 1024KB

The Android Asset Studio is a helpful tool for creating suitable launcher icons and the high-resolution application icon.

Screenshots

Google play requires a minimum of two and a maximum of eight screenshots for an application. They will be displayed on an application's details page in Google Play.

The specs for screenshots are:

1. 24 bit PNG or JPG with no alpha channel
2. 320w x 480h or 480w x 800h or 480w x 854h. Landscaped images will be cropped.

Promotional Graphic

This is an optional image used by Google Play:

1. It is a 180w x 120h 24 bit PNG or JPG with no alpha channel.
2. No border in art.

Feature Graphic

Used by the featured section of Google Play. This graphic may be displayed alone without an application icon.

1. 1024w x 500h PNG or JPG with no alpha channel and no transparency.
2. All of the important content should be within a frame of 924x500. Pixels outside of this frame may be cropped for stylistic purposes.
3. This graphic may be scaled down: use large text and keep graphics simple.

Video Link

This is a URL to a YouTube video showcasing the application. The video should be 30 seconds to 2 minutes in length and showcase the best parts of your application.

Publishing to Google Play

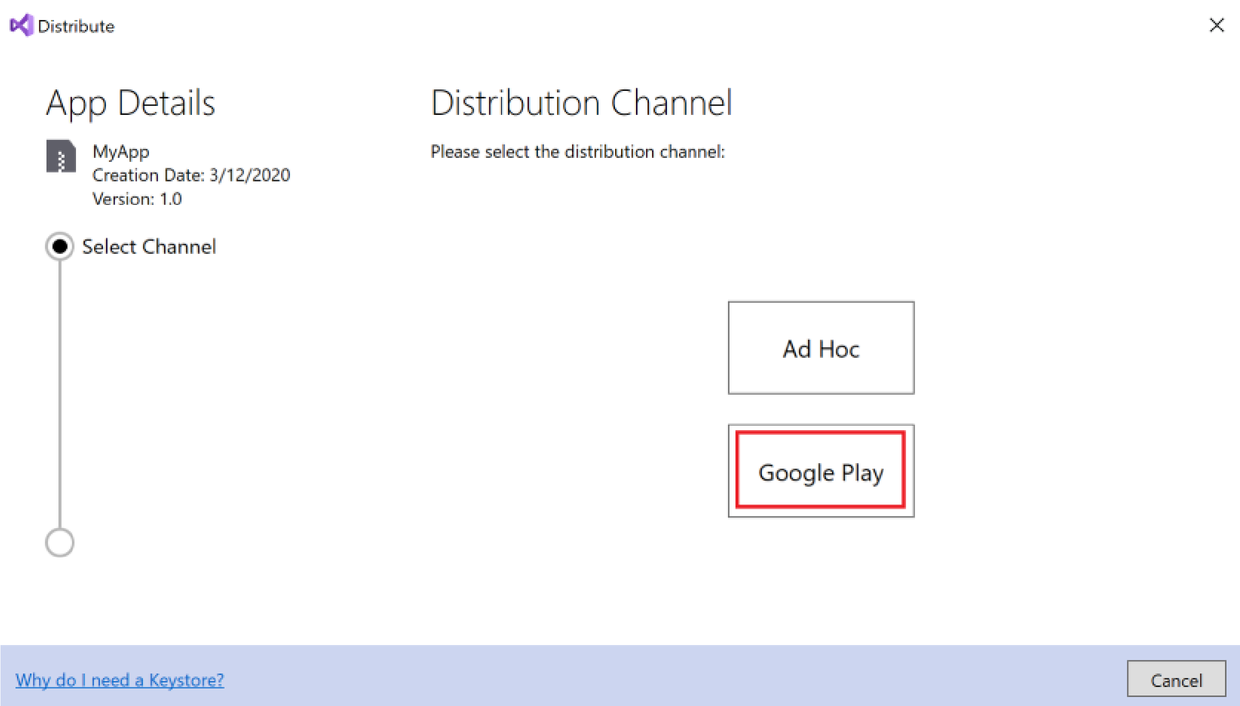
- Visual Studio
- Visual Studio for Mac

Xamarin Android 7.0 introduces an integrated workflow for publishing apps to Google Play from Visual Studio. If you are using a version of Xamarin Android earlier than 7.0, you must manually upload your APK via the Google Play Developer Console. Also, you must have at least one APK already uploaded before you can use the integrated workflow. If you have not yet uploaded your first APK, you must upload it manually. For more information, see Manually Uploading the APK.

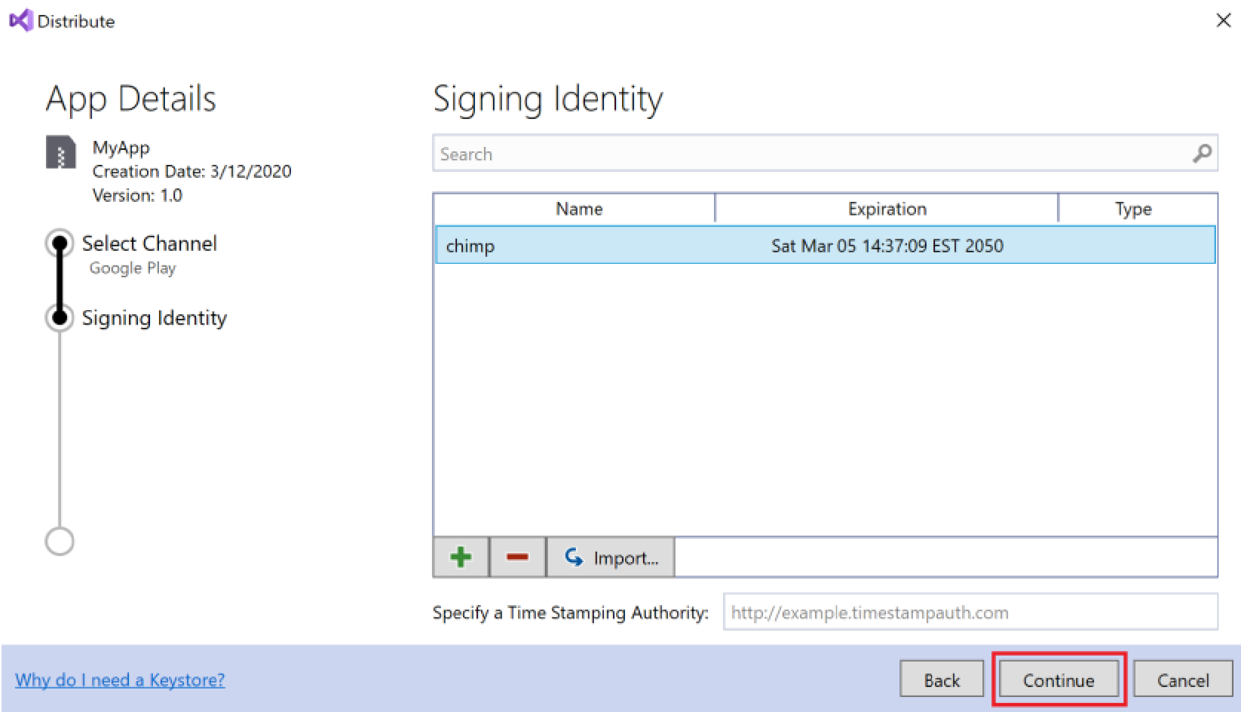
Creating a New Certificate, explained how to create a new certificate for signing Android apps. The next step is to publish a signed app to Google Play:

1. Sign into your Google Play Developer account to create a new project that is linked to your Google Play Developer account.
2. Create an **OAuth Client** that authenticates your app.
3. Enter the resulting Client ID and Client secret into Visual Studio.
4. Register your account with Visual Studio.
5. Sign the app with your certificate.
6. Publish your signed app to Google Play.

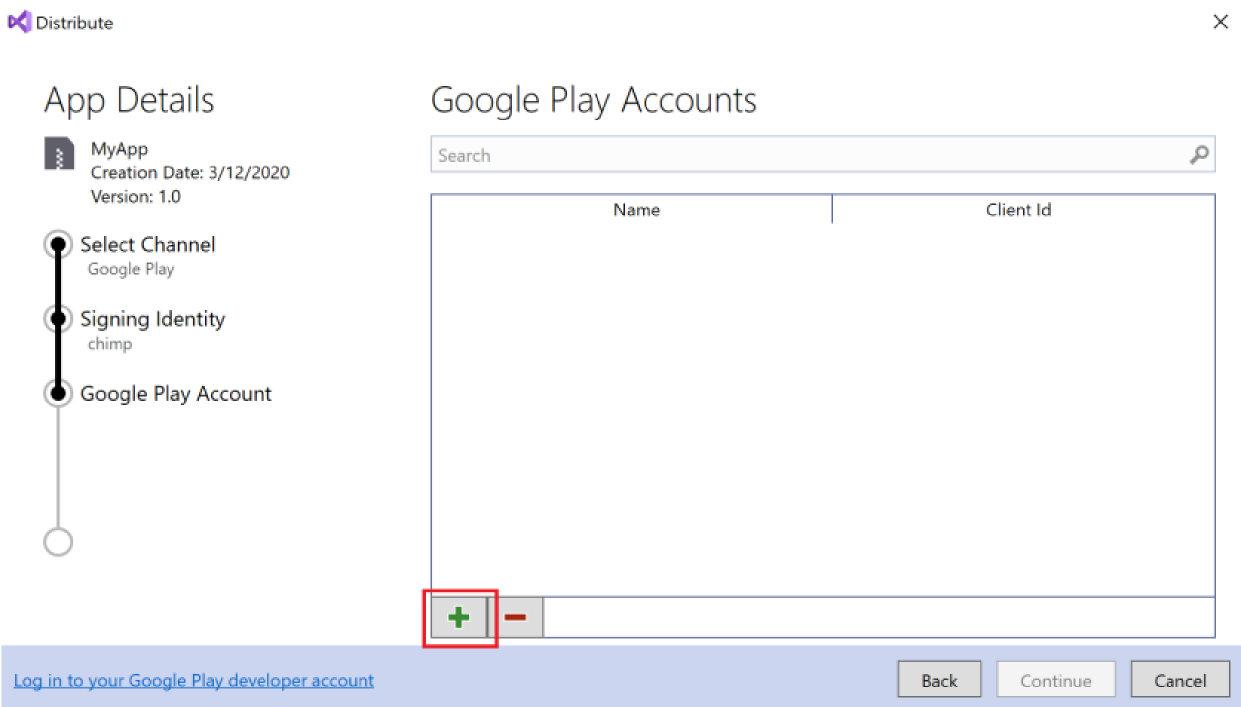
In Archive for Publishing, the **Distribution Channel** dialog presented two choices for distribution: **Ad Hoc** and **Google Play**. If the **Signing Identity** dialog is displayed instead, click **Back** to return to the **Distribution Channel** dialog. Select **Google Play**:



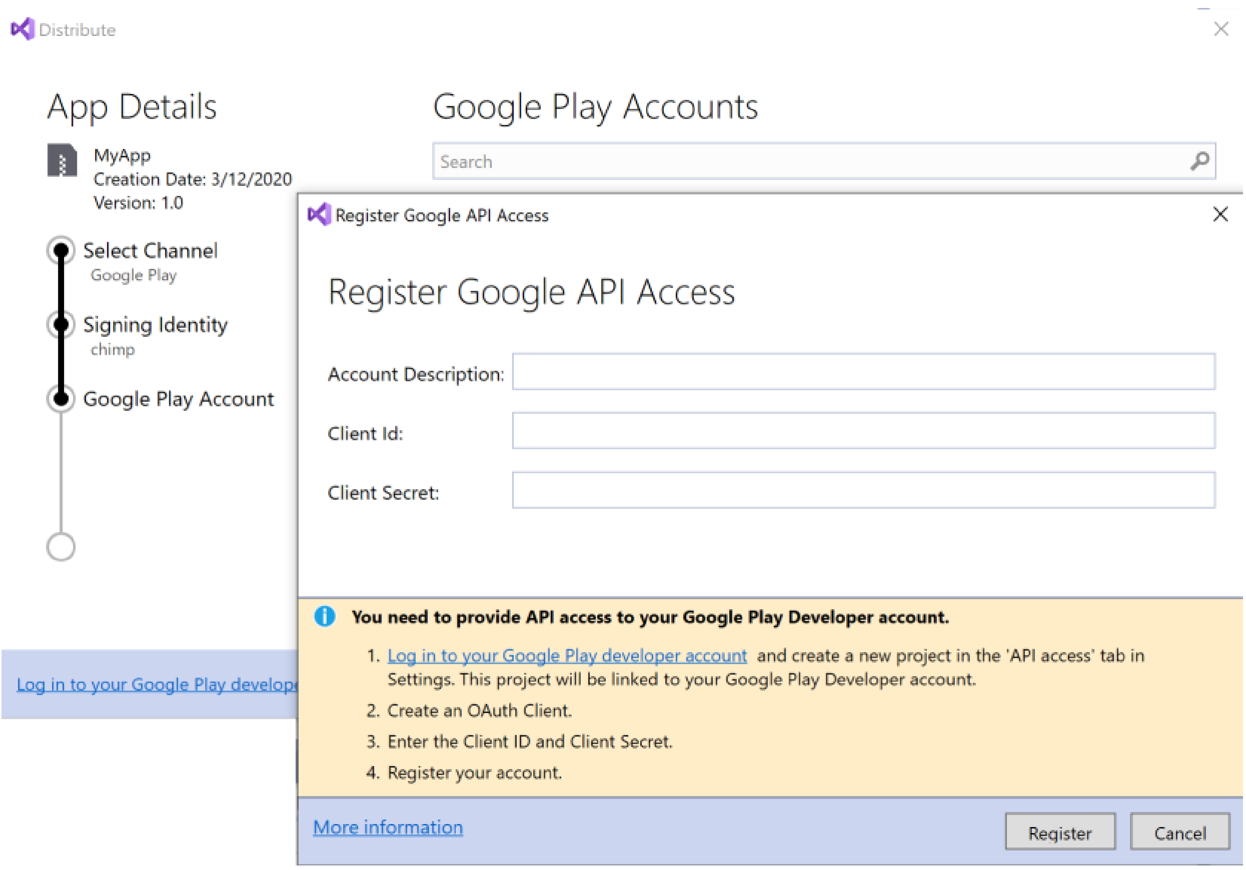
In the **Signing Identity** dialog, select the identity created in Creating a New Certificate and click **Continue**:



In the **Google Play Accounts** dialog, click the **+** button to add a new Google Play Account:



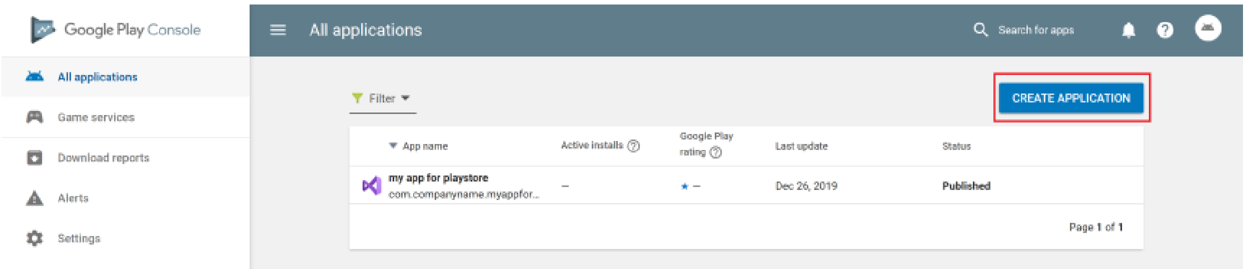
In the **Register Google API Access** dialog, you must provide the *Client ID* and *Client secret* that provides API access to your Google Play Developer account:



The next section explains how to create a new Google API project and generate the needed *Client ID* and *Client secret*.

Create a Google API Project

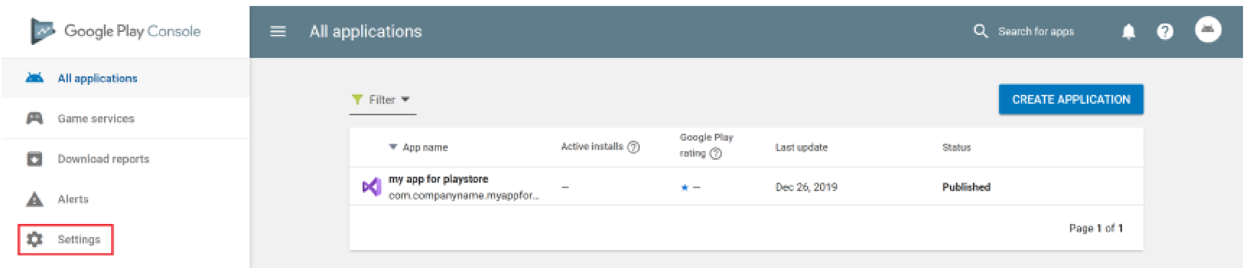
First, sign into your Google Play Developer account. If you do not already have a Google Play Developer account, see Get Started with Publishing. Also, the Google Play Developer API Getting Started explains how to use the Google Play Developer API. After you sign into the Google Play Developer Console, click **CREATE APPLICATION**:



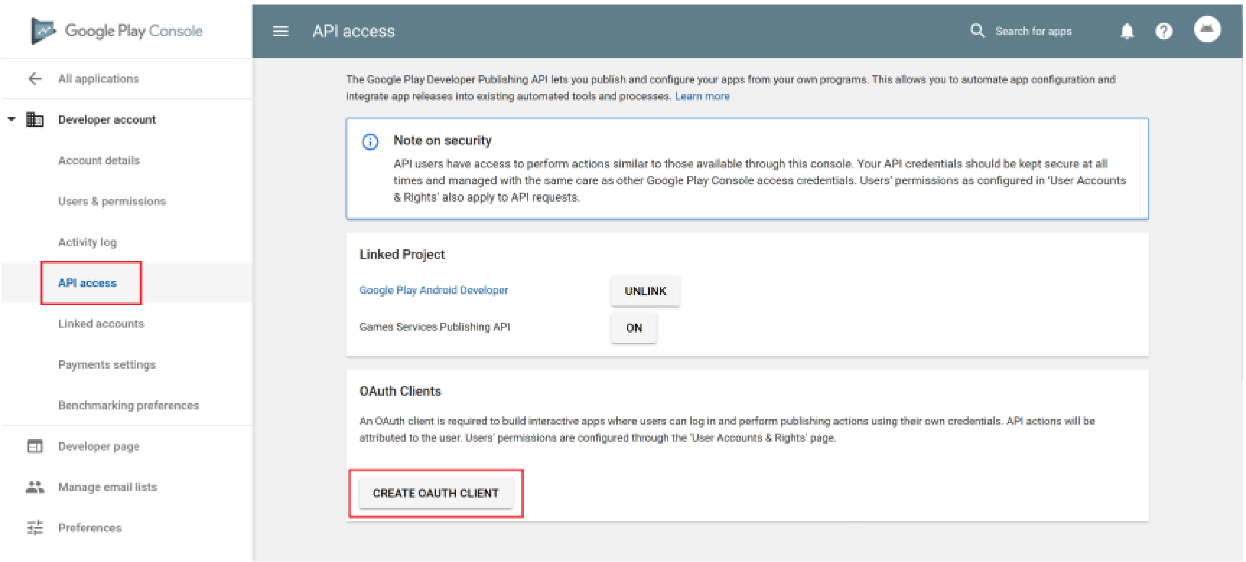
After creating the new project, it will be linked to your Google Play Developer Console account.

The next step is to create an OAuth Client for the app (if one has not already been created). When users request access to their private data using your app, your OAuth Client ID is used to authenticate your app.

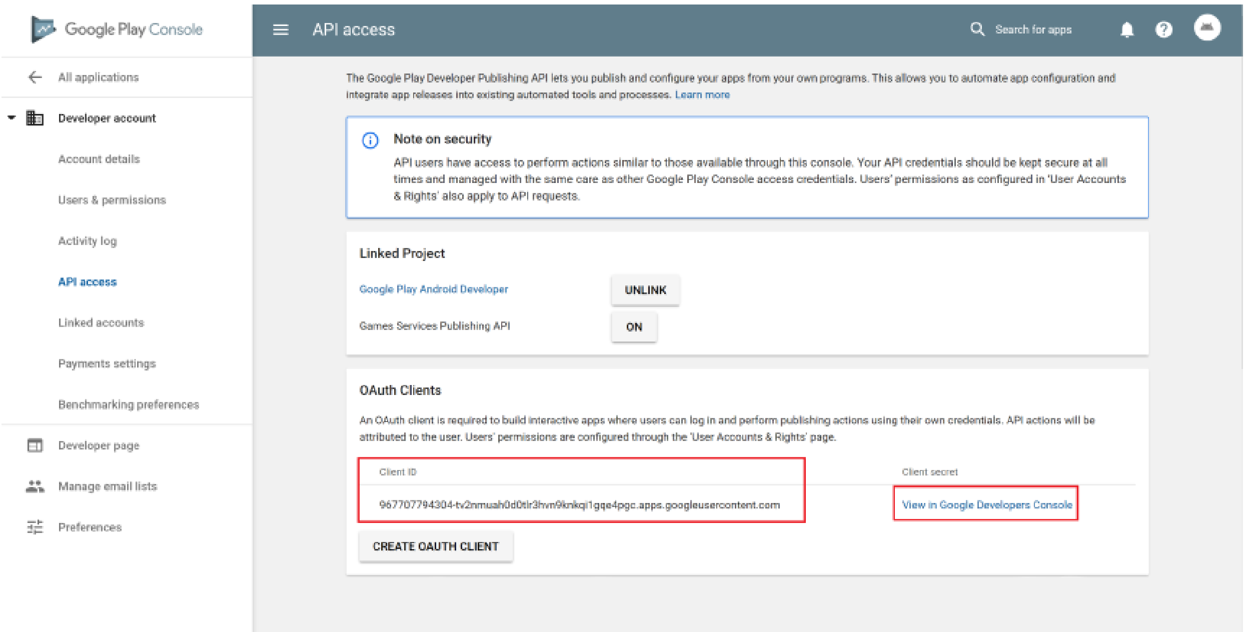
Go to the **Settings** page.



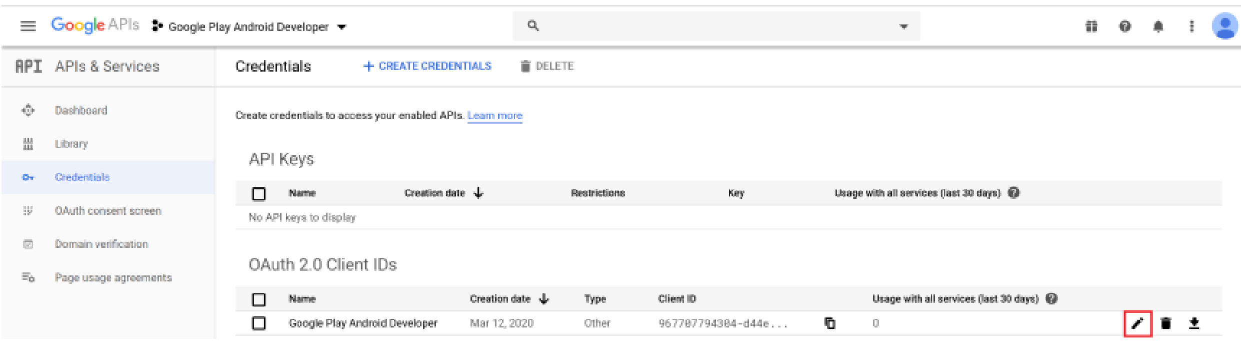
In the **Settings** page, select **API access** and click **CREATE OAUTH CLIENT** to create a new OAuth client:



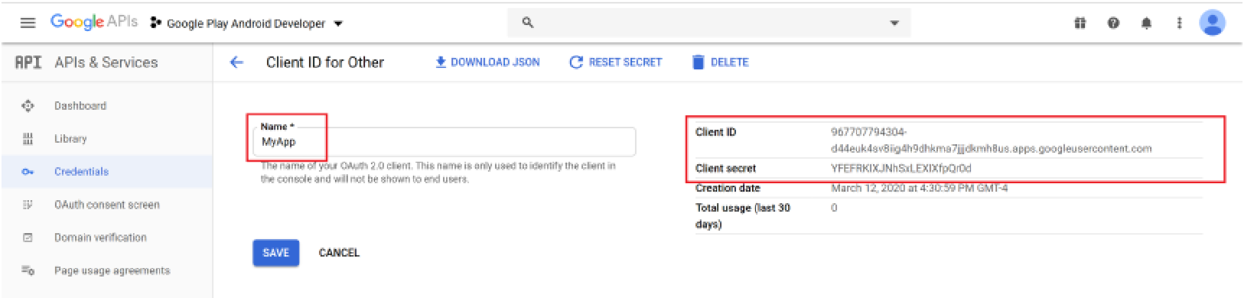
After a few seconds, a new Client ID is generated. Click **View in Google Developers Console** to see your new Client ID in the Google Developer's Console:



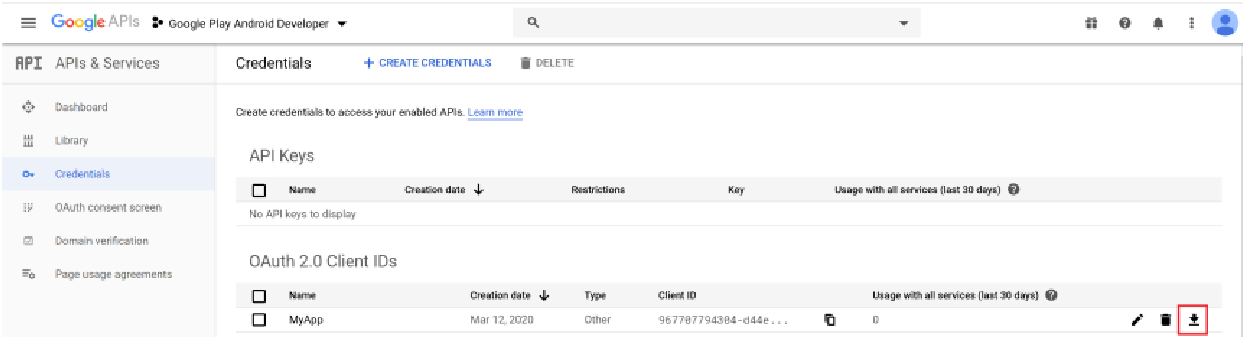
The Client ID is displayed along its name and creation date. Click the **Edit OAuth Client** icon to view the Client secret for your app:



The default name of the OAuth client is *Google Play Android Developer*. This can be changed to the name of Xamarin.Android app, or any suitable name. In this example, the OAuth Client name is changed to the name of the app, **MyApp**:



Click **Save** to save changes. This returns to the **Credentials** page where to download the credentials by clicking on the **Download JSON** icon:

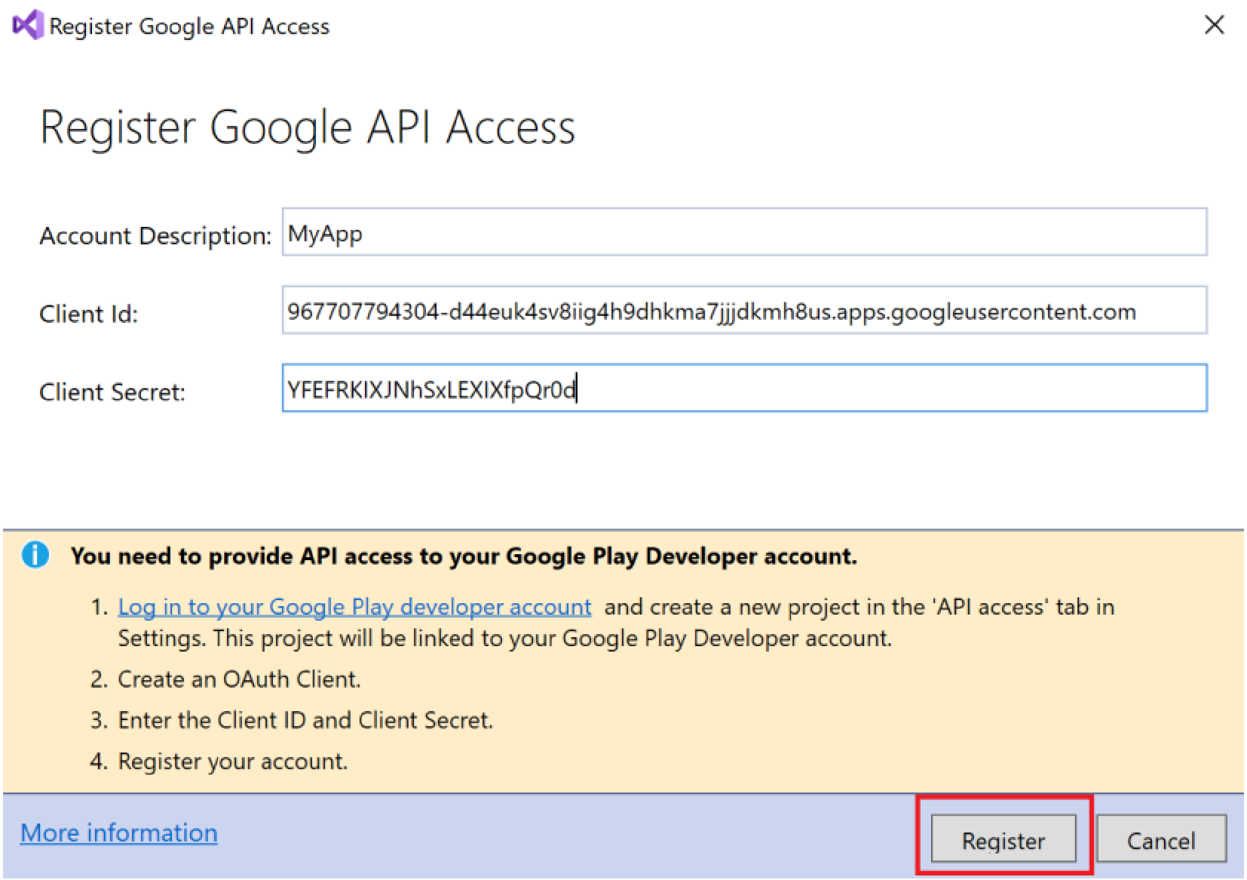


This JSON file contains the Client ID and Client secret that you can cut and paste into the **Sign and Distribute** dialog in the next step.

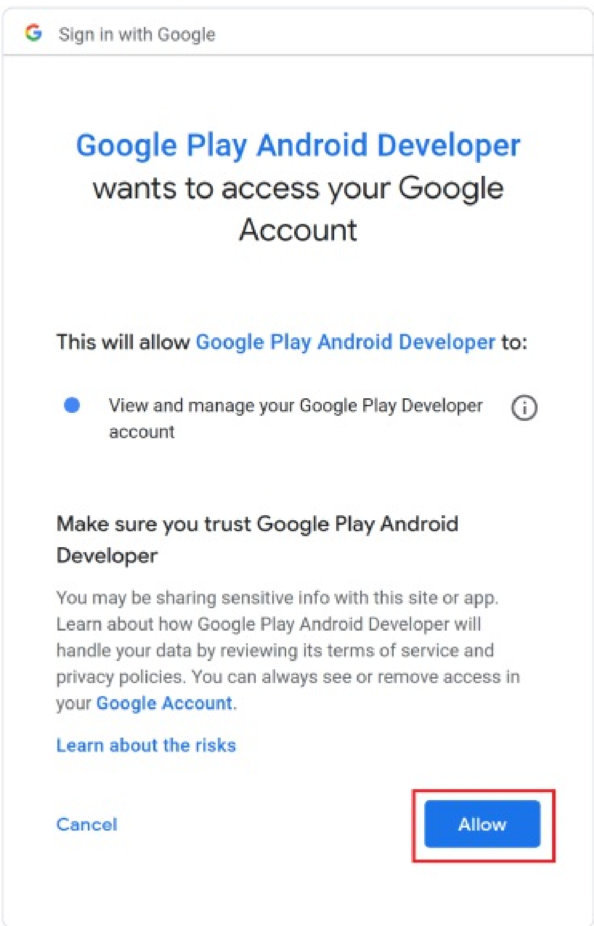
Register Google API Access

- Visual Studio
- Visual Studio for Mac

Use the Client ID and Client secret to complete the **Google Play API Account** dialog in Visual Studio for Mac. It is possible to give the account a description – this makes it possible to register more than one Google Play account and upload future APK's to different Google Play accounts. Copy the Client ID and Client secret to this dialog and click **Register**:



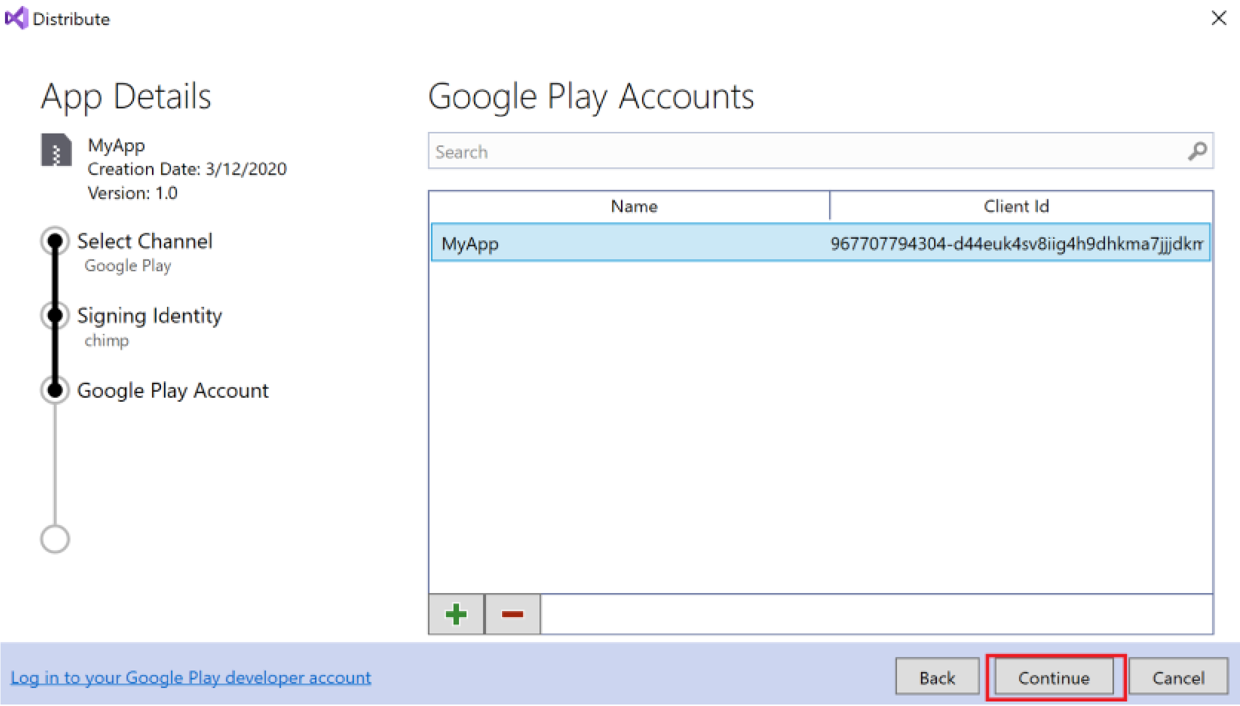
A web browser will open and prompt you to sign into your Google Play Android Developer account (if you are not already signed in). After you sign in, the following prompt is displayed in the web browser. Click **Allow** to authorize the app:



Publish

After clicking **Allow**, the browser reports *Received verification code. Closing...* and the app is added to the list of

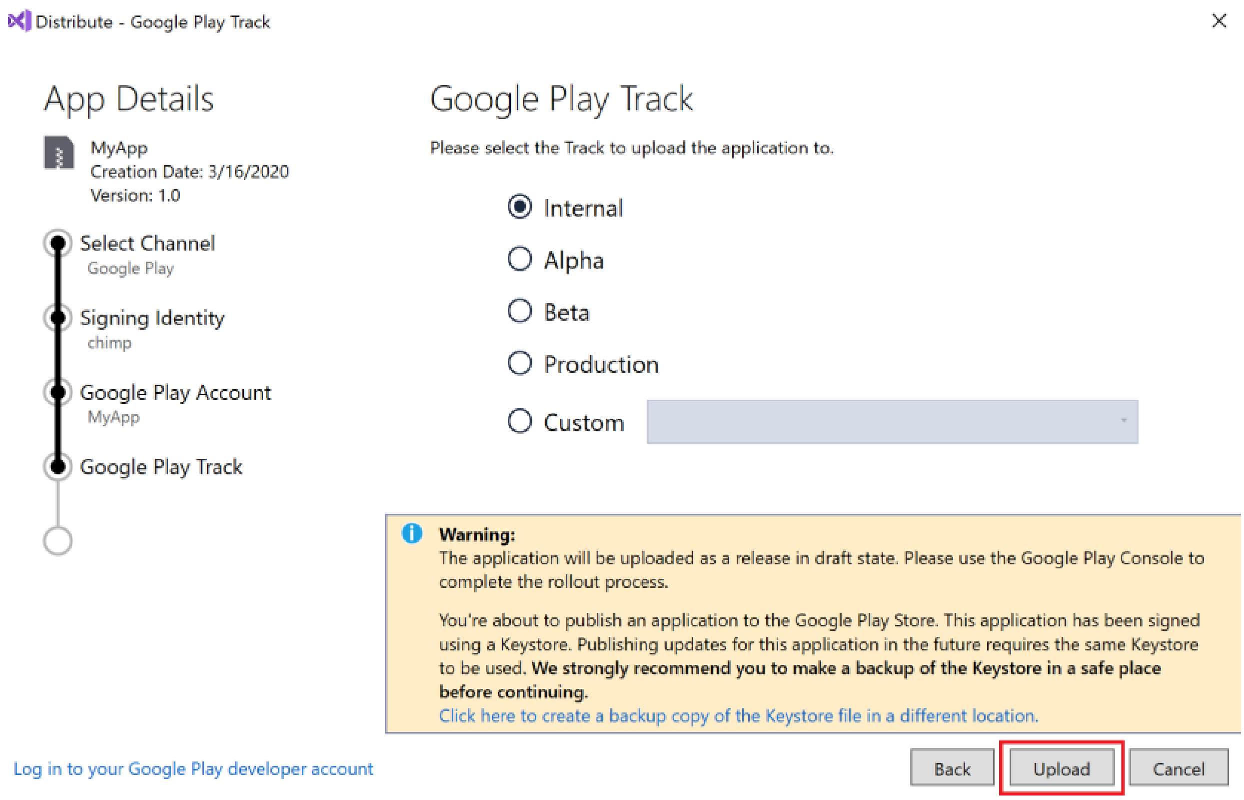
Google Play Accounts in Visual Studio. In the **Google Play Accounts** dialog, click **Continue**:



Next, the **Google Play Track** dialog is presented. Google Play offers five possible tracks for uploading your app:

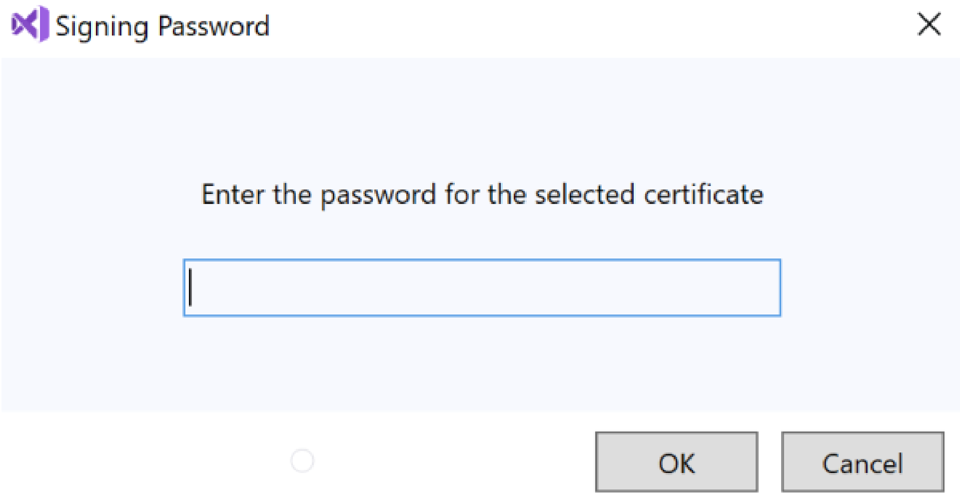
- **Internal** – Used for quickly distributing your app for internal testing and quality assurance checks.
- **Alpha** – Used for uploading an early version of your app to a small list of testers.
- **Beta** – Used for uploading an early version of your app to a larger list of testers.
- **Production** – Used for full distribution to the Google Play store.
- **Custom** – Used for testing pre-release versions of your app with specific users by creating a list of testers by email address.

Choose which Google Play track will be used for uploading the app and click **Upload**.

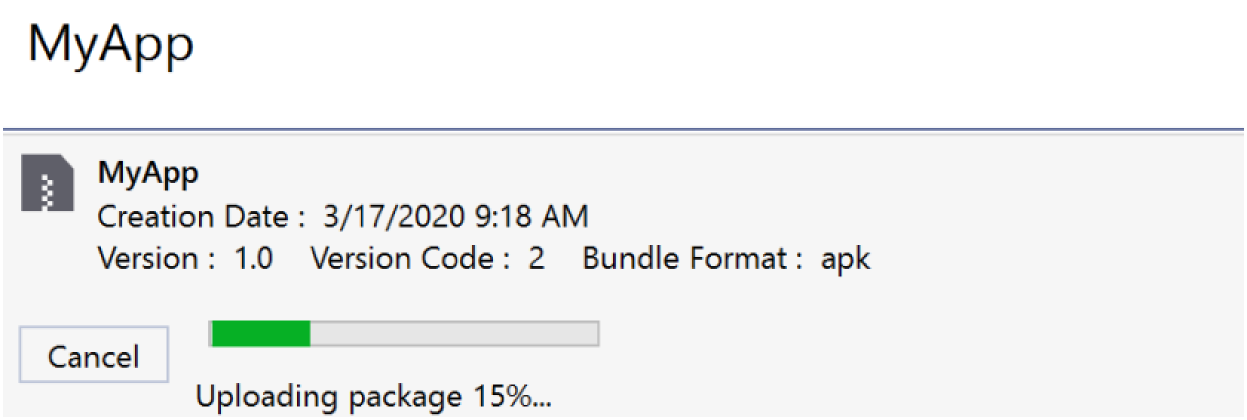


For more information about Google Play testing, see Set up open/closed/internal tests.

Next, a dialog is presented to enter the password for the signing certificate. Enter the password and click **OK**:



The **Archive Manager** displays the progress of the upload:

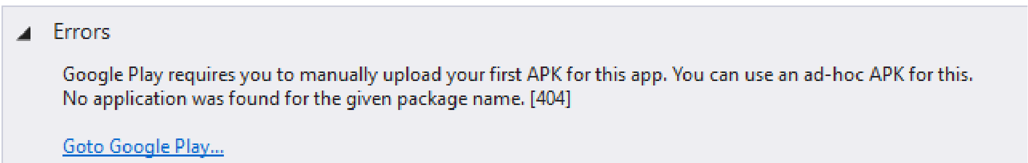


When the upload finishes, completion status is shown in the lower left hand corner of Visual Studio:

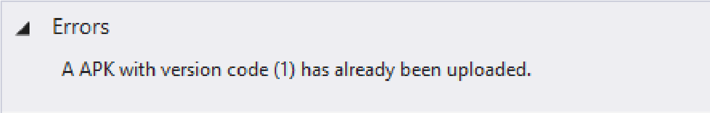
Troubleshooting

If you do not see your custom track when selecting a Google Play track, make sure you have created a release for that track on the Google Play Developer Console. For instructions on how to create a release, see Prepare & roll out releases.

Note that one APK must have already been submitted to the Google Play store before the **Publish to Google Play** will work. If an APK is not already uploaded the Publishing Wizard will display the following error in the **Errors** pane:



When this error occurs, manually upload an APK (such as an Ad Hoc build) via the Google Play Developer Console and use the **Distribution Channel** dialog for subsequent APK updates. For more information, see Manually Uploading the APK. The version code of the APK must change with each upload, otherwise the following error will occur:



To resolve this error, rebuild the app with a different version number and resubmit it to Google Play via the **Distribution Channel** dialog.

Google Licensing Services

7/8/2021 • 2 minutes to read • Edit Online

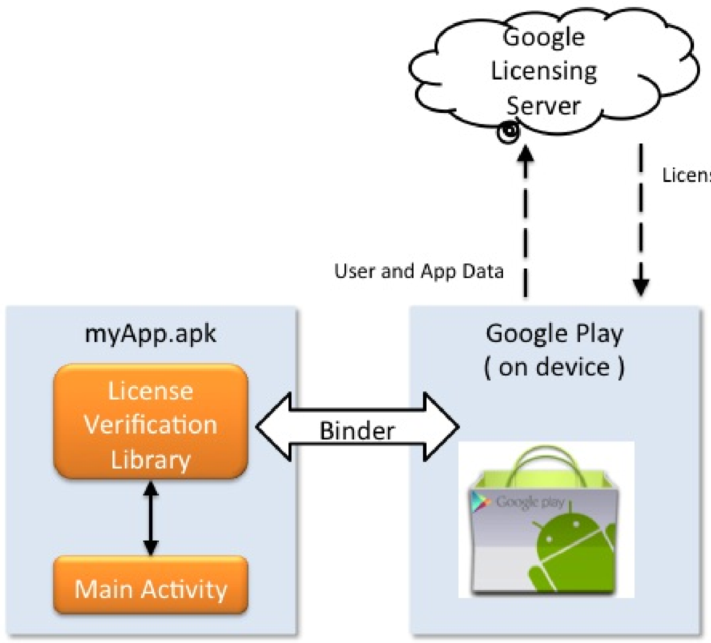
Prior to Google Play, Android applications relied on the legacy Copy Protection provided by Google Market to ensure that only authorized users could run applications on their devices. The limitations of the Copy Protection mechanism made it a less-than-ideal solution for application protection.

Google Licensing is a replacement for this legacy Copy Protection mechanism. Google Licensing is a flexible, secure, network-based service that Android applications may query to determine if an application is licensed to run on a given device.

Google Licensing is flexible in that Android applications have full control over when to check the license, how often to check the license, and how to handle the response from the licensing server.

Google Licensing is secure in that each response is signed using an RSA key pair that is shared exclusively between the Google Play server and the application. Google Play provides a public key for developers that is embedded within the Android application and is used to authenticate the responses. The Google Play server keeps the private key internally.

An application that has implemented Google Licensing makes a request to a service hosted by the Google Play application on the device. Google Play then sends this request on to the Google Licensing server, which responds with the license status:



The above diagram illustrates this workflow:

- The application provides the package name, a *nonce* (a cryptographic authenticator) that is used to validate server response, and a callback that can handle the response asynchronously.
- Google Play provides information such as the Google account and the device itself, such as the IMSI number.

Google Licensing service is also a key component of APK expansion files (which are discussed later in this document). APK expansion files utilize Google Licensing services to obtain the URLs of the expansion files that will be downloaded.

Requirements

Applications that are not purchased through Google Play will receive no benefit from the Google Licensing services. If Google Play is not installed on a device, then applications that use Licensing Services will still operate normally on that device.

Google Play requires Internet access for functionality. An application can cache the license to accommodate scenarios where the device does not have access to the Google Play Licensing servers.

Free applications only require Google Licensing when the application uses APK expansion files.

APK Expansion Files

7/8/2021 • 6 minutes to read • Edit Online

Some applications (some games, for instance) require more resources and assets than can be provided in the maximum Android app size limit imposed by Google Play. This limit depends on the version of Android that your APK is targeted for:

- 100MB for APKs that target Android 4.0 or higher (API level 14 or higher).
- 50MB for APKs that target Android 3.2 or lower (API level 13 or higher).

To overcome this limitation, Google Play will host and distribute two *expansion files* to go along with an APK, allowing an application to indirectly exceed this limit.

On most devices, when an application is installed, expansion files will be downloaded along with the APK and will be saved to the shared storage location (the SD card or the USB-mountable partition) on the device. On a few older devices, the expansion files may not automatically install with the APK. In these situations, it is necessary for the application to contain code that will download the expansion files when the user first runs the applications.

Expansion files are treated as *opaque binary blobs (obb)* and may be up to 2GB in size. Android does not perform any special processing on these files after they are downloaded – the files can be in any format that is appropriate for the application. Conceptually, the recommended approach to expansion files is as follows:

- **Main expansion** – This file is the primary expansion file for resources and assets that will not fit in the APK size limit. The main expansion file should contain the primary assets that an application needs and should rarely be updated.
- **Patch expansion** – This is intended for small updates to the main expansion file. This file can be updated. It is the responsibility of the application to perform any necessary patches or updates from this file.

The expansion files must be uploaded at the same time as the APK is uploaded. Google play does not allow an expansion file to be uploaded to an existing APK or for existing APKs to be updated. If it is necessary to update an expansion file, then a new APK must be uploaded with the `versionCode` updated.

Expansion File Storage

When the files are downloaded to a device, they will be stored in ***shared-store*/Android/obb/*package-name***:

- ***shared-store*** – This is the directory specified by `Android.OS.Environment.ExternalStorageDirectory` .
- ***package-name*** – This is the application's Java-style package name.

Once downloaded, expansion files should not be moved, altered, renamed, or deleted from their location on the device. To do so will cause the expansion files to be downloaded again, and the old file(s) will be deleted. Additionally, the expansion file directory should contain only the expansion pack files.

Expansion files offer no security or protection around their content – other applications or users may access any files saved on the shared storage.

If it is necessary to unpack an expansion file, the unpacked files should be stored in a separate directory, such as one in `Android.OS.Environment.ExternalStorageDirectory` .

An alternative to extracting files from an expansion file is to read the assets or resources directly from the expansion file. The expansion file is nothing more than a zip file that can be used with an appropriate

`ContentProvider`. The Android.Play.ExpansionLibrary contains an assembly, System.IO.Compression.Zip, which includes a `ContentProvider` that will allow for direct file access to some media files. If media files are being packaged into a zip file, media playback calls may directly use files in the zip without having to unpack the zip file. The media files should not be compressed when added to the zip file.

FileName Format

When the expansion files are downloaded, Google Play will use the following scheme to name the expansion:

```
[main|patch].<expansion-version>.<package-name>.obb
```

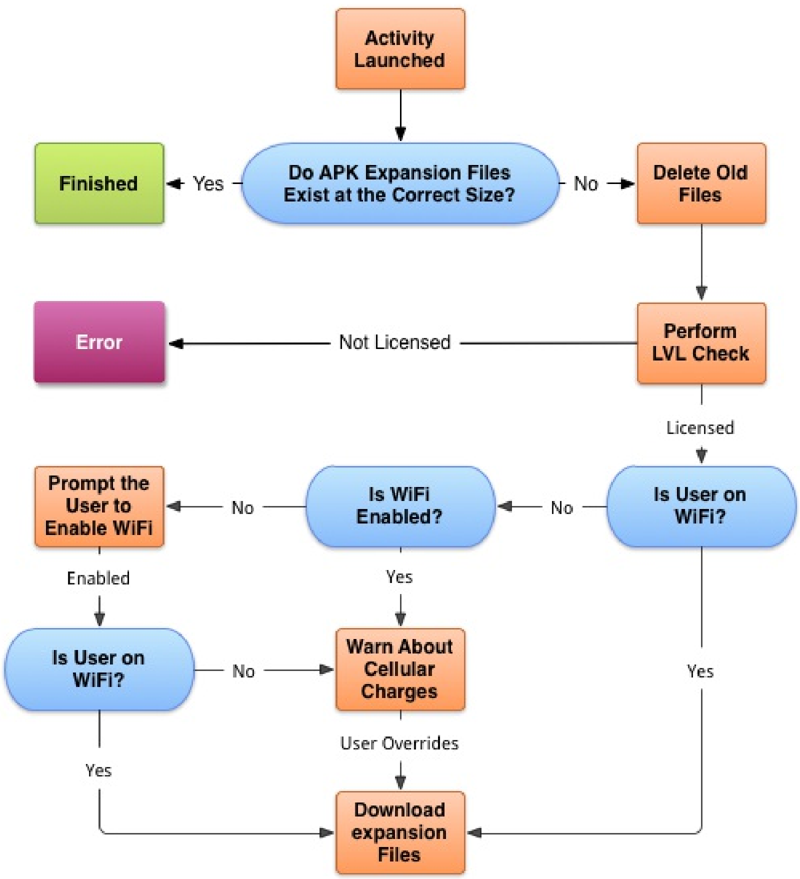
The three components of this scheme are:

- `main` or `patch` – This specifies whether this is the main or patch expansion file. There can be only one of each.
- `<expansion-version>` – This is an integer that matches the `versionCode` of the APK that the file was first associated with.
- `<package-name>` – This is the application's Java-style package name.

For example, if the APK version is 21, and the package name is `mono.samples.helloworld`, the main expansion file will be named **main.21.mono.samples.helloworld**.

Download Process

When an application is installed from Google Play, the expansion files should be downloaded and saved along with the APK. In certain situations this may not happen, or expansion files may be deleted. To handle this condition, an app needs to check to see whether the expansion files exist and then download them, if necessary. The following flowchart displays the recommended workflow of this process:



When an application starts up, it should check to see if the appropriate expansion files exist on the current device. If they do not, then the application must make a request from Google Play's Application Licensing. This check is made by using the *License Verification Library (LVL)*, and must be made for both free and licensed applications. The LVL is primarily used by paid applications to enforce license restrictions. However, Google has extended the LVL so that it can be used with expansion libraries as well. Free applications have to perform the LVL check, but can ignore the license restrictions. The LVL request is responsible for providing the following information about the expansion files that the application requires:

- **File Size** – The file sizes of the expansion files are used as part of the check that determines whether or not the correct expansion files have already been downloaded.
- **Filenames** – This is the file name (on the current device) to which the expansion packs must be saved.
- **URL for Download** – The URL that should be used to download the expansion packs. This is unique for every download and will expire shortly after it is provided.

After the LVL check has been performed, the application should download the expansion files, taking into consideration the following points as part of the download:

- The device may not have enough space to store the expansion files.
- If Wi-Fi is not available, then the user should be allowed to pause or cancel the download to prevent unwanted data charges.
- The expansion files are downloaded in the background to avoid blocking user interactions.
- While the download is occurring in the background, a progress indicator should be displayed.
- Errors that occur during the download are gracefully handled and recoverable.

Architectural Overview

When the main activity starts, it checks to see if the expansion files are downloaded. If the files are downloaded, they must be checked for validity.

If the expansion files have not been downloaded or if the current files are invalid, then new expansion files must be downloaded. A bounded service is created as part of the application. When the main activity of the application is started, it uses the bounded service to perform a check against the Google Licensing services to find out the expansion file names and the URL of the files to download. The bounded service will then download the files on a background thread.

To ease the effort required to integrate expansion files into an application, Google created several libraries in Java. The libraries in question are:

- **Downloader Library** – This is a library that reduces the effort required to integrate expansion files in an application. The library will download the expansion files in a background service, display user notifications, handle network connectivity issues, resume downloads, etc.
- **License Verification Library (LVL)** – A library for making and processing the calls to the Application Licensing services. It can also be used to perform licensing checks, to see if the application is authorized for use on the device.
- **APK Expansion Zip Library (optional)** – If the expansion files are in a zip file, this library will act as a content provider and allow an application to read resources and assets directly from the zip file without having to expand the zip file.

These libraries have been ported to C# and are available under the Apache 2.0 license. To quickly integrate expansion files into an existing application, these libraries can be added to an existing Xamarin.Android application. The code is available at the Android.Play.ExpansionLibrary on GitHub.

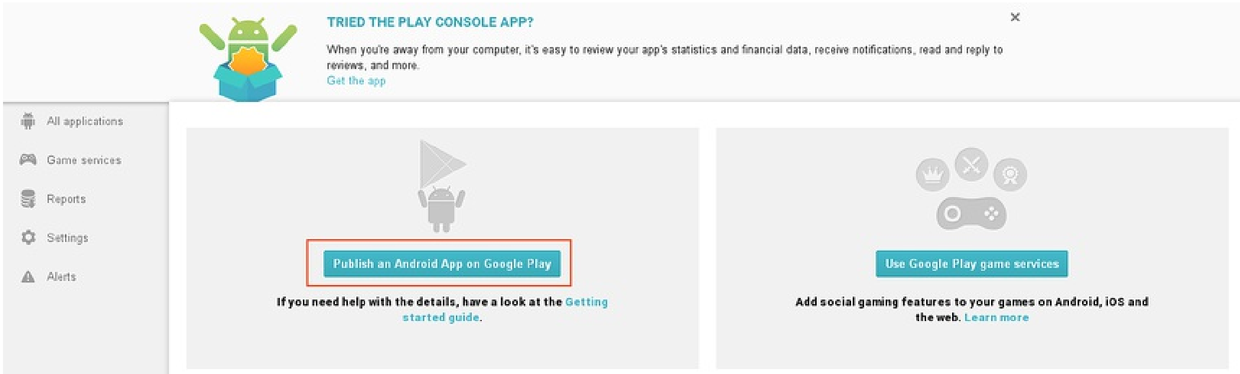
Manually Uploading the APK

7/8/2021 • 7 minutes to read • Edit Online

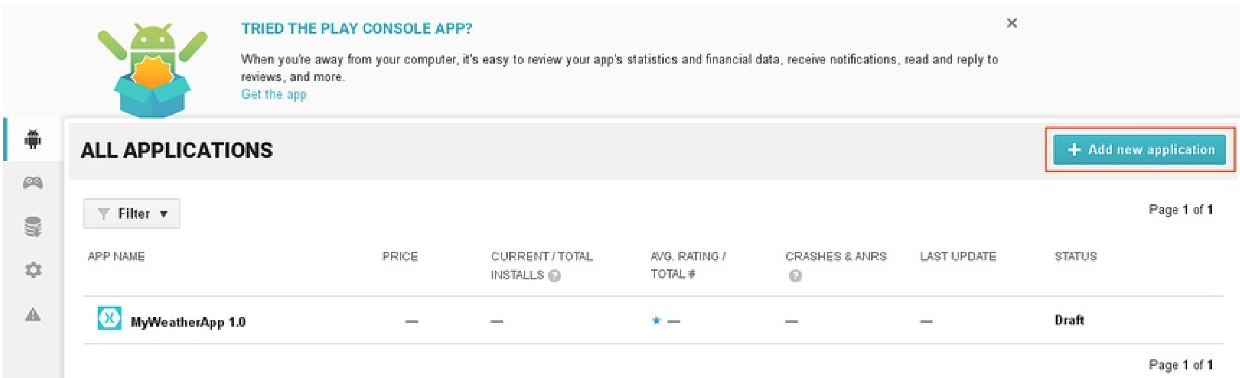
The first time an APK is submitted to Google Play (or if an early version of Xamarin.Android is used) the APK must be manually uploaded through the Google Play Developer Console. This guide explains the steps required for this process.

Google Play Developer Console

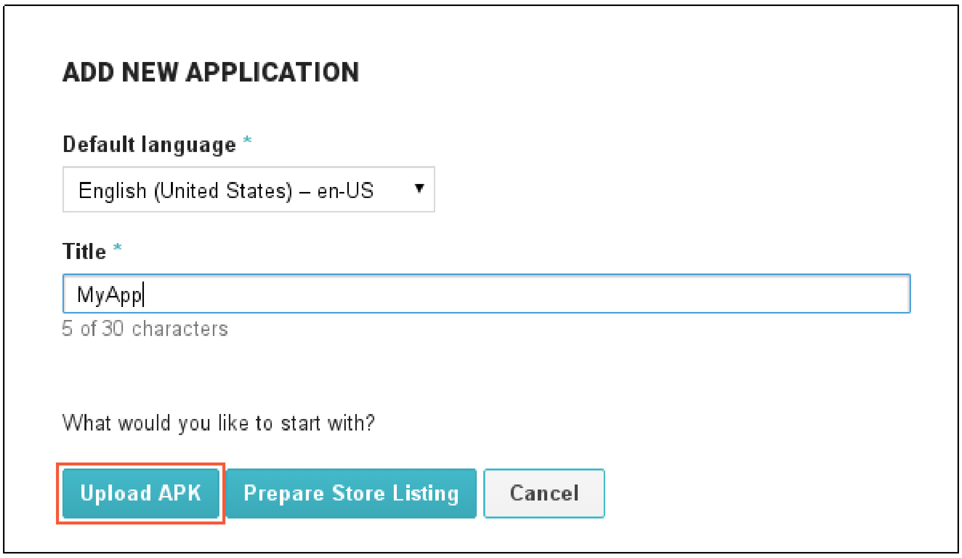
Once the APK has been compiled and the promotional assets prepared, the application must be uploaded to Google Play. This is done by logging in to the Google Play Developer Console, pictured next. Click the **Publish an Android App on Google Play** button to initialize the process of distributing an application.



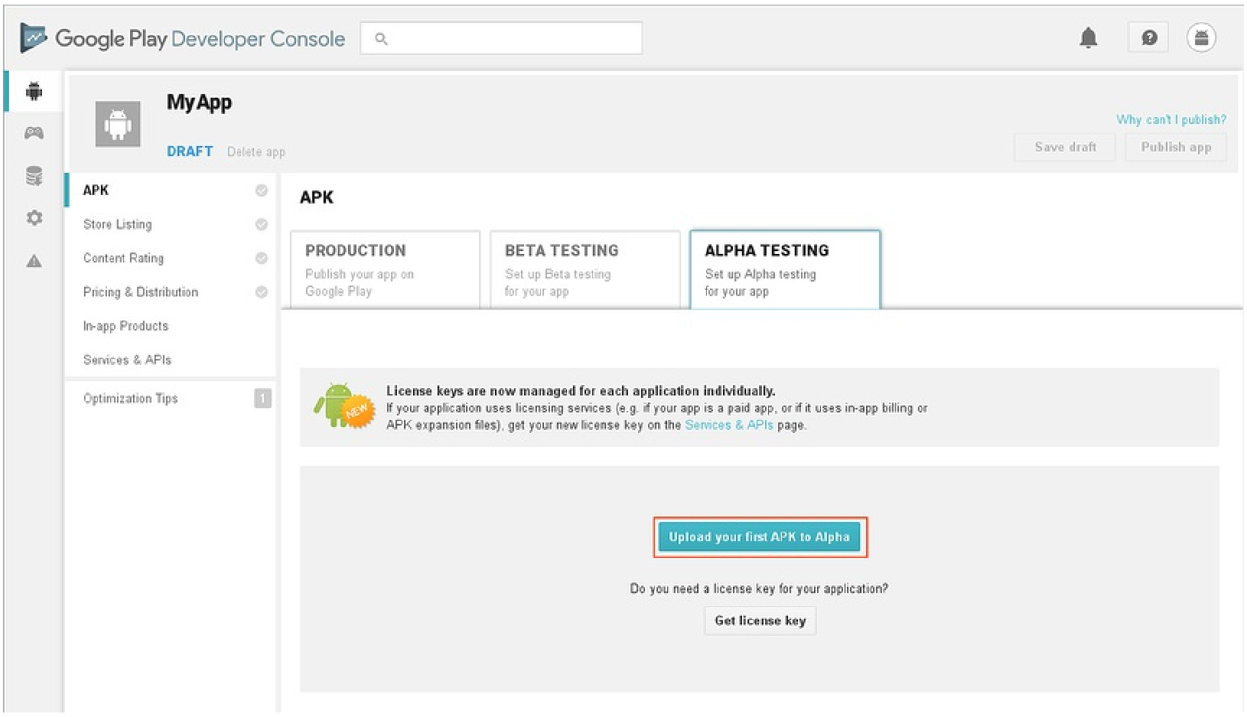
If you already have an existing app registered with Google Play, click the **Add new application** button:



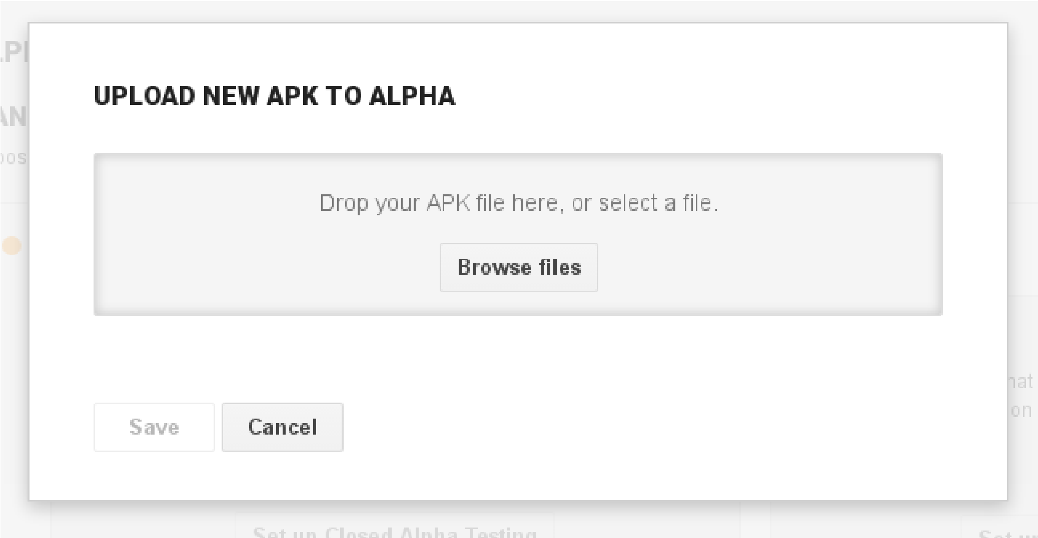
When the **ADD NEW APPLICATION** dialog is displayed, enter the name of the app and click **Upload APK**:



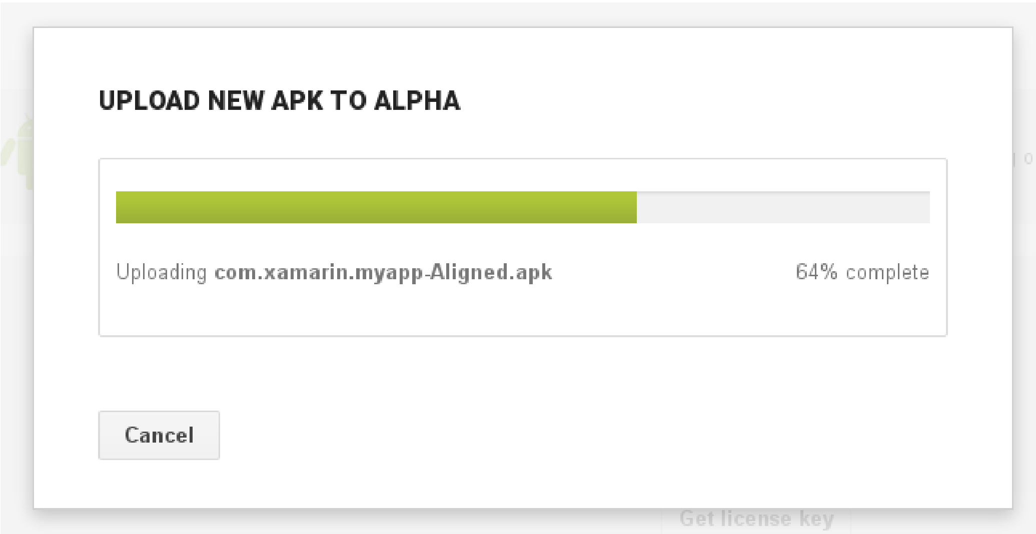
The next screen allows the app to be published for alpha testing, beta testing, or production. In the following example, the **ALPHA TESTING** tab is selected. Because **MyApp** does not use licensing services, the **Get license key** button does not have to be clicked for this example. Here, the **Upload your first APK to Alpha** button is clicked to publish to the Alpha channel:



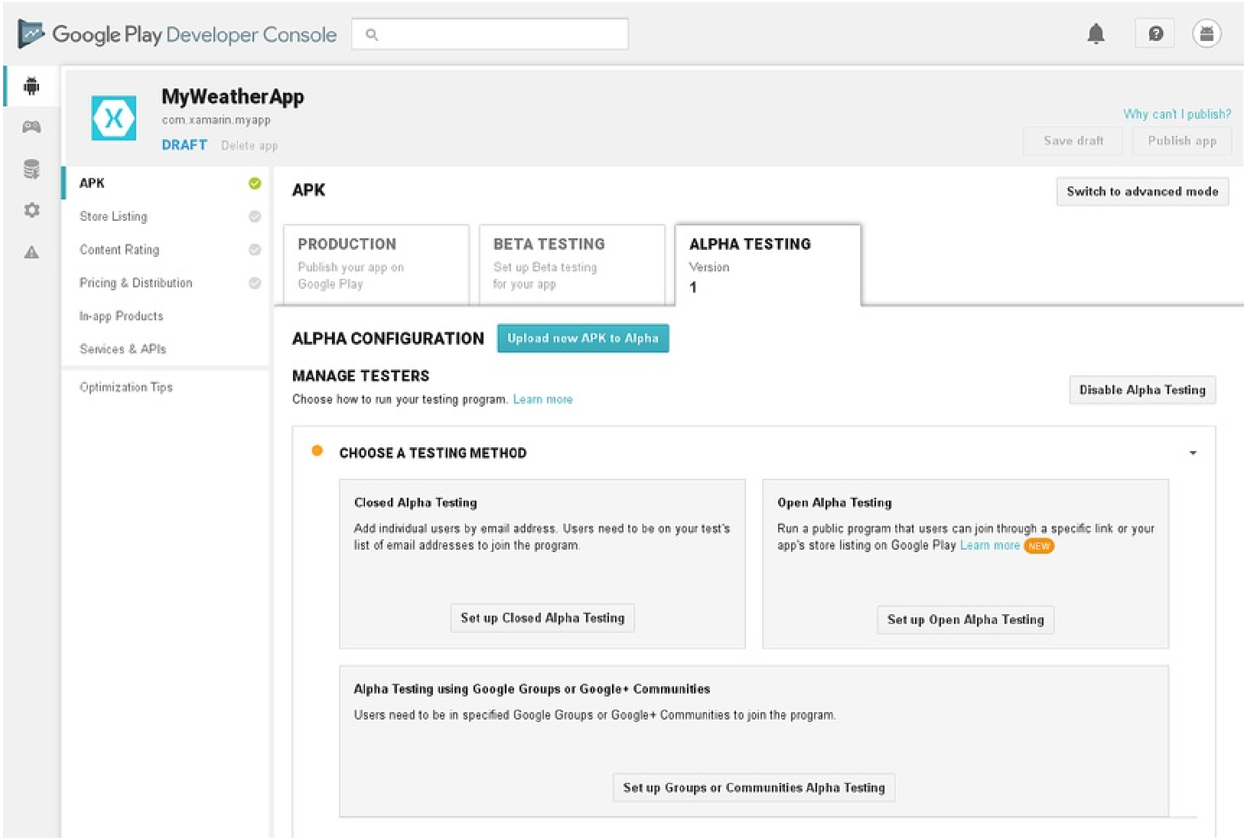
The **UPLOAD NEW APK TO ALPHA** dialog is displayed. The APK can be uploaded by either clicking the **Browse files** button or by dragging-and-dropping the APK:



Be sure to upload the release-ready APK that is to be distributed. The next dialog indicates the progress of the APK upload:



After the APK is uploaded, it is possible to select a testing method:

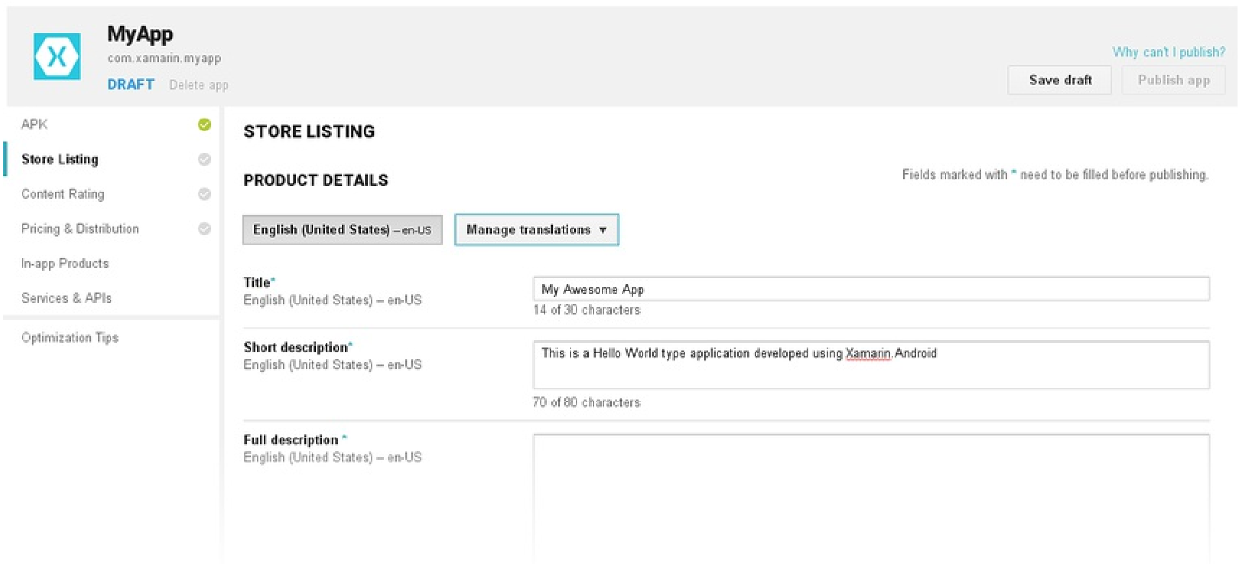


For more information about app testing, see Google's Set up alpha/beta tests guide.

After the APK is uploaded, it is saved as a draft. It cannot be published until more details are provided to Google Play as described next.

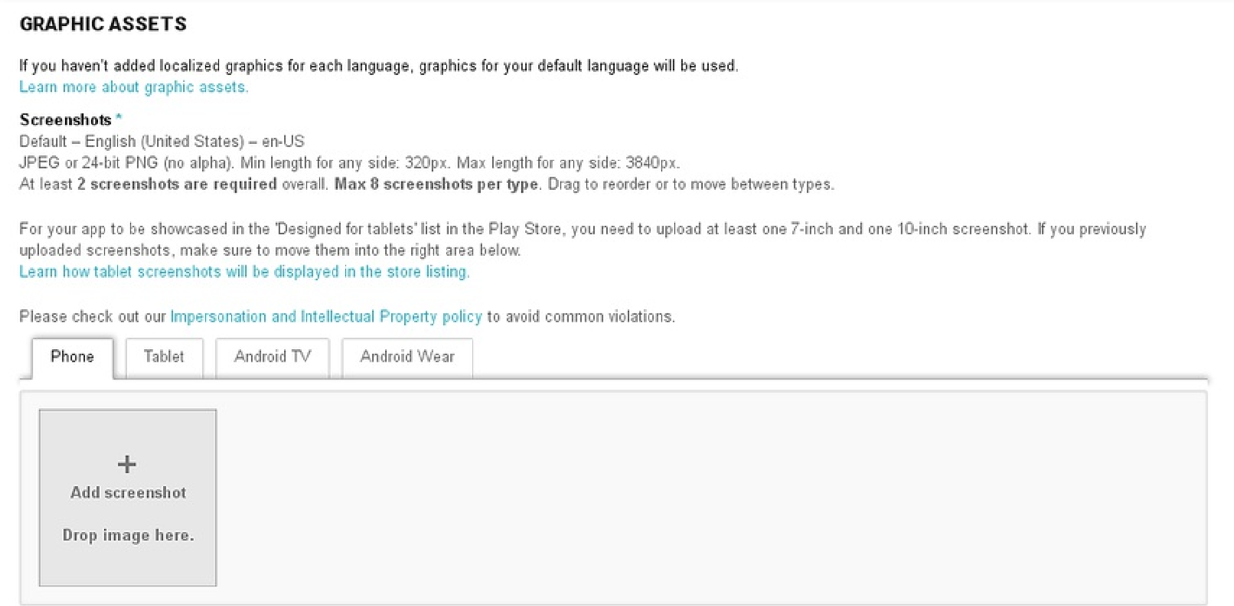
Store Listing

Click **Store Listing** in the **Google Play Developer Console** to enter the information that Google Play will display to potentials users of the application:



Graphics Assets

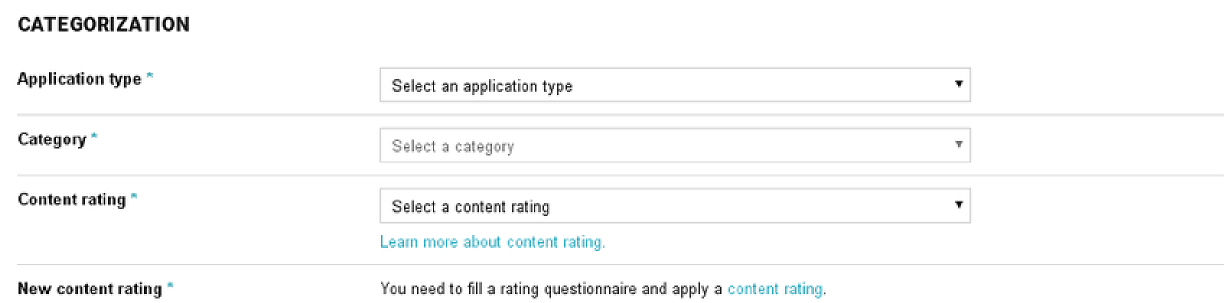
Scroll down to the **GRAPHICS ASSETS** section of the **Store Listing** page:



All of the promotional assets that were prepared earlier are uploaded in this section. Guidance is provided as to what promotional assets must be provided and what format they should be provided in.

Categorization

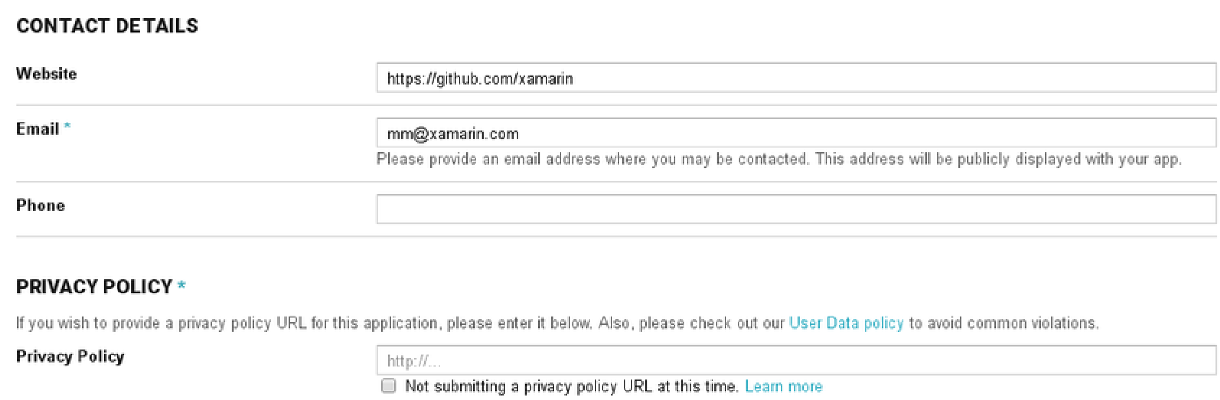
After the **GRAPHICS ASSETS** section is a **CATEGORIZATION** section, select the application type and category:



Content rating is covered after the next section.

Contact Details

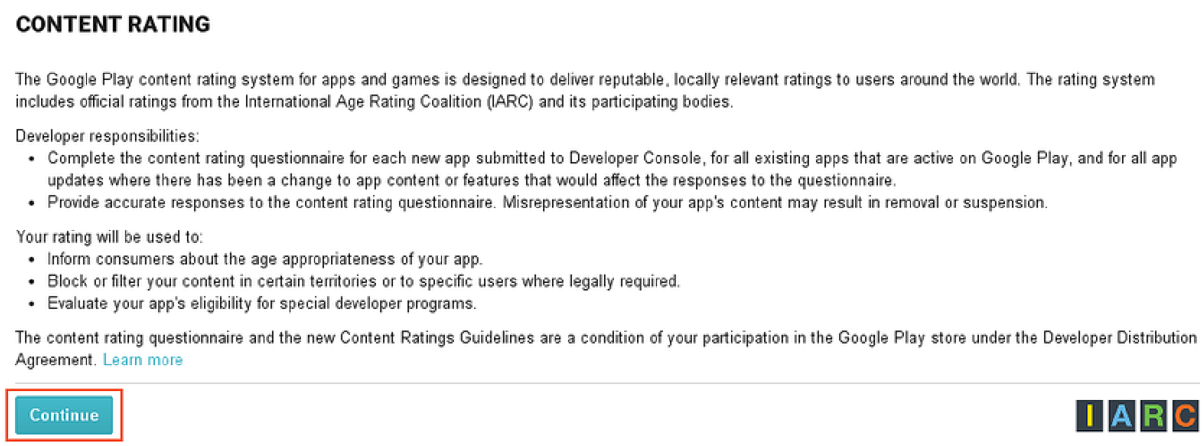
The final section of this page is a **CONTACT DETAILS** section. This section is used to collect contact information about the developer of the application:



It is possible to provide a URL for the privacy policy of the App in the **PRIVACY POLICY** section, as indicated above.

Content Rating

Click **Content Rating** in the **Google Play Developer Console**. In this page, you specify the content rating for your app. Google Play requires that all applications specify a content rating. Click the **Continue** button to complete the content rating questionaire:



All applications on Google Play must be rated according to the Google Play ratings system. In addition to the content rating, all applications must adhere to Google's Developer Content Policy.

The following lists the four levels in the Google Play rating system and provides some guidelines as features or content that would require or force the rating level:

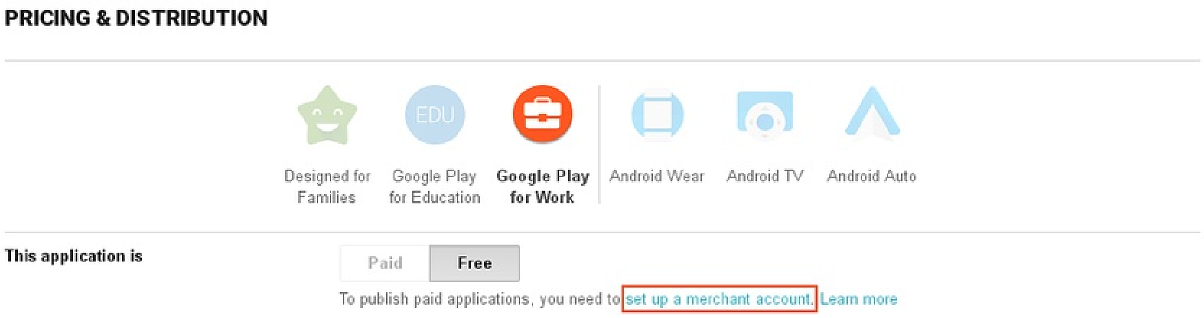
- **Everyone** – May not access, publish, or share location data. May not host any user-generated content. May not enable communication between users.
- **Low maturity** – Applications that access, but do not share, location data. Depictions of mild or cartoon violence.
- **Medium maturity** – References to drugs, alcohol or tobacco. Gambling themes or simulated gambling. Inflammatory content. Profanity or crude humor. Suggestive or sexual references. Intense fantasy violence. Realistic violence. Allowing users to find each other. Allowing users to communicate with each other. Sharing of a user's location data.
- **High maturity** – A focus on the consumption or sale of alcohol, tobacco, or drugs. A focus on suggestive or sexual references. Graphic violence.

The items in the Medium maturity list are subjective, as such it is possible that a guideline that may seem to dictate a Medium maturity rating may be intense enough to warrant a High maturity rating.

Pricing & Distribution

Click **Pricing and Distribution** in the **Google Play Developer Console**. In this page, set a price if the app is a paid app. Alternately, the application can be distributed free of charge to all users. Once an application is specified as free, it must remain free. Google Play will not allow an application that is free to be changed to a priced app (however, it is possible to sell content with in-app billing with a free app). Google Play will allow a paid app to change to a free app at any time.

A merchant account is required to before publishing a paid app.To do so, click **set up a merchant account** and follow the instructions.

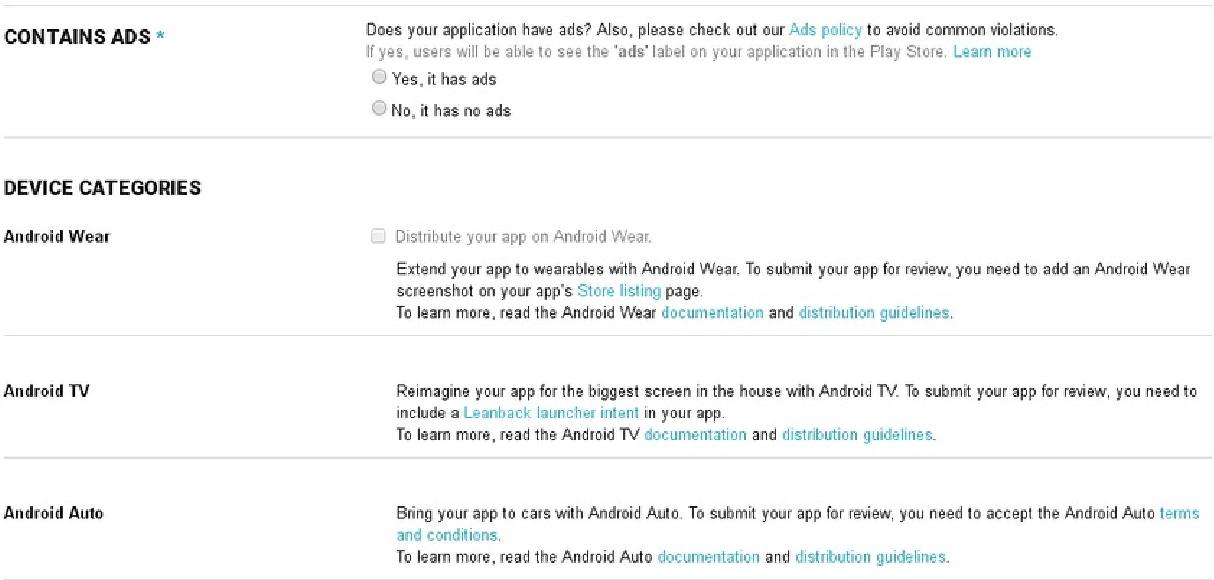


Manage Countries

The next section, **Manage Countries**, provides control over what countries an app may be distributed to:

Other Information

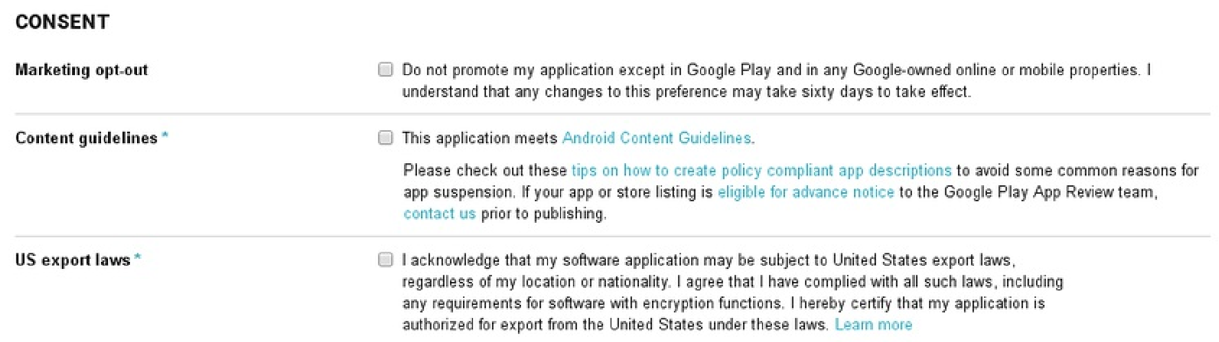
Scroll down further to specify whether the app contains ads. Also, the **DEVICE CATEGORIES** section provides options to optionally distribute the app for Android Wear, Android TV, or Android Auto:



After this section are additional options that may be selected, such as opting into **Designed for Families** and distributing the app through Google Play for Education.

Consent

At the bottom of the **Pricing & Distribution** page is the **CONSENT** section. This is a mandatory section and is used to declare that the application meets the Android Content Guidelines and acknowledgement that the application is subject to U.S. export laws:



There is much more to publishing a Xamarin.Android app than can be covered in this guide. For more information about publishing your app in Google Play, see Welcome to the Google Play Developer Console Help Center.

Google Play Filters

When users browse the Google Play website for applications, they are able to search all published applications. When users browse Google Play from an Android device, the results are slightly different. The results will be filtered according to compatibility with the device that is being used. For example, if an application must send SMS messages, then Google Play will not show that application to any device which cannot send SMS messages. The filters that are applied to a search are created from the following:

1. The hardware configuration of the device.
2. Declarations in the applications manifest file.
3. The carrier that is used (if any).
4. The location of the device.

It is possible to add elements to the app's manifest to help control how app is filtered in the Google Play store. The following lists manifest elements and attributes that can be used to filter applications:

- supports-screen – Google Play will use the attributes to determine if an application can be deployed to a device based on the screen size. Google Play will assume that Android can adapt smaller layout to larger screens, but not vice-versa. So an application that declares support for normal screens will appear in searches for large screens, but not small screens. If a Xamarin.Android application does not provide a `<supports-screen>` element in the manifest file, then Google Play will assume all attributes have a value of true and that the application supports all screen sizes. This element must be added to **AndroidManifest.xml** manually.

- uses-configuration – This manifest element is used to request certain hardware features, such as the type of keyboard, navigation devices, a touch screen, etc. This element must be added to **AndroidManifest.xml** manually.

- uses-feature – This manifest element declares hardware or software features that a device must have in order for the application to function. This attribute is informational only. Google Play will not display the application to devices that do not meet this filter. It's still possible to install the application by other means (manually or downloading). This element must be added to **AndroidManifest.xml** manually.

- uses-library – This element specifies that certain shared libraries must be present on the device, for example Google Maps. This element may also be specified with the `Android.App.UsesLibraryAttribute`. For example:

```
[assembly: UsesLibrary("com.google.android.maps", true)]
```

- uses-permission – This element is used to infer certain hardware features that are required for the application to run that may not have been properly declared with a `<uses-feature>` element. For example, if an application requests permission to use the camera, then Google Play assumes that devices must have a camera, even if there is no `<uses-feature>` element declaring the camera. This element may be set with the `Android.App.UsesPermissionsAttribute`. For example:

```
[assembly: UsesPermission(Manifest.Permission.Camera)]
```

- uses-sdk – The element is used to declare the minimum Android API Level required for the application. This element may set in the Xamarin.Android options of a Xamarin.Android project.

- compatible-screens – This element is used to filter applications that do not match the screen size and density specified by this element. Most applications should not use this filter. It is intended for specific high performance games or applications that required strict controls on application distribution. The `<support-screen>` attribute mentioned above is preferred.

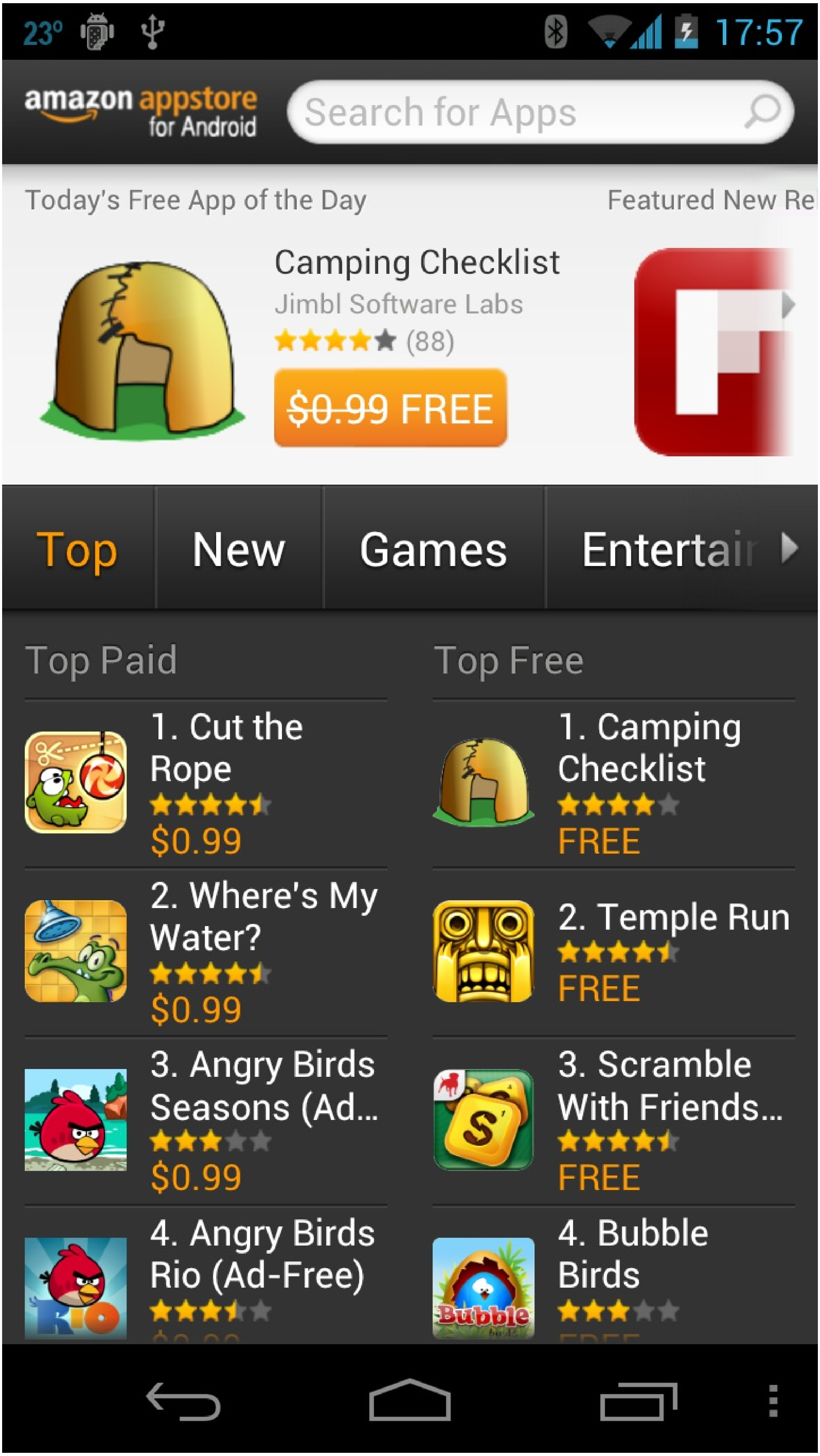
- supports-gl-texture – This element is used to declare GL texture compression formations that the application requires. Most applications should not use this filter. It is intended for specific high performance games or applications that required strict controls on application distribution.

For more information about configuring the app manifest, see the Android App Manifest topic.

Publishing to the Amazon App Store

7/8/2021 • 2 minutes to read • Edit Online

The Amazon Mobile App Distribution Program enables mobile app developers to publish their applications on Amazon. This section briefly covers the Amazon App Store for Android.


23°
17:57
amazon appstore for Android
Search for Apps
Today's Free App of the Day
Featured New Re
Camping Checklist
Jimbl Software Labs
(88)
$0.99 FREE
Top
New
Games
Entertai
Top Paid
Top Free
1. Cut the Rope
$0.99
1. Camping Checklist
FREE
2. Where's My Water?
$0.99
2. Temple Run
FREE
3. Angry Birds Seasons (Ad...
$0.99
3. Scramble With Friends...
FREE
4. Angry Birds Rio (Ad-Free)
4. Bubble Birds

Amazon does not limit the size of APKs. However, if an APK is larger than 30MB, then it will use FTP for distribution rather than the Amazon Mobile App Distribution Portal.

Submitting Apps: Binary Info

Submitting an application to the Amazon App Store is a similar process to submitting an application to Google Play. Applications distributed by Amazon require the following assets:

- **Icon** – This is a 114 x 114 .png file with a transparent background. It is required.
- **Thumbnail** – This is a larger version of the icon above. It is 512 x 512 pixels with a transparent background. This icon is also mandatory.
- **Screenshots** – Amazon requires a minimum of three and a maximum of 10 screenshots. The screenshots must be 1024w x 600h pixels or 800w x 480h pixels. Both .png and .jpg formats are acceptable.
- **Promotional Image** – In order for an application to be featured in promotional placements such as the home page, a promotional image may be optionally submitted. It should be a 1024w x 500h pixel .png or .jpg file, in landscape orientation. It may not have any animation.
- Updates to five videos may be provided.

Approval Process

Once an application has been submitted, it goes through an approval process. Amazon will review your application to ensure that it works as outlined in the product description, does not put customer data at risk, and will not impair the operation of the device. Once the approval process is complete, Amazon will send out a notification and distribute the application.

Publishing Independently

7/8/2021 • 3 minutes to read • Edit Online

It is possible to publish an application without using any of the existing Android marketplaces. This section will explain these other publishing methods and the licensing levels of Xamarin.Android.

Xamarin Licensing

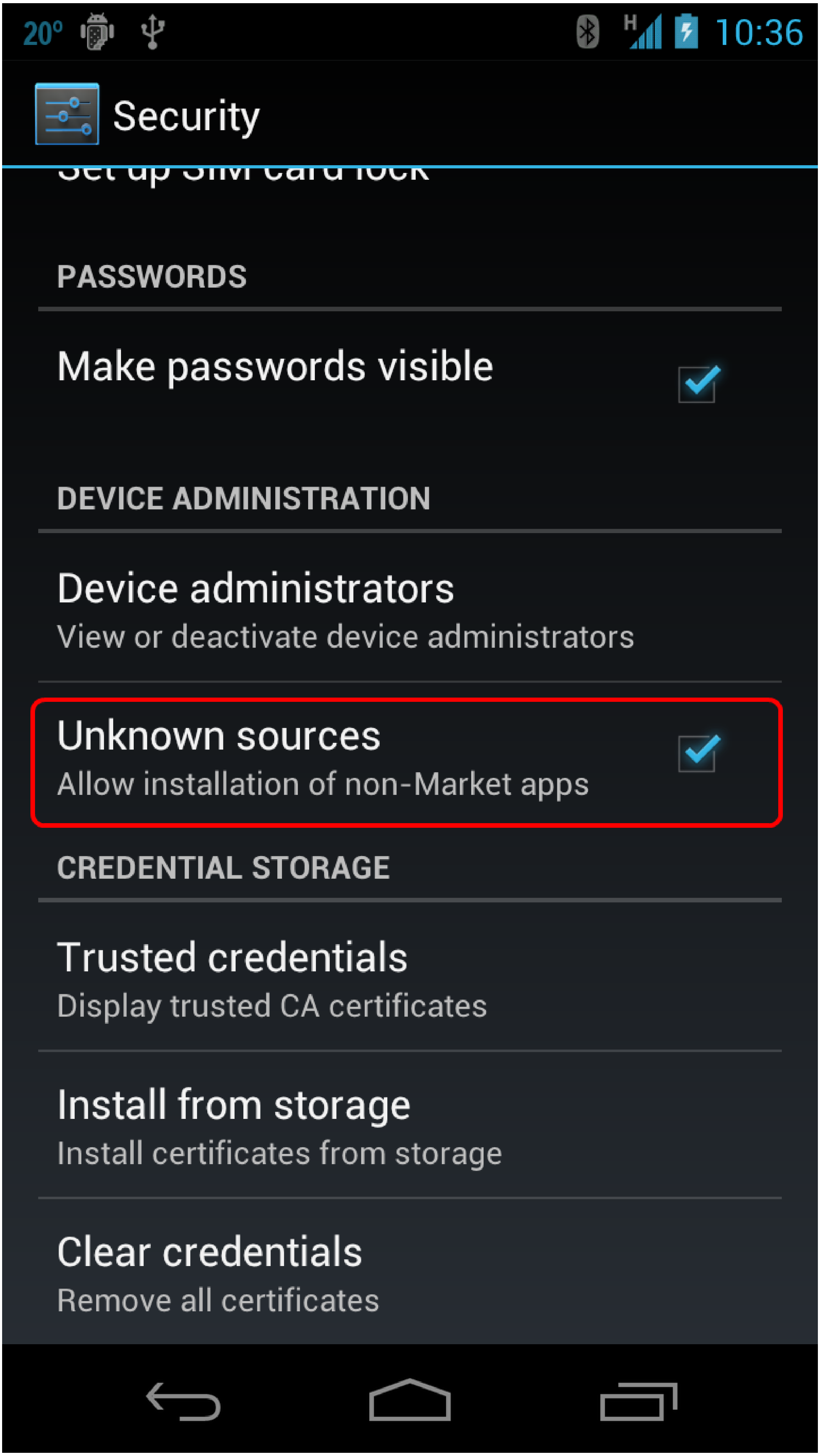
A number of licenses are available for development, deployment, and distribution of Xamarin.Android apps:

- **Visual Studio Community** – For students, small teams, and OSS developers who use Windows.
- **Visual Studio Professional** – For individual developers or small teams (Windows only). This license offers a standard or cloud subscription and no usage restrictions.
- **Visual Studio Enterprise** – For teams of any size (Windows only). This license includes enterprise capabilities, a standard or cloud subscription.

Visit the visualstudio.com to download the Community Edition or to learn more about purchasing the Professional and Enterprise editions.

Allow Installation from Unknown Sources

By default, Android prevents users from downloading and installing applications from locations other than Google Play. To allow installation from non-marketplace sources, a user must enable the *Unknown sources* setting on a device before attempting to install an application. The setting for this may be found under **Settings > Security**, as shown in the following diagram:

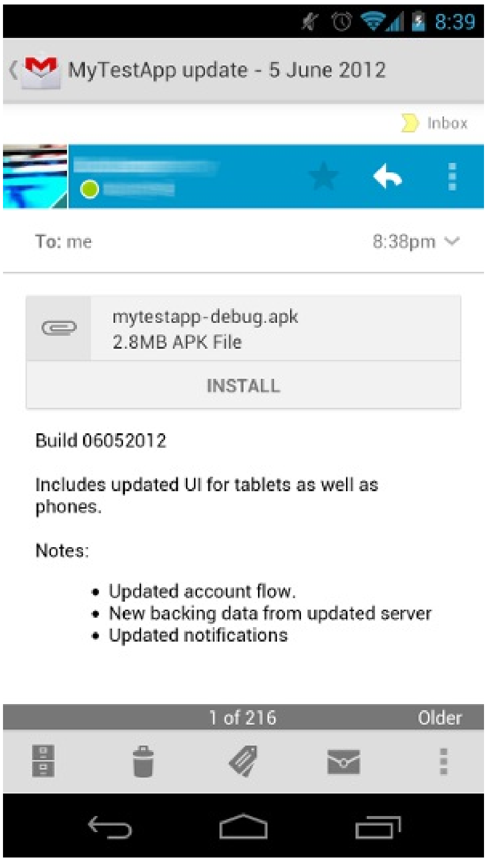

20°
10:36
Security
PASSWORDS
Make passwords visible
DEVICE ADMINISTRATION
Device administrators
View or deactivate device administrators
Unknown sources
Allow installation of non-Market apps
CREDENTIAL STORAGE
Trusted credentials
Display trusted CA certificates
Install from storage
Install certificates from storage
Clear credentials
Remove all certificates

IMPORTANT

Some network providers might prevent the installation of applications from unknown sources, regardless of this setting.

Publishing by E-Mail

Attaching the release APK to an e-mail is a quick and easy way to distribute an application to users. When the user opens the e-mail on an Android-powered device, Android will recognize the APK attachment and display an **Install** button as shown in the following image:



Although distribution via e-mail is simple, it provides few protections against piracy or unauthorized distribution. It is best reserved for situations where the recipients of the application are few, and they are trusted not to distribute the application.

Publishing by Web

It is possible to distribute an application by a web server. This is accomplished by uploading the application to the web server, and then providing a download link to users. When an Android-powered device browses to a link and then downloads the application, that application will automatically be installed once the download is complete.

Manually Installing an APK

Manual installation is a third option for installing applications. To effect a manual installation of an application:

1. **Distribute a copy of the APK to user** – For example, this copy may be distributed on a CD or USB flash drive.
2. **(The user) installs the application on an Android device** – Use the command-line *Android Debug Bridge* (**adb**) tool. **adb** is a versatile command-line tool that enables communication with either an emulator instance or an Android-powered device. The Android SDK includes **adb**; it can be found in the directory

<sdk>/platform-tools/.

The Android device must be connected with a USB cable to the computer. Windows computers might also require additional USB drivers from the phone vendor to be recognized by **adb**. Installation instructions for these additional USB drivers is beyond the scope of this document.

Before issuing any **adb** commands, it is helpful to know which emulator instances or devices are connected, if any. It is possible to see a list of what is attached by using the `devices` command, as demonstrated in the following snippet:

```
$ adb devices
List of devices attached
        0149B2EC03012005device
```

After the connected devices have been confirmed, the application can be installed by issuing the `install` command with **adb**:

```
$ adb install <path-to-apk>
```

The following snippet shows an example of installing an application to a connected device:

```
$ adb install helloworld.apk
3772 KB/s (3013594 bytes in 0.780s)
        pkg: /data/local/tmp/helloworld.apk
Success
```

If the application is already installed, the `adb install` will be unable to install the APK and will report a failure, as shown in the following example:

```
$ adb install helloworld.apk
4037 KB/s (3013594 bytes in 0.728s)
        pkg: /data/local/tmp/helloworld.apk
Failure [INSTALL_FAILED_ALREADY_EXISTS]
```

It will be necessary to uninstall the application from the device. First, issue the `adb uninstall` command:

```
adb uninstall <package_name>
```

The following snippet is an example of uninstalling an application:

```
$ adb uninstall mono.samples.helloworld
Success
```

Installing Xamarin.Android as a System App

7/8/2021 • 3 minutes to read • Edit Online

This guide will discuss the differences between a system app and a user app, and how to install a Xamarin.Android application as a system application. This guide applies to authors of custom Android ROM images. It will not explain how to create a custom ROM.

System App

Authors of custom Android ROM images or manufacturers of Android devices may wish to include a Xamarin.Android application as a *system app* when distributing a ROM or a device. A system app is an app that is considered to be important to the functioning of the device or provide functionality that the custom ROM author always wants to be available.

System apps are installed in the folder **/system/app/** (a read-only directory on the file system) and cannot be deleted or moved by the user unless that user has root access. In contrast, an application that is installed by the user (typically from Google Play or by sideloading the app) is known as a *user app*. User apps can be deleted by the user and in many cases can be moved to a different location on the device (such as some kind of external storage).

System apps behave exactly like user apps, but have the following notable exceptions:

- System apps are upgradable just like a normal *user app*. However, because a copy of the app always exists in **/system/app/**, it is always possible to roll back the application to the original version.
- System apps may be granted certain system-only permissions that are not available to a user app. An example of a system-only permission is `BLUETOOTH_PRIVILEGED`, which allows applications to pair with Bluetooth devices without any user interaction.

It is possible to distribute a Xamarin.Android app as a system application. In addition to providing an APK to the custom ROM, there are two shared libraries, **libmonodroid.so** and **libmonosgen-2.0.so** that must be manually copied from the APK to the filesytem of the ROM image. This guide will explain the steps involved.

Restrictions

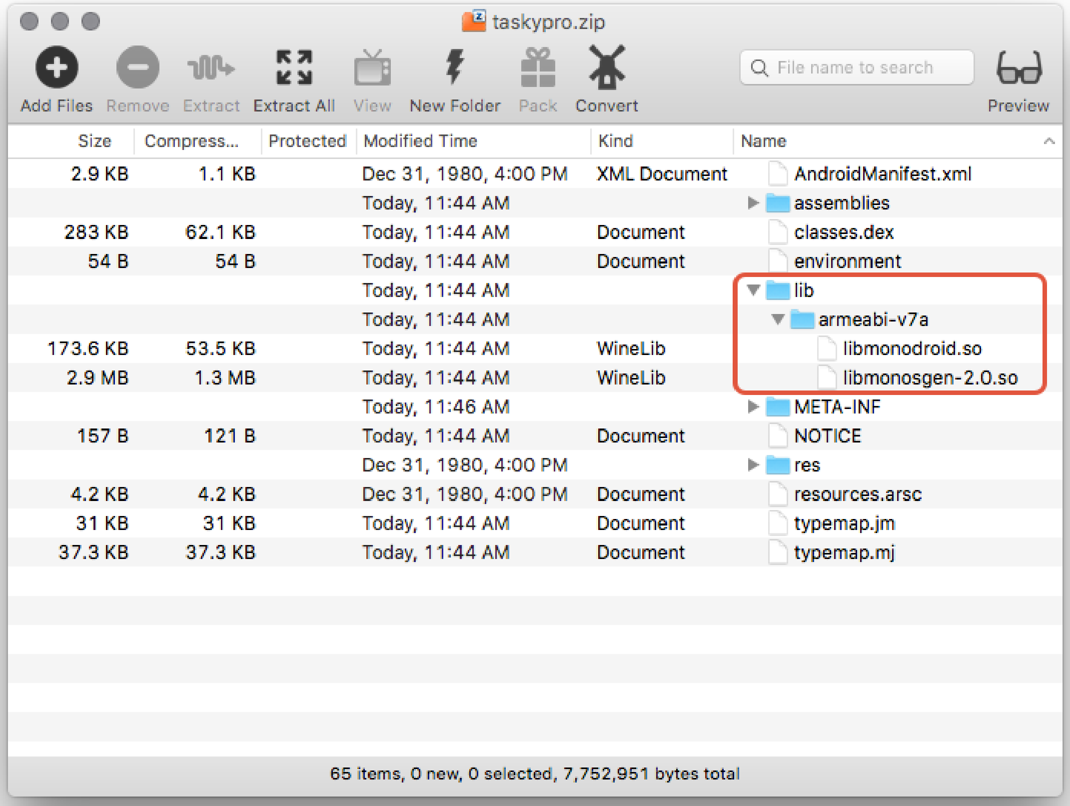
This guide applies to authors of custom Android ROM images. It will not explain how to create a custom ROM.

This guide assumes familiarity with packaging a release APK for a Xamarin.Android and an understanding of CPU Architectures for Android applications.

Install a Xamarin.Android App as a System App

The following steps describe how to install a Xamarin.Android app as a system app.

1. **Package a release APK of the Xamarin.Android app** – This is described in more detail by the Publishing an Application guide.
2. **Extract shared libraries from the APK** – Using any ZIP utility program, open up the APK file and examine the contents of the **/lib/** folder. This folder will have a subdirectory for each *application binary interface* (ABI)that is supported by the application; the contents of this folder will include all of the shared libraries that are required by the app on that particular ABI:



In the previous screenshot, there is only one supported ABI (**armeabi-v7a**) holding the two **.so** files that are required by the app. Note that it is only necessary to extract the ABI files that are appropriate for the device or the target architecture of the device ROM, i.e. do not copy **.so** files from the **x86** folder to an **armeabi-v7a** device or ROM.

3. **Copy .so files to /system/lib** – Copy the **.so** files that were extracted from the APK in the previous step to the **/system/lib/** folder on the custom ROM.

4. **Copy the APK file to /system/app** – The final step is to copy the APK file to the **/system/app** folder on the ROM.

Summary

This guide discussed the difference between a *system app* and a *user app*, and explained how to install a Xamarin.Android application as a system app.

Related Links

- Publishing an Application
- CPU Architectures
- BLUETOOTH_PRIVILEGED
- ABI Management

Advanced Concepts and Internals

10/28/2019 • 2 minutes to read • Edit Online

This section contains topics that explain the architecture, API design, and limitations of Xamarin.Android. In addition, it includes topics that explain its garbage collection implementation and the assemblies that are available in Xamarin.Android. Because Xamarin.Android is open-source, it is also possible to understand the inner workings of Xamarin.Android by examining its source code.

Architecture

This article explains the underlying architecture behind a Xamarin.Android application. It explains how Xamarin.Android applications run inside a Mono execution environment alongside with the Android runtime Virtual Machine and explains such key concepts as Android Callable Wrappers and Managed Callable Wrappers.

API Design

In addition to the core Base Class Libraries that are part of Mono, Xamarin.Android ships with bindings for various Android APIs to allow developers to create native Android applications with Mono.

At the core of Xamarin.Android there is an interop engine that bridges the C# world with the Java world and provides developers with access to the Java APIs from C# or other .NET languages.

Assemblies

Xamarin.Android ships with several assemblies. Just as Silverlight is an extended subset of the desktop .NET assemblies, Xamarin.Android is also an extended subset of several Silverlight and desktop .NET assemblies.

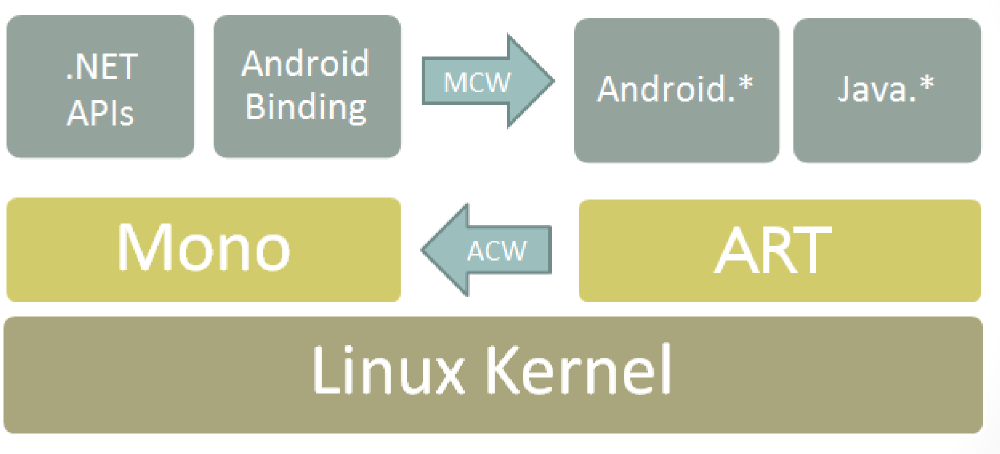
Architecture

7/8/2021 • 8 minutes to read • Edit Online

Xamarin.Android applications run within the Mono execution environment. This execution environment runs side-by-side with the Android Runtime (ART) virtual machine. Both runtime environments run on top of the Linux kernel and expose various APIs to the user code that allows developers to access the underlying system. The Mono runtime is written in the C language.

You can be using the System, System.IO, System.Net and the rest of the .NET class libraries to access the underlying Linux operating system facilities.

On Android, most of the system facilities like Audio, Graphics, OpenGL and Telephony are not available directly to native applications, they are only exposed through the Android Runtime Java APIs residing in one of the Java.* namespaces or the Android.* namespaces. The architecture is roughly like this:



Xamarin.Android developers access the various features in the operating system either by calling into .NET APIs that they know (for low-level access) or using the classes exposed in the Android namespaces which provides a bridge to the Java APIs that are exposed by the Android Runtime.

For more information on how the Android classes communicate with the Android Runtime classes see the API Design document.

Application Packages

Android application packages are ZIP containers with a *.apk* file extension. Xamarin.Android application packages have the same structure and layout as normal Android packages, with the following additions:

- The application assemblies (containing IL) are *stored* uncompressed within the *assemblies* folder. During process startup in Release builds the *.apk* is *mmap()* ed into the process and the assemblies are loaded from memory. This permits faster app startup, as assemblies do not need to be extracted prior to execution.
- *Note:* Assembly location information such as Assembly.Location and Assembly.CodeBase *cannot be relied upon* in Release builds. They do not exist as distinct filesystem entries, and they have no usable location.
- Native libraries containing the Mono runtime are present within the *.apk* . A Xamarin.Android application must contain native libraries for the desired/targeted Android architectures, e.g. *armeabi* , *armeabi-v7a* , *x86* . Xamarin.Android applications cannot run on a platform unless it contains the appropriate runtime libraries.

Xamarin.Android applications also contain *Android Callable Wrappers* to allow Android to call into managed code.

Android Callable Wrappers

- **Android callable wrappers** are a JNI bridge which are used any time the Android runtime needs to invoke managed code. Android callable wrappers are how virtual methods can be overridden and Java interfaces can be implemented. See the Java Integration Overview doc for more.

Managed Callable Wrappers

Managed callable wrappers are a JNI bridge which are used any time managed code needs to invoke Android code and provide support for overriding virtual methods and implementing Java interfaces. The entire Android.* and related namespaces are managed callable wrappers generated via .jar binding. Managed callable wrappers are responsible for converting between managed and Android types and invoking the underlying Android platform methods via JNI.

Each created managed callable wrapper holds a Java global reference, which is accessible through the Android.Runtime.IJavaObject.Handle property. Global references are used to provide the mapping between Java instances and managed instances. Global references are a limited resource: emulators allow only 2000 global references to exist at a time, while most hardware allows over 52,000 global references to exist at a time.

To track when global references are created and destroyed, you can set the debug.mono.log system property to contain gref.

Global references can be explicitly freed by calling Java.Lang.Object.Dispose() on the managed callable wrapper. This will remove the mapping between the Java instance and the managed instance and allow the Java instance to be collected. If the Java instance is re-accessed from managed code, a new managed callable wrapper will be created for it.

Care must be exercised when disposing of Managed Callable Wrappers if the instance can be inadvertently shared between threads, as disposing the instance will impact references from any other threads. For maximum safety, only `Dispose()` of instances which have been allocated via `new` *or* from methods which you *know* always allocate new instances and not cached instances which may cause accidental instance sharing between threads.

Managed Callable Wrapper Subclasses

Managed callable wrapper subclasses are where all the "interesting" application-specific logic may live. These include custom Android.App.Activity subclasses (such as the Activity1 type in the default project template). (Specifically, these are any *Java.Lang.Object* subclasses which do *not* contain a RegisterAttribute custom attribute or RegisterAttribute.DoNotGenerateAcw is *false*, which is the default.)

Like managed callable wrappers, managed callable wrapper subclasses also contain a global reference, accessible through the Java.Lang.Object.Handle property. Just as with managed callable wrappers, global references can be explicitly freed by calling Java.Lang.Object.Dispose(). Unlike managed callable wrappers, *great care* should be taken before disposing of such instances, as *Dispose()*-ing of the instance will break the mapping between the Java instance (an instance of an Android Callable Wrapper) and the managed instance.

Java Activation

When an Android Callable Wrapper (ACW) is created from Java, the ACW constructor will cause the corresponding C# constructor to be invoked. For example, the ACW for *MainActivity* will contain a default constructor which will invoke *MainActivity*'s default constructor. (This is done through the *TypeManager.Activate()* call within the ACW constructors.)

There is one other constructor signature of consequence: the *(IntPtr, JniHandleOwnership)* constructor. The *(IntPtr, JniHandleOwnership)* constructor is invoked whenever a Java object is exposed to managed code and a Managed Callable Wrapper needs to be constructed to manage the JNI handle. This is usually done automatically.

There are two scenarios in which the *(IntPtr, JniHandleOwnership)* constructor must be manually provided on a Managed Callable Wrapper subclass:

1. Android.App.Application is subclassed. *Application* is special; the default *Applicaton* constructor will *never* be invoked, and the (IntPtr, JniHandleOwnership) constructor must instead be provided.

2. Virtual method invocation from a base class constructor.

Note that (2) is a leaky abstraction. In Java, as in C#, calls to virtual methods from a constructor always invoke the most derived method implementation. For example, the TextView(Context, AttributeSet, int) constructor invokes the virtual method TextView.getDefaultMovementMethod(), which is bound as the TextView.DefaultMovementMethod property. Thus, if a type LogTextBox were to (1) subclass TextView, (2) override TextView.DefaultMovementMethod, and (3) activate an instance of that class via XML, the overridden *DefaultMovementMethod* property would be invoked before the ACW constructor had a chance to execute, and it would occur before the C# constructor had a chance to execute.

This is supported by instantiating an instance LogTextBox through the LogTextView(IntPtr, JniHandleOwnership) constructor when the ACW LogTextBox instance first enters managed code, and then invoking the LogTextBox(Context, IAttributeSet, int) constructor *on the same instance* when the ACW constructor executes.

Order of events:

1. Layout XML is loaded into a ContentView.

2. Android instantiates the Layout object graph, and instantiates an instance of *monodroid.apidemo.LogTextBox* , the ACW for *LogTextBox* .

3. The *monodroid.apidemo.LogTextBox* constructor executes the android.widget.TextView constructor.

4. The *TextView* constructor invokes *monodroid.apidemo.LogTextBox.getDefaultMovementMethod()* .

5. *monodroid.apidemo.LogTextBox.getDefaultMovementMethod()* invokes *LogTextBox.n_getDefaultMovementMethod()* , which invokes *TextView.n_GetDefaultMovementMethod()* , which invokes Java.Lang.Object.GetObject<TextView> (handle, JniHandleOwnership.DoNotTransfer) .

6. *Java.Lang.Object.GetObject<TextView>()* checks to see if there is already a corresponding C# instance for *handle* . If there is, it is returned. In this scenario, there isn't, so *Object.GetObject<T>()* must create one.

7. *Object.GetObject<T>()* looks for the *LogTextBox(IntPtr, JniHandleOwneship)* constructor, invokes it, creates a mapping between *handle* and the created instance, and returns the created instance.

8. *TextView.n_GetDefaultMovementMethod()* invokes the *LogTextBox.DefaultMovementMethod* property getter.

9. Control returns to the *android.widget.TextView* constructor, which finishes execution.

10. The *monodroid.apidemo.LogTextBox* constructor executes, invoking *TypeManager.Activate()* .

11. The *LogTextBox(Context, IAttributeSet, int)* constructor executes *on the same instance created in (7)* .

12. If the (IntPtr, JniHandleOwnership) constructor cannot be found, then a System.MissingMethodException] (xref:System.MissingMethodException) will be thrown.

Premature Dispose() Calls

There is a mapping between a JNI handle and the corresponding C# instance. Java.Lang.Object.Dispose() breaks this mapping. If a JNI handle enters managed code after the mapping has been broken, it looks like Java Activation, and the *(IntPtr, JniHandleOwnership)* constructor will be checked for and invoked. If the constructor doesn't exist, then an exception will be thrown.

For example, given the following Managed Callable Wraper subclass:

```
class ManagedValue : Java.Lang.Object {

    public string Value {get; private set;}

    public ManagedValue (string value)
    {
        Value = value;
    }

    public override string ToString ()
    {
        return string.Format ("[Managed: Value={0}]", Value);
    }
}
```

If we create an instance, Dispose() of it, and cause the Managed Callable Wrapper to be re-created:

```
var list = new JavaList<IJavaObject>();
list.Add (new ManagedValue ("value"));
list [0].Dispose ();
Console.WriteLine (list [0].ToString ());
```

The program will die:

```
E/mono    ( 2906): Unhandled Exception: System.NotSupportedException: Unable to activate instance of type
Scratch.PrematureDispose.ManagedValue from native handle 4051c8c8 --->
System.MissingMethodException: No constructor found for
Scratch.PrematureDispose.ManagedValue::.ctor(System.IntPtr, Android.Runtime.JniHandleOwnership)
E/mono    ( 2906):   at Java.Interop.TypeManager.CreateProxy (System.Type type, IntPtr handle,
JniHandleOwnership transfer) [0x00000] in <filename unknown>:0
E/mono    ( 2906):   at Java.Interop.TypeManager.CreateInstance (IntPtr handle, JniHandleOwnership transfer,
System.Type targetType) [0x00000] in <filename unknown>:0
E/mono    ( 2906):   --- End of inner exception stack trace ---
E/mono    ( 2906):   at Java.Interop.TypeManager.CreateInstance (IntPtr handle, JniHandleOwnership transfer,
System.Type targetType) [0x00000] in <filename unknown>:0
E/mono    ( 2906):   at Java.Lang.Object.GetObject (IntPtr handle, JniHandleOwnership transfer, System.Type
type) [0x00000] in <filename unknown>:0
E/mono    ( 2906):   at Java.Lang.Object._GetObject[IJavaObject] (IntPtr handle, JniHandleOwnership
transfer) [0x00000
```

If the subclass does contain an *(IntPtr, JniHandleOwnership)* constructor, then a *new* instance of the type will be created. As a result, the instance will appear to "lose" all instance data, as it's a new instance. (Note that the Value is null.)

```
I/mono-stdout( 2993): [Managed: Value=]
```

Only *Dispose()* of managed callable wrapper subclasses when you know that the Java object will not be used anymore, or the subclass contains no instance data and a *(IntPtr, JniHandleOwnership)* constructor has been provided.

Application Startup

When an activity, service, etc. is launched, Android will first check to see if there is already a process running to host the activity/service/etc. If no such process exists, then a new process will be created, the AndroidManifest.xml is read, and the type specified in the /manifest/application/@android:name attribute is loaded and instantiated. Next, all types specified by the /manifest/application/provider/@android:name attribute values are instantiated and have their ContentProvider.attachInfo%28) method invoked. Xamarin.Android hooks into this by adding a *mono.MonoRuntimeProvider ContentProvider* to AndroidManifest.xml during the build process. The *mono.MonoRuntimeProvider.attachInfo()* method is responsible for loading the Mono runtime into the process. Any attempts to use Mono prior to this point will fail. (*Note*: This is why types which subclass Android.App.Application need to provide an (IntPtr, JniHandleOwnership) constructor, as the Application instance is created before Mono can be initialized.)

Once process initialization has completed, `AndroidManifest.xml` is consulted to find the class name of the activity/service/etc. to launch. For example, the /manifest/application/activity/@android:name attribute is used to determine the name of an Activity to load. For Activities, this type must inherit android.app.Activity. The specified type is loaded via Class.forName() (which requires that the type be a Java type, hence the Android Callable Wrappers), then instantiated. Creation of an Android Callable Wrapper instance will trigger creation of an instance of the corresponding C# type. Android will then invoke Activity.onCreate(Bundle) , which will cause the corresponding Activity.OnCreate(Bundle) to be invoked, and you're off to the races.

Available Assemblies

7/12/2021 • 2 minutes to read • Edit Online

Xamarin.iOS, Xamarin.Android, and Xamarin.Mac all ship with over a dozen assemblies. Just as Silverlight is an extended subset of the desktop .NET assemblies, Xamarin platforms is also an extended subset of several Silverlight and desktop .NET assemblies.

Xamarin platforms are not ABI compatible with existing assemblies compiled for a different profile. You must recompile your source code to generate assemblies targeting the correct profile (just as you need to recompile source code to target Silverlight and .NET 3.5 separately).

Xamarin.Mac applications can be compiled in three modes: one that uses Xamarin's curated Mobile Profile, the Xamarin.Mac .NET 4.5 Framework which allows you target existing full desktop assemblies, and an unsupported one that uses the .NET API found in a system Mono installation. For more information, please see our Target Frameworks documentation.

.NET Standard Libraries

In addition to the iOS, Android, and Mac bindings, Xamarin projects can consume .NET Standard libraries.

Portable Class Libraries

Xamarin projects can also consume .NET Portable Class Libraries, although this technology is being deprecated in favor of .NET Standard.

Supported Assemblies

These are the assemblies available in the **Reference Manager > Assemblies > Framework** (Visual Studio 2017) and **Edit References > Packages** (Visual Studio for Mac), and their compatibility with Xamarin platforms.

ASSEMBLY	API COMPATIBILITY	XAMARIN IOS	XAMARIN ANDROID	XAMARIN MAC
FSharp.Core.dll		✓	✓	✓
I18N.dll	Includes CJK, MidEast, Other, Rare, West	✓	✓	✓
Microsoft.CSharp.dll		✓	✓	✓
Mono.CSharp.dll		✓	✓	✓
Mono.Data.Sqlite.dll	ADO.NET provider for SQLite; see limitations.	✓	✓	✓

ASSEMBLY	API COMPATIBILITY	XAMARIN IOS	XAMARIN ANDROID	XAMARIN MAC
Mono.Data.Tds.dll	TDS Protocol support; used for System.Data.SqlClient support within System.Data.	✓	✓	✓
Mono.Dynamic.Interpreter.dll		✓		
Mono.Security.dll	Cryptographic APIs.	✓	✓	✓
monotouch.dll	This assembly contains the C# binding to the CocoaTouch API. This is only available within Classic iOS Projects.	✓		
MonoTouch.Dialog-1.dll		✓		
MonoTouch.NUnitLite.dll		✓		
mscorlib.dll	Silverlight	✓	✓	✓
OpenTK-1.0.dll	The OpenGL/OpenAL object oriented APIs, extended to provide iPhone device support.	✓	✓	✓

ASSEMBLY	API COMPATIBILITY	XAMARIN IOS	XAMARIN ANDROID	XAMARIN MAC
System.dll	Silverlight, plus types from the following namespaces: System.Collections.Specialized System.ComponentModel System.ComponentModel.Design System.Diagnostics System.IO System.IO.Compression System.IO.Compression.FileSystem System.Net System.Net.Cache System.Net.Mail System.Net.Mime System.Net.NetworkInformation System.Net.Security System.Net.Sockets System.Runtime.InteropServices System.Runtime.Versioning System.Security.AccessControl System.Security.Authentication System.Security.Cryptography System.Security.Permissions System.Threading System.Timers	✓	✓	✓
System.ComponentModel.Composition.dll		✓	✓	✓
System.ComponentModel.DataAnnotations.dll		✓	✓	✓
System.Core.dll	Silverlight	✓	✓	✓
System.Data.dll	.NET 3.5 , with some functionality removed.	✓	✓	✓
System.Data.Services.Client.dll	Full oData client.	✓	✓	✓
System.IO.Compression		✓	✓	✓

ASSEMBLY	API COMPATIBILITY	XAMARIN IOS	XAMARIN ANDROID	XAMARIN MAC
System.IO.Compression.FileSystem		✓	✓	✓
System.Json.dll	Silverlight	✓	✓	✓
System.Net.Http.dll		✓	✓	✓
System.Numerics.dll		✓	✓	✓
System.Runtime.Serialization.dll	Silverlight	✓	✓	✓
System.ServiceModel.dll	WCF stack as present in Silverlight	✓	✓	✓
System.ServiceModel.Internals.dll		✓	✓	✓
System.ServiceModel.Web.dll	Silverlight, plus types from the following namespaces: System System.ServiceModel.Channels System.ServiceModel.Description System.ServiceModel.Web	✓	✓	✓
System.Transactions.dll	.NET 3.5; part of System.Data support.	✓	✓	✓
System.Web.Services.dll	Basic Web services from the .NET 3.5 profile, with the server features removed.	✓	✓	✓
System.Windows.dll		✓	✓	✓
System.Xml.dll	.NET 3.5	✓	✓	✓
System.Xml.Linq.dll	.NET 3.5	✓	✓	✓
System.Xml.Serialization.dll		✓	✓	✓
Xamarin.iOS.dll	This assembly contains the C# binding to the CocoaTouch API. This is only used in Unified iOS Projects.	✓		

ASSEMBLY	API COMPATIBILITY	XAMARIN IOS	XAMARIN ANDROID	XAMARIN MAC
Java.Interop.dll			✓	
Mono.Android.dll			✓	
Mono.Android.Export.dll			✓	
Mono.Posix.dll			✓	
System.EnterpriseServices.dll			✓	
Xamarin.Android.NUnitLite.dll			✓	
Mono.CompilerServices.SymbolWriter.dll	For compiler writers.			✓
Xamarin.Mac.dll				✓
System.Drawing.dll	System.Drawing is not supported in the Unified API for the Xamarin.Mac, .NET 4.5, or Mobile frameworks. System.Drawing support can be added to iOS and macOS using the sysdrawing-coregraphics library	✓		✓

Xamarin.Android API Design Principles

11/2/2020 • 11 minutes to read • Edit Online

In addition to the core Base Class Libraries that are part of Mono, Xamarin.Android ships with bindings for various Android APIs to allow developers to create native Android applications with Mono.

At the core of Xamarin.Android there is an interop engine that bridges the C# world with the Java world and provides developers with access to the Java APIs from C# or other .NET languages.

Design Principles

These are some of our design principles for the Xamarin.Android binding

- Conform to the .NET Framework Design Guidelines.
- Allow developers to subclass Java classes.
- Subclass should work with C# standard constructs.
- Derive from an existing class.
- Call base constructor to chain.
- Overriding methods should be done with C#'s override system.
- Make common Java tasks easy, and hard Java tasks possible.
- Expose JavaBean properties as C# properties.
- Expose a strongly typed API:
  - Increase type-safety.
  - Minimize runtime errors.
  - Get IDE intellisense on return types.
  - Allows for IDE popup documentation.
- Encourage in-IDE exploration of the APIs:
  - Utilize Framework Alternatives to Minimize Java Classlib exposure.
  - Expose C# delegates (lambdas, anonymous methods and System.Delegate) instead of single-method interfaces when appropriate and applicable.
  - Provide a mechanism to call arbitrary Java libraries (Android.Runtime.JNIEnv).

Assemblies

Xamarin.Android includes a number of assemblies that constitute the *MonoMobile Profile*. The Assemblies page has more information.

The bindings to the Android platform are contained in the `Mono.Android.dll` assembly. This assembly contains the entire binding for consuming Android APIs and communicating with the Android runtime VM.

Binding Design

Collections

The Android APIs utilize the java.util collections extensively to provide lists, sets, and maps. We expose these elements using the System.Collections.Generic interfaces in our binding. The fundamental mappings are:

- java.util.Set<E> maps to system type ICollection<T>, helper class Android.Runtime.JavaSet<T>.
- java.util.List<E> maps to system type IList<T>, helper class Android.Runtime.JavaList<T>.
- java.util.Map<K,V> maps to system type IDictionary<TKey,TValue>, helper class Android.Runtime.JavaDictionary<K,V>.
- java.util.Collection<E> maps to system type ICollection<T>, helper class Android.Runtime.JavaCollection<T>.

We have provided helper classes to facilitate faster copyless marshaling of these types. When possible, we recommend using these provided collections instead of the framework provided implementation, like `List<T>` or `Dictionary<TKey, TValue>`. The Android.Runtime implementations utilize a native Java collection internally and therefore do not require copying to and from a native collection when passing to an Android API member.

You can pass any interface implementation to an Android method accepting that interface, e.g. pass a `List<int>` to the ArrayAdapter<int>(Context, int, IList<int>) constructor. *However*, for all implementations *except* for the Android.Runtime implementations, this involves *copying* the list from the Mono VM into the Android runtime VM. If the list is later changed within the Android runtime (e.g. by invoking the ArrayAdapter<T>.Add(T) method), those changes *will not* be visible in managed code. If a `JavaList<int>` were used, those changes would be visible.

Rephrased, collections interface implementations that are *not* one of the above listed **Helper Class**es only marshal [In]:

```
// This fails:
var badSource  = new List<int> { 1, 2, 3 };
var badAdapter = new ArrayAdapter<int>(context, textViewResourceId, badSource);
badAdapter.Add (4);
if (badSource.Count != 4) // true
    throw new InvalidOperationException ("this is thrown");

// this works:
var goodSource  = new JavaList<int> { 1, 2, 3 };
var goodAdapter = new ArrayAdapter<int> (context, textViewResourceId, goodSource);
goodAdapter.Add (4);
if (goodSource.Count != 4) // false
    throw new InvalidOperationException ("should not be reached.");
```

Properties

Java methods are transformed into properties, when appropriate:

- The Java method pair `T getFoo()` and `void setFoo(T)` are transformed into the `Foo` property. Example: Activity.Intent.
- The Java method `getFoo()` is transformed into the read-only Foo property. Example: Context.PackageName.
- Set-only properties are not generated.
- Properties are *not* generated if the property type would be an array.

Events and Listeners

The Android APIs are built on top of Java and its components follow the Java pattern for hooking up event listeners. This pattern tends to be cumbersome as it requires the user to create an anonymous class and declare the methods to override, for example, this is how things would be done in Android with Java:

```
final android.widget.Button button = new android.widget.Button(context);

button.setText(this.count + " clicks!");
button.setOnClickListener (new View.OnClickListener() {
    public void onClick (View v) {
        button.setText(++this.count + " clicks!");
    }
});
```

The equivalent code in C# using events would be:

```
var button = new Android.Widget.Button (context) {
    Text = string.Format ("{0} clicks!", this.count),
};
button.Click += (sender, e) => {
    button.Text = string.Format ("{0} clicks!", ++this.count);
};
```

Note that both of the above mechanisms are available with Xamarin.Android. You can implement a listener interface and attach it with View.SetOnClickListener, or you can attach a delegate created via any of the usual C# paradigms to the Click event.

When the listener callback method has a void return, we create API elements based on an EventHandler<TEventArgs> delegate. We generate an event like the above example for these listener types. However, if the listener callback returns a non-void and non- **boolean** value, events and EventHandlers are not used. We instead generate a specific delegate for the signature of the callback and add properties instead of events. The reason is to deal with delegate invocation order and return handling. This approach mirrors what is done with the Xamarin.iOS API.

C# events or properties are only automatically generated if the Android event-registration method:

1. Has a `set` prefix, e.g. *set*OnClickListener.

2. Has a `void` return type.

3. Accepts only one parameter, the parameter type is an interface, the interface has only one method, and the interface name ends in `Listener` , e.g. View.OnClick *Listener*.

Furthermore, if the Listener interface method has a return type of **boolean** instead of **void**, then the generated *EventArgs* subclass will contain a *Handled* property. The value of the *Handled* property is used as the return value for the *Listener* method, and it defaults to `true`.

For example, the Android View.setOnKeyListener() method accepts the View.OnKeyListener interface, and the View.OnKeyListener.onKey(View, int, KeyEvent) method has a boolean return type. Xamarin.Android generates a corresponding View.KeyPress event, which is an EventHandler<View.KeyEventArgs>. The *KeyEventArgs* class in turn has a View.KeyEventArgs.Handled property, which is used as the return value for the *View.OnKeyListener.onKey()* method.

We intend to add overloads for other methods and ctors to expose the delegate-based connection. Also, listeners with multiple callbacks require some additional inspection to determine if implementing individual callbacks is reasonable, so we are converting these as they are identified. If there is no corresponding event, listeners must be used in C#, but please bring any that you think could have delegate usage to our attention. We have also done some conversions of interfaces without the "Listener" suffix when it was clear they would benefit from a delegate alternative.

All of the listeners interfaces implement the `Android.Runtime.IJavaObject` interface, because of the implementation details of the binding, so listener classes must implement this interface. This can be done by implementing the listener interface on a subclass of Java.Lang.Object or any other wrapped Java object, such as an Android activity.

Runnables

Java utilizes the java.lang.Runnable interface to provide a delegation mechanism. The java.lang.Thread class is a notable consumer of this interface. Android has employed the interface in the API as well. Activity.runOnUiThread() and View.post() are notable examples.

The `Runnable` interface contains a single void method, run(). It therefore lends itself to binding in C# as a System.Action delegate. We have provided overloads in the binding which accept an `Action` parameter for all API members which consume a `Runnable` in the native API, e.g. Activity.RunOnUiThread() and View.Post().

We left the IRunnable overloads in place instead of replacing them since several types implement the interface and can therefore be passed as runnables directly.

Inner Classes

Java has two different types of nested classes: static nested classes and non-static classes.

Java static nested classes are identical to C# nested types.

Non-static nested classes, also called *inner classes*, are significantly different. They contain an implicit reference to an instance of their enclosing type and cannot contain static members (among other differences outside the scope of this overview).

When it comes to binding and C# use, static nested classes are treated as normal nested types. Inner classes, meanwhile, have two significant differences:

1. The implicit reference to the containing type must be provided explicitly as a constructor parameter.
2. When inheriting from an inner class, the inner class *must* be nested within a type that inherits from the containing type of the base inner class, and the derived type must provide a constructor of the same type as the C# containing type.

For example, consider the Android.Service.Wallpaper.WallpaperService.Engine inner class. Since it's an inner class, the WallpaperService.Engine() constructor takes a reference to a WallpaperService instance (compare and contrast to the Java WallpaperService.Engine() constructor, which takes no parameters).

An example derivation of an inner class is CubeWallpaper.CubeEngine:

```
class CubeWallpaper : WallpaperService {
    public override WallpaperService.Engine OnCreateEngine ()
    {
        return new CubeEngine (this);
    }

    class CubeEngine : WallpaperService.Engine {
        public CubeEngine (CubeWallpaper s)
                : base (s)
        {
        }
    }
}
```

Note how `CubeWallpaper.CubeEngine` is nested within `CubeWallpaper`, `CubeWallpaper` inherits from the containing class of `WallpaperService.Engine`, and `CubeWallpaper.CubeEngine` has a constructor which takes the declaring type -- `CubeWallpaper` in this case -- all as specified above.

Interfaces

Java interfaces can contain three sets of members, two of which cause problems from C#:

1. Methods
2. Types
3. Fields

Java interfaces are translated into two types:

1. An (optional) interface containing method declarations. This interface has the same name as the Java interface, *except* it also has an ' *I* ' prefix.
2. An (optional) static class containing any fields declared within the Java interface.

Nested types are "relocated" to be siblings of the enclosing interface instead of nested types, with the enclosing interface name as a prefix.

For example, consider the android.os.Parcelable interface. The *Parcelable* interface contains methods, nested types, and constants. The *Parcelable* interface methods are placed into the Android.OS.IParcelable interface. The *Parcelable* interface constants are placed into the Android.OS.ParcelableConsts type. The nested android.os.Parcelable.ClassLoaderCreator<T> and android.os.Parcelable.Creator<T> types are currently not bound due to limitations in our generics support; if they were supported, they would be present as the *Android.OS.IParcelableClassLoaderCreator* and *Android.OS.IParcelableCreator* interfaces. For example, the nested android.os.IBinder.DeathRecipient interface is bound as the Android.OS.IBinderDeathRecipient interface.

> **NOTE**
>
> Beginning with Xamarin.Android 1.9, Java interface constants are *duplicated* in an effort to simplify porting Java code. This helps to improve porting Java code that relies on android provider interface constants.

In addition to the above types, there are four further changes:

1. A type with the same name as the Java interface is generated to contain constants.
2. Types containing interface constants also contain all constants that come from implemented Java interfaces.
3. All classes that implement a Java interface containing constants get a new nested InterfaceConsts type which contains constants from all implemented interfaces.
4. The *Consts* type is now obsolete.

For the *android.os.Parcelable* interface, this means that there will now be an *Android.OS.Parcelable* type to contain the constants. For example, the Parcelable.CONTENTS_FILE_DESCRIPTOR constant will be bound as the *Parcelable.ContentsFileDescriptor* constant, instead of as the *ParcelableConsts.ContentsFileDescriptor* constant.

For interfaces containing constants which implement other interfaces containing yet more constants, the union of all constants is now generated. For example, the android.provider.MediaStore.Video.VideoColumns interface implements the android.provider.MediaStore.MediaColumns interface. However, prior to 1.9, the Android.Provider.MediaStore.Video.VideoColumnsConsts type has no way of accessing the constants declared on Android.Provider.MediaStore.MediaColumnsConsts. As a result, the Java expression *MediaStore.Video.VideoColumns.TITLE* needs to be bound to the C# expression *MediaStore.Video.MediaColumnsConsts.Title* which is hard to discover without reading lots of Java documentation. In 1.9, the equivalent C# expression will be *MediaStore.Video.VideoColumns.Title*.

Furthermore, consider the android.os.Bundle type, which implements the Java *Parcelable* interface. Since it implements the interface, all constants on that interface are accessible "through" the Bundle type, e.g.

Bundle.CONTENTS_FILE_DESCRIPTOR is a perfectly valid Java expression. Previously, to port this expression to C# you would need to look at all the interfaces which are implemented to see from which type the *CONTENTS_FILE_DESCRIPTOR* came from. Starting in Xamarin.Android 1.9, classes implementing Java interfaces which contain constants will have a nested *InterfaceConsts* type, which will contain all the inherited interface constants. This will allow translating *Bundle.CONTENTS_FILE_DESCRIPTOR* to *Bundle.InterfaceConsts.ContentsFileDescriptor*.

Finally, types with a *Consts* suffix such as *Android.OS.ParcelableConsts* are now Obsolete, other than the newly introduced InterfaceConsts nested types. They will be removed in Xamarin.Android 3.0.

Resources

Images, layout descriptions, binary blobs and string dictionaries can be included in your application as resource files. Various Android APIs are designed to operate on the resource IDs instead of dealing with images, strings or binary blobs directly.

For example, a sample Android app that contains a user interface layout (`main.axml`), an internationalization table string (`strings.xml`) and some icons (`drawable-*/icon.png`) would keep its resources in the "Resources" directory of the application:

```
Resources/
    drawable-hdpi/
        icon.png

    drawable-ldpi/
        icon.png

    drawable-mdpi/
        icon.png

    layout/
        main.axml

    values/
        strings.xml
```

The native Android APIs do not operate directly with filenames, but instead operate on resource IDs. When you compile an Android application that uses resources, the build system will package the resources for distribution and generate a class called `Resource` that contains the tokens for each one of the resources included. For example, for the above Resources layout, this is what the R class would expose:

```
public class Resource {
    public class Drawable {
        public const int icon = 0x123;
    }

    public class Layout {
        public const int main = 0x456;
    }

    public class String {
        public const int first_string = 0xabc;
        public const int second_string = 0xbcd;
    }
}
```

You would then use `Resource.Drawable.icon` to reference the `drawable/icon.png` file, or `Resource.Layout.main` to reference the `layout/main.xml` file, or `Resource.String.first_string` to reference the first string in the dictionary file `values/strings.xml`.

Constants and Enumerations

The native Android APIs have many methods that take or return an int that must be mapped to a constant field to determine what the int means. To use these methods, the user is required to consult the documentation to see which constants are appropriate values, which is less than ideal.

For example, consider Activity.requestWindowFeature(int featureID).

In these cases, we endeavor to group related constants together into a .NET enumeration, and remap the method to take the enumeration instead. By doing this, we are able to offer IntelliSense selection of the potential values.

The above example becomes: Activity.RequestWindowFeature(WindowFeatures featureId).

Note that this is a very manual process to figure out which constants belong together, and which APIs consume these constants. Please file bugs for any constants used in the API that would be better expressed as an enumeration.

Garbage Collection

11/2/2020 • 15 minutes to read • Edit Online

Xamarin.Android uses Mono's Simple Generational garbage collector. This is a mark-and-sweep garbage collector with two generations and a *large object space*, with two kinds of collections:

- Minor collections (collects Gen0 heap)
- Major collections (collects Gen1 and large object space heaps).

> **NOTE**
>
> In the absence of an explicit collection via GC.Collect() collections are *on demand*, based upon heap allocations. *This is not a reference counting system*; objects *will not be collected as soon as there are no outstanding references*, or when a scope has exited. The GC will run when the minor heap has run out of memory for new allocations. If there are no allocations, it will not run.

Minor collections are cheap and frequent, and are used to collect recently allocated and dead objects. Minor collections are performed after every few MB of allocated objects. Minor collections may be manually performed by calling GC.Collect (0)

Major collections are expensive and less frequent, and are used to reclaim all dead objects. Major collections are performed once memory is exhausted for the current heap size (before resizing the heap). Major collections may be manually performed by calling GC.Collect () or by calling GC.Collect (int) with the argument GC.MaxGeneration.

Cross-VM Object Collections

There are three categories of object types.

- **Managed objects**: types which do *not* inherit from Java.Lang.Object , e.g. System.String. These are collected normally by the GC.
- **Java objects**: Java types which are present within the Android runtime VM but not exposed to the Mono VM. These are boring, and won't be discussed further. These are collected normally by the Android runtime VM.
- **Peer objects**: types which implement IJavaObject , e.g. all Java.Lang.Object and Java.Lang.Throwable subclasses. Instances of these types have two "halfs" a *managed peer* and a *native peer*. The managed peer is an instance of the C# class. The native peer is an instance of a Java class within the Android runtime VM, and the C# IJavaObject.Handle property contains a JNI global reference to the native peer.

There are two types of native peers:

- **Framework peers** : "Normal" Java types which know nothing of Xamarin.Android, e.g. android.content.Context.
- **User peers** : Android Callable Wrappers which are generated at build time for each Java.Lang.Object subclass present within the application.

As there are two VMs within a Xamarin.Android process, there are two types of garbage collections:

- Android runtime collections
- Mono collections

Android runtime collections operate normally, but with a caveat: a JNI global reference is treated as a GC root. Consequently, if there is a JNI global reference holding onto an Android runtime VM object, the object *cannot* be collected, even if it's otherwise eligible for collection.

Mono collections are where the fun happens. Managed objects are collected normally. Peer objects are collected by performing the following process:

1. All Peer objects eligible for Mono collection have their JNI global reference replaced with a JNI weak global reference.

2. An Android runtime VM GC is invoked. Any Native peer instance may be collected.

3. The JNI weak global references created in (1) are checked. If the weak reference has been collected, then the Peer object is collected. If the weak reference has *not* been collected, then the weak reference is replaced with a JNI global reference and the Peer object is not collected. Note: on API 14+, this means that the value returned from `IJavaObject.Handle` may change after a GC.

The end result of all this is that an instance of a Peer object will live as long as it is referenced by either managed code (e.g. stored in a `static` variable) or referenced by Java code. Furthermore, the lifetime of Native peers will be extended beyond what they would otherwise live, as the Native peer won't be collectible until both the Native peer and the Managed peer are collectible.

Object Cycles

Peer objects are logically present within both the Android runtime and Mono VM's. For example, an Android.App.Activity managed peer instance will have a corresponding android.app.Activity framework peer Java instance. All objects that inherit from Java.Lang.Object can be expected to have representations within both VMs.

All objects that have representation in both VMs will have lifetimes which are extended compared to objects which are present only within a single VM (such as a `System.Collections.Generic.List<int>`). Calling GC.Collect won't necessarily collect these objects, as the Xamarin.Android GC needs to ensure that the object isn't referenced by either VM before collecting it.

To shorten object lifetime, Java.Lang.Object.Dispose() should be invoked. This will manually "sever" the connection on the object between the two VMs by freeing the global reference, thus allowing the objects to be collected faster.

Automatic Collections

Beginning with Release 4.1.0, Xamarin.Android automatically performs a full GC when a gref threshold is crossed. This threshold is 90% of the known maximum grefs for the platform: 1800 grefs on the emulator (2000 max), and 46800 grefs on hardware (maximum 52000). *Note:* Xamarin.Android only counts the grefs created by Android.Runtime.JNIEnv, and will not know about any other grefs created in the process. This is a heuristic *only*.

When an automatic collection is performed, a message similar to the following will be printed to the debug log:

```
I/monodroid-gc(PID): 46800 outstanding GREFs. Performing a full GC!
```

The occurrence of this is non-deterministic, and may happen at inopportune times (e.g. in the middle of graphics rendering). If you see this message, you may want to perform an explicit collection elsewhere, or you may want to try to reduce the lifetime of peer objects.

GC Bridge Options

Xamarin.Android offers transparent memory management with Android and the Android runtime. It is implemented as an extension to the Mono garbage collector called the *GC Bridge*.

The GC Bridge works during a Mono garbage collection and figures out which peer objects needs their "liveness" verified with the Android runtime heap. The GC Bridge makes this determination by doing the following steps (in order):

1. Induce the mono reference graph of unreachable peer objects into the Java objects they represent.
2. Perform a Java GC.
3. Verify which objects are really dead.

This complicated process is what enables subclasses of `Java.Lang.Object` to freely reference any objects; it removes any restrictions on which Java objects can be bound to C#. Because of this complexity, the bridge process can be very expensive and it can cause noticeable pauses in an application. If the application is experiencing significant pauses, it's worth investigating one of the following three GC Bridge implementations:

- **Tarjan** - A completely new design of the GC Bridge based on Robert Tarjan's algorithm and backwards reference propagation. It has the best performance under our simulated workloads, but it also has the larger share of experimental code.
- **New** - A major overhaul of the original code, fixing two instances of quadratic behavior but keeping the core algorithm (based on Kosaraju's algorithm for finding strongly connected components).
- **Old** - The original implementation (considered the most stable of the three). This is the bridge that an application should use if the `GC_BRIDGE` pauses are acceptable.

The only way to figure out which GC Bridge works best is by experimenting in an application and analyzing the output. There are two ways to collect the data for benchmarking:

- **Enable logging** - Enable logging (as describe in the Configuration section) for each GC Bridge option, then capture and compare the log outputs from each setting. Inspect the `GC` messages for each option; in particular, the `GC_BRIDGE` messages. Pauses up to 150ms for non-interactive applications are tolerable, but pauses above 60ms for very interactive applications (such as games) are a problem.
- **Enable bridge accounting** - Bridge accounting will display the average cost of the objects pointed by each object involved in the bridge process. Sorting this information by size will provide hints as to what is holding the largest amount of extra objects.

The default setting is **Tarjan**. If you find a regression, you may find it necessary to set this option to **Old**. Also, you may choose to use the more stable **Old** option if **Tarjan** does not produce an improvement in performance.

To specify which `GC_BRIDGE` option an application should use, pass `bridge-implementation=old`, `bridge-implementation=new` or `bridge-implementation=tarjan` to the `MONO_GC_PARAMS` environment variable. This is accomplished by adding a new file to your project with a **Build action** of `AndroidEnvironment`. For example:

```
MONO_GC_PARAMS=bridge-implementation=tarjan
```

For more information, see Configuration.

Helping the GC

There are multiple ways to help the GC to reduce memory use and collection times.

Disposing of Peer instances

The GC has an incomplete view of the process and may not run when memory is low because the GC doesn't know that memory is low.

For example, an instance of a Java.Lang.Object type or derived type is at least 20 bytes in size (subject to change without notice, etc., etc.). Managed Callable Wrappers do not add additional instance members, so when you have a Android.Graphics.Bitmap instance that refers to a 10MB blob of memory, Xamarin.Android's GC won't know that – the GC will see a 20-byte object and will be unable to determine that it's linked to Android runtime-allocated objects that's keeping 10MB of memory alive.

It is frequently necessary to help the GC. Unfortunately, *GC.AddMemoryPressure()* and *GC.RemoveMemoryPressure()* are not supported, so if you *know* that you just freed a large Java-allocated object graph you may need to manually call GC.Collect() to prompt a GC to release the Java-side memory, or you can explicitly dispose of *Java.Lang.Object* subclasses, breaking the mapping between the managed callable wrapper and the Java instance. For example, see Bug 1084.

> **NOTE**
> You must be *extremely* careful when disposing of `Java.Lang.Object` subclass instances.

To minimize the possibility of memory corruption, observe the following guidelines when calling `Dispose()`.

Sharing Between Multiple Threads

If the *Java or managed* instance may be shared between multiple threads, *it should not be* `Dispose()`*d*, **ever**. For example, `Typeface.Create()` may return a *cached instance*. If multiple threads provide the same arguments, they will obtain the *same* instance. Consequently, `Dispose()`ing of the `Typeface` instance from one thread may invalidate other threads, which can result in `ArgumentException`s from `JNIEnv.CallVoidMethod()` (among others) because the instance was disposed from another thread.

Disposing Bound Java Types

If the instance is of a bound Java type, the instance can be disposed of *as long as* the instance won't be reused from managed code *and* the Java instance can't be shared amongst threads (see previous `Typeface.Create()` discussion). (Making this determination may be difficult.) The next time the Java instance enters managed code, a *new* wrapper will be created for it.

This is frequently useful when it comes to Drawables and other resource-heavy instances:

```
using (var d = Drawable.CreateFromPath ("path/to/filename"))
    imageView.SetImageDrawable (d);
```

The above is safe because the Peer that Drawable.CreateFromPath() returns will refer to a Framework peer, *not* a User peer. The `Dispose()` call at the end of the `using` block will break the relationship between the managed Drawable and framework Drawable instances, allowing the Java instance to be collected as soon as the Android runtime needs to. This would *not* be safe if Peer instance referred to a User peer; here we're using "external" information to *know* that the `Drawable` cannot refer to a User peer, and thus the `Dispose()` call is safe.

Disposing Other Types

If the instance refers to a type that isn't a binding of a Java type (such as a custom `Activity`), **DO NOT** call `Dispose()` unless you *know* that no Java code will call overridden methods on that instance. Failure to do so results in `NotSupportedException`s.

For example, if you have a custom click listener:

```
partial class MyClickListener : Java.Lang.Object, View.IOnClickListener {
    // ...
}
```

You *should not* dispose of this instance, as Java will attempt to invoke methods on it in the future:

```
// BAD CODE; DO NOT USE
Button b = FindViewById<Button> (Resource.Id.myButton);
using (var listener = new MyClickListener ())
    b.SetOnClickListener (listener);
```

Using Explicit Checks to Avoid Exceptions

If you've implemented a Java.Lang.Object.Dispose overload method, avoid touching objects that involve JNI. Doing so may create a *double-dispose* situation that makes it possible for your code to (fatally) attempt to access an underlying Java object that has already been garbage-collected. Doing so produces an exception similar to the following:

```
System.ArgumentException: 'jobject' must not be IntPtr.Zero.
Parameter name: jobject
at Android.Runtime.JNIEnv.CallVoidMethod
```

This situation often occurs when the first dispose of an object causes a member to become null, and then a subsequent access attempt on this null member causes an exception to be thrown. Specifically, the object's `Handle` (which links a managed instance to its underlying Java instance) is invalidated on the first dispose, but managed code still attempts to access this underlying Java instance even though it is no longer available (see Managed Callable Wrappers for more about the mapping between Java instances and managed instances).

A good way to prevent this exception is to explicitly verify in your `Dispose` method that the mapping between the managed instance and the underlying Java instance is still valid; that is, check to see if the object's `Handle` is null (`IntPtr.Zero`) before accessing its members. For example, the following `Dispose` method accesses a `childViews` object:

```
class MyClass : Java.Lang.Object, ISomeInterface
{
    protected override void Dispose (bool disposing)
    {
        base.Dispose (disposing);
        for (int i = 0; i < this.childViews.Count; ++i)
        {
            // ...
        }
    }
}
```

If an initial dispose pass causes `childViews` to have an invalid `Handle` , the `for` loop access will throw an `ArgumentException` . By adding an explicit `Handle` null check before the first `childViews` access, the following `Dispose` method prevents the exception from occurring:

```
class MyClass : Java.Lang.Object, ISomeInterface
{
    protected override void Dispose (bool disposing)
    {
        base.Dispose (disposing);

        // Check for a null handle:
        if (this.childViews.Handle == IntPtr.Zero)
            return;

        for (int i = 0; i < this.childViews.Count; ++i)
        {
            // ...
        }
    }
}
```

Reduce Referenced Instances

Whenever an instance of a `Java.Lang.Object` type or subclass is scanned during the GC, the entire *object graph* that the instance refers to must also be scanned. The object graph is the set of object instances that the "root instance" refers to, *plus* everything referenced by what the root instance refers to, recursively.

Consider the following class:

```
class BadActivity : Activity {

    private List<string> strings;

    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);

        strings.Value = new List<string> (
                Enumerable.Range (0, 10000)
                .Select(v => new string ('x', v % 1000)));
    }
}
```

When `BadActivity` is constructed, the object graph will contain 10004 instances (1x `BadActivity`, 1x `strings`, 1x `string[]` held by `strings`, 10000x string instances), *all* of which will need to be scanned whenever the `BadActivity` instance is scanned.

This can have detrimental impacts on your collection times, resulting in increased GC pause times.

You can help the GC by *reducing* the size of object graphs which are rooted by User peer instances. In the above example, this can be done by moving `BadActivity.strings` into a separate class which doesn't inherit from Java.Lang.Object:

```
class HiddenReference<T> {

    static Dictionary<int, T> table = new Dictionary<int, T> ();
    static int idgen = 0;

    int id;

    public HiddenReference ()
    {
        lock (table) {
            id = idgen ++;
        }
    }

    ~HiddenReference ()
    {
        lock (table) {
            table.Remove (id);
        }
    }

    public T Value {
        get { lock (table) { return table [id]; } }
        set { lock (table) { table [id] = value; } }
    }
}

class BetterActivity : Activity {

    HiddenReference<List<string>> strings = new HiddenReference<List<string>>();

    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);

        strings.Value = new List<string> (
                Enumerable.Range (0, 10000)
                .Select(v => new string ('x', v % 1000)));
    }
}
```

Minor Collections

Minor collections may be manually performed by calling GC.Collect(0). Minor collections are cheap (when compared to major collections), but do have a significant fixed cost, so you don't want to trigger them too often, and should have a pause time of a few milliseconds.

If your application has a "duty cycle" in which the same thing is done over and over, it may be advisable to manually perform a minor collection once the duty cycle has ended. Example duty cycles include:

- The rendering cycle of a single game frame.
- The whole interaction with a given app dialog (opening, filling, closing)
- A group of network requests to refresh/sync app data.

Major Collections

Major collections may be manually performed by calling GC.Collect() or `GC.Collect(GC.MaxGeneration)`.

They should be performed rarely, and may have a pause time of a second on an Android-style device when collecting a 512MB heap.

Major collections should only be manually invoked, if ever:

- At the end of lengthy duty cycles and when a long pause won't present a problem to the user.
- Within an overridden Android.App.Activity.OnLowMemory() method.

Diagnostics

To track when global references are created and destroyed, you can set the debug.mono.log system property to contain *gref* and/or *gc*.

Configuration

The Xamarin.Android garbage collector can be configured by setting the `MONO_GC_PARAMS` environment variable. Environment variables may be set with a Build action of AndroidEnvironment.

The `MONO_GC_PARAMS` environment variable is a comma-separated list of the following parameters:

- `nursery-size` = *size* : Sets the size of the nursery. The size is specified in bytes and must be a power of two. The suffixes `k` , `m` and `g` can be used to specify kilo-, mega- and gigabytes, respectively. The nursery is the first generation (of two). A larger nursery will usually speed up the program but will obviously use more memory. The default nursery size 512 kb.
- `soft-heap-limit` = *size* : The target maximum managed memory consumption for the app. When memory use is below the specified value, the GC is optimized for execution time (fewer collections). Above this limit, the GC is optimized for memory usage (more collections).
- `evacuation-threshold` = *threshold* : Sets the evacuation threshold in percent. The value must be an integer in the range 0 to 100. The default is 66. If the sweep phase of the collection finds that the occupancy of a specific heap block type is less than this percentage, it will do a copying collection for that block type in the next major collection, thereby restoring occupancy to close to 100 percent. A value of 0 turns evacuation off.
- `bridge-implementation` = *bridge implementation* : This will set the GC Bridge option to help address GC performance issues. There are three possible values: *old* , *new* , *tarjan*.
- `bridge-require-precise-merge` : The Tarjan bridge contains an optimization which may, on rare occasions, cause an object to be collected one GC after it first becomes garbage. Including this option disables that optimization, making GCs more predictable but potentially slower.

For example, to configure the GC to have a heap size limit of 128MB, add a new file to your Project with a **Build action** of `AndroidEnvironment` with the contents:

```
MONO_GC_PARAMS=soft-heap-limit=128m
```

Limitations

10/28/2019 • 3 minutes to read • Edit Online

Since applications on Android require generating Java proxy types during the build process, it is not possible to generate all code at runtime.

These are the Xamarin.Android limitations compared to desktop Mono:

Limited Dynamic Language Support

Android callable wrappers are needed any time the Android runtime needs to invoke managed code. Android callable wrappers are generated at compile time, based on static analysis of IL. The net result of this: you *cannot* use dynamic languages (IronPython, IronRuby, etc.) in any scenario where subclassing of Java types is required (including indirect subclassing), as there's no way of extracting these dynamic types at compile time to generate the necessary Android callable wrappers.

Limited Java Generation Support

Android Callable Wrappers need to be generated in order for Java code to call managed code. *By default*, Android callable wrappers will only contain (certain) declared constructors and methods which override a virtual Java method (i.e. it has `RegisterAttribute`) or implement a Java interface method (interface likewise has `Attribute`).

Prior to the 4.1 release, no additional methods could be declared. With the 4.1 release, the `Export` and `ExportField` custom attributes can be used to declare Java methods and fields within the Android Callable Wrapper.

Missing constructors

Constructors remain tricky, unless `ExportAttribute` is used. The algorithm for generating Android callable wrapper constructors is that a Java constructor will be emitted if:

1. There is a Java mapping for all the parameter types
2. The base class declares the same constructor – This is required because the Android callable wrapper *must* invoke the corresponding base class constructor; no default arguments can be used (as there's no easy way to determine what values should be used within Java).

For example, consider the following class:

```
[Service]
class MyIntentService : IntentService {
    public MyIntentService (): base ("value")
    {
    }
}
```

While this looks perfectly logical, the resulting Android callable wrapper *in Release builds* will not contain a default constructor. Consequently, if you attempt to start this service (e.g. `Context.StartService`, it will fail:

```
E/AndroidRuntime(31766): FATAL EXCEPTION: main
E/AndroidRuntime(31766): java.lang.RuntimeException: Unable to instantiate service example.MyIntentService:
java.lang.InstantiationException: can't instantiate class example.MyIntentService; no empty constructor
E/AndroidRuntime(31766):        at android.app.ActivityThread.handleCreateService(ActivityThread.java:2347)
E/AndroidRuntime(31766):        at android.app.ActivityThread.access$1600(ActivityThread.java:130)
E/AndroidRuntime(31766):        at android.app.ActivityThread$H.handleMessage(ActivityThread.java:1277)
E/AndroidRuntime(31766):        at android.os.Handler.dispatchMessage(Handler.java:99)
E/AndroidRuntime(31766):        at android.os.Looper.loop(Looper.java:137)
E/AndroidRuntime(31766):        at android.app.ActivityThread.main(ActivityThread.java:4745)
E/AndroidRuntime(31766):        at java.lang.reflect.Method.invokeNative(Native Method)
E/AndroidRuntime(31766):        at java.lang.reflect.Method.invoke(Method.java:511)
E/AndroidRuntime(31766):        at
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:786)
E/AndroidRuntime(31766):        at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:553)
E/AndroidRuntime(31766):        at dalvik.system.NativeStart.main(Native Method)
E/AndroidRuntime(31766): Caused by: java.lang.InstantiationException: can't instantiate class
example.MyIntentService; no empty constructor
E/AndroidRuntime(31766):        at java.lang.Class.newInstanceImpl(Native Method)
E/AndroidRuntime(31766):        at java.lang.Class.newInstance(Class.java:1319)
E/AndroidRuntime(31766):        at android.app.ActivityThread.handleCreateService(ActivityThread.java:2344)
E/AndroidRuntime(31766):        ... 10 more
```

The workaround is to declare a default constructor, adorn it with the `ExportAttribute`, and set the `ExportAttribute.SuperStringArgument`:

```
[Service]
class MyIntentService : IntentService {
    [Export (SuperArgumentsString = "\"value\"")]
    public MyIntentService (): base("value")
    {
    }

    // ...
}
```

Generic C# classes

Generic C# classes are only partially supported. The following limitations exist:

- Generic types may not use `[Export]` or `[ExportField`]. Attempting to do so will generate an `XA4207` error.

  ```
  public abstract class Parcelable<T> : Java.Lang.Object, IParcelable
  {
      // Invalid; generates XA4207
      [ExportField ("CREATOR")]
      public static IParcelableCreator CreateCreator ()
      {
          ...
  }
  ```

- Generic methods may not use `[Export]` or `[ExportField]`:

```
public class Example : Java.Lang.Object
{

    // Invalid; generates XA4207
    [Export]
    public static void Method<T>(T value)
    {
        ...
    }
}
```

- `[ExportField]` may not be used on methods which return `void`:

```
public class Example : Java.Lang.Object
{
    // Invalid; generates XA4208
    [ExportField ("CREATOR")]
    public static void CreateSomething ()
    {
    }
}
```

- Instances of Generic types *must not* be created from Java code. They can only safely be created from managed code:

```
[Activity (Label="Die!", MainLauncher=true)]
public class BadGenericActivity<T> : Activity
{
    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);
    }
}
```

Partial Java Generics Support

The Java generics binding support is limited. Particularly, members in a generic instance class that is derived from another generic (non-instantiated) class are left exposed as Java.Lang.Object. For example, Android.Content.Intent.GetParcelableExtra method returns Java.Lang.Object. This is due to erased Java generics. We have some classes that do not apply this limitation, but they are manually adjusted.

Related Links

- Android Callable Wrappers
- Working with JNI
- ExportAttribute
- SuperString
- RegisterAttribute

Troubleshooting Xamarin.Android

7/12/2021 • 2 minutes to read • Edit Online

Documents in this section cover features specific to troubleshooting with Android.

Troubleshooting Tips

Troubleshooting tips and tricks.

Frequently Asked Questions

Frequently asked Xamarin.Android troubleshooting questions.

Resolving Library Installation Errors

This guide provides workarounds for some common errors that may occur while referencing and automatically downloading Android Support Libraries or Google Play services.

Changes to the Android SDK Tooling

Starting in 26.0.1 of the Android SDK Tools, Google has removed the existing AVD and SDK managers in favour of new command line tooling.

Xamarin.Android Errors Reference

An errors reference guide, showing the most common errors you may experience when using Xamarin.Android in Visual Studio

Troubleshooting Tips

7/8/2021 • 22 minutes to read • Edit Online

Getting Diagnostic Information

Xamarin.Android has a few places to look when tracking down various bugs. These include:

1. Diagnostic MSBuild output.
2. Device deployment logs.
3. Android Debug Log Output.

Diagnostic MSBuild Output

Diagnostic MSBuild can contain additional information relating to package building and may contain some package deployment information.

To enable diagnostic MSBuild output within Visual Studio:

1. Click **Tools** > **Options...**
2. In the left-hand tree view, select **Projects and Solutions** > **Build and Run**
3. In the right-hand panel, set the MSBuild build output verbosity dropdown to Diagnostic
4. Click **OK**
5. Clean and rebuild your package.
6. Diagnostic output is visible within the Output panel.

To enable diagnostic MSBuild output within Visual Studio for Mac/OS X:

1. Click **Visual Studio for Mac** > **Preferences...**
2. In the left-hand tree view, select **Projects** > **Build**
3. In the right-hand panel, set the Log verbosity drop-down to Diagnostic
4. Click **OK**
5. Restart Visual Studio for Mac
6. Clean and rebuild your package.
7. Diagnostic output is visible within the Errors Pad (**View** > **Pads** > **Errors**), by clicking the Build Output button.

Device Deployment Logs

To enable device deployment logging within Visual Studio:

1. **Tools** > **Options...**>
2. In the left-hand tree view, select **Xamarin** > **Android Settings**
3. In the right-hand panel, enable the [X] **extension debug logging (writes monodroid.log to your desktop)** check box.
4. Log messages are written to the monodroid.log file on your desktop.

Visual Studio for Mac always writes device deployment logs. FInding them is slightly more difficult; a *AndroidUtils* log file is created for every day + time that a deployment occurs, for example: **AndroidTools-2012-10-24_12-35-45.log**.

- On Windows, log files are written to `%LOCALAPPDATA%\XamarinStudio-{VERSION}\Logs`.
- On OS X, log files are written to `$HOME/Library/Logs/XamarinStudio-{VERSION}`.

Android Debug Log Output

Android will write many messages to the Android Debug Log. Xamarin.Android uses Android system properties to control the generation of additional messages to the Android Debug Log. Android system properties can be set through the *setprop* command within the Android Debug Bridge (adb):

```
adb shell setprop PROPERTY_NAME PROPERTY_VALUE
```

System properties are read during process startup, and thus must be either set before the application is launched or the application must be restarted after the system properties are changed.

Xamarin.Android System Properties

Xamarin.Android supports the following system properties:

- *debug.mono.debug*: If a non-empty string, this is equivalent to `*mono-debug*`.
- *debug.mono.env*: A pipe-separated ('|') list of environment variables to export during application startup, *before* mono has been initialized. This allows setting environment variables that control mono logging.

  > **NOTE**
  >
  > Since the value is '|'-separated, the value must have an extra level of quoting, as the \`*adb shell*\` command will remove a set of quotes.

  > **NOTE**
  >
  > Android system property values can be no longer than 92 characters in length.

  Example:

  ```
  adb shell setprop debug.mono.env "'MONO_LOG_LEVEL=info|MONO_LOG_MASK=asm'"
  ```

- *debug.mono.log*: A comma-separated (',') list of components that should print additional messages to the Android Debug Log. By default, nothing is set. Components include:
  - *all*: Print all messages
  - *gc*: Print GC-related messages.
  - *gref*: Print (weak, global) reference allocation and deallocation messages.
  - *lref*: Print local reference allocation and deallocation messages.

  > **NOTE**
  >
  > These are *extremely* verbose. Do not enable unless you really need to.

- *debug.mono.trace*: Allows setting the mono --trace `=PROPERTY_VALUE` setting.

Deleting `bin` and `obj`

Xamarin.Android has suffered in the past from a situation such as:

- You encounter a strange build or runtime error.
- You `Clean`, `Rebuild`, or manually delete your `bin` and `obj` directories.
- The problem goes away.

We are heavily invested into fixing problems such as these due to their impact on developer productivity.

If a problem such as this happens to you:

1. Make a mental note. What was the last action that got your project into this state?
2. Save your current build log. Try building again, and record a diagnostic build log.
3. Submit a bug report.

Before deleting your `bin` and `obj` directories, zip them up and save them for later diagnosis if needed. You can probably merely `Clean` your Xamarin.Android application project to get things working again.

Xamarin.Android cannot resolve System.ValueTuple

This error occurs due to an incompatibility with Visual Studio.

- **Visual Studio 2017 Update 1** (version 15.1 or older) is only compatible with the **System.ValueTuple NuGet 4.3.0** (or older).
- **Visual Studio 2017 Update 2** (version 15.2 or newer) is only compatible with the **System.ValueTuple NuGet 4.3.1** (or newer).

Please choose the correct System.ValueTuple NuGet that corresponds with your Visual Studio 2017 installation.

GC Messages

GC component messages can be viewed by setting the debug.mono.log system property to a value that contains gc.

GC messages are generated whenever the GC executes and provides information about how much work the GC did:

```
I/monodroid-gc(12331): GC cleanup summary: 81 objects tested - resurrecting 21.
```

Additional GC information such as timing information can be generated by setting the `MONO_LOG_LEVEL` environment variable to `debug`:

```
adb shell setprop debug.mono.env MONO_LOG_LEVEL=debug
```

This will result in (lots of) additional Mono messages, including these three of consequence:

```
D/Mono (15723): GC_BRIDGE num-objects 1 num_hash_entries 81226 sccs size 81223 init 0.00ms df1 285.36ms sort
38.56ms dfs2 50.04ms setup-cb 9.95ms free-data 106.54ms user-cb 20.12ms clenanup 0.05ms links
5523436/5523436/5523096/1 dfs passes 1104 6883/11046605
D/Mono (15723): GC_MINOR: (Nursery full) pause 2.01ms, total 287.45ms, bridge 225.60 promoted 0K major
325184K los 1816K
D/Mono ( 2073): GC_MAJOR: (user request) pause 2.17ms, total 2.47ms, bridge 28.77 major 576K/576K los 0K/16K
```

In the `GC_BRIDGE` message, `num-objects` is the number of bridge objects this pass is considering, and `num_hash_entries` is the number of objects processed during this invocation of the bridge code.

In the `GC_MINOR` and `GC_MAJOR` messages, `total` is the amount of time while the world is paused (no threads are executing), while `bridge` is the amount of time taken in the bridge processing code (which deals with the Java VM). The world is *not* paused while bridge processing occurs.

In general, the larger the value of `num_hash_entries`, the more time that the `bridge` collections will take, and the larger the `total` time spent collecting will be.

Global Reference Messages

To enable Global Reference loggig (GREF) logging, the *debug.mono.log* system property must contain *gref*, e.g.:

```
adb shell setprop debug.mono.log gref
```

Xamarin.Android uses Android global references to provide mappings between Java instances and the associated managed instances, as when invoking a Java method a Java instance needs to be provided to Java.

Unfortunately, Android emulators only allow 2000 global references to exist at a time. Hardware has a much higher limit of 52000 global references. The lower limit can be problematic when running applications on the emulator, so knowing *where* the instance came from can be very useful.

> **NOTE**
>
> The global reference count is internal to Xamarin.Android, and does not (and cannot) include global references taken out by other native libraries loaded into the process. Use the global reference count as an estimate.

```
I/monodroid-gref(12405): +g+ grefc 108 gwrefc 0 obj-handle 0x40517468/L -> new-handle 0x40517468/L from
at Java.Lang.Object.RegisterInstance(IJavaObject instance, IntPtr value, JniHandleOwnership transfer)
I/monodroid-gref(12405):    at Java.Lang.Object.SetHandle(IntPtr value, JniHandleOwnership transfer)
I/monodroid-gref(12405):    at Java.Lang.Object..ctor(IntPtr handle, JniHandleOwnership transfer)
I/monodroid-gref(12405):    at Java.Lang.Thread+RunnableImplementor..ctor(System.Action handler, Boolean
removable)
I/monodroid-gref(12405):    at Java.Lang.Thread+RunnableImplementor..ctor(System.Action handler)
I/monodroid-gref(12405):    at Android.App.Activity.RunOnUiThread(System.Action action)
I/monodroid-gref(12405):    at Mono.Samples.Hello.HelloActivity.UseLotsOfMemory(Android.Widget.TextView
textview)
I/monodroid-gref(12405):    at Mono.Samples.Hello.HelloActivity.<OnCreate>m__3(System.Object o)
I/monodroid-gref(12405): handle 0x40517468; key_handle 0x40517468: Java Type:
`mono/java/lang/RunnableImplementor`; MCW type: `Java.Lang.Thread+RunnableImplementor`
I/monodroid-gref(12405): Disposing handle 0x40517468
I/monodroid-gref(12405): -g- grefc 107 gwrefc 0 handle 0x40517468/L from    at
Java.Lang.Object.Dispose(System.Object instance, IntPtr handle, IntPtr key_handle, JObjectRefType
handle_type)
I/monodroid-gref(12405):    at Java.Lang.Object.Dispose()
I/monodroid-gref(12405):    at Java.Lang.Thread+RunnableImplementor.Run()
I/monodroid-gref(12405):    at Java.Lang.IRunnableInvoker.n_Run(IntPtr jnienv, IntPtr native__this)
I/monodroid-gref(12405):    at System.Object.c200fe6f-ac33-441b-a3a0-47659e3f6750(IntPtr , IntPtr )
I/monodroid-gref(27679): +w+ grefc 1916 gwrefc 296 obj-handle 0x406b2b98/G -> new-handle 0xde68f4bf/W from
take_weak_global_ref_jni
I/monodroid-gref(27679): -w- grefc 1915 gwrefc 294 handle 0xde691aaf/W from take_global_ref_jni
```

There are four messages of consequence:

- Global reference creation: these are the lines that start with *+g+*, and will provide a stack trace for the creating code path.
- Global reference destruction: these are the lines that start with *-g-*, and may provide a stack trace for the code path disposing of the global reference. If the GC is disposing of the gref, no stack trace will be provided.
- Weak global reference creation: these are the lines that start with *+w+*.
- Weak global reference destruction: these are lines that start with *-w-*.

In all messages, The *grefc* value is the count of global references that Xamarin.Android has created, while the *grefwc* value is the count of weak global references that Xamarin.Android has created. The *handle* or *obj-handle* value is the JNI handle value, and the character after the ' */* is the type of handle value: */L* for local reference, */G* for global references, and */W* for weak global references.

As part of the GC process, global references (+g+) are converted into weak global references (causing a +w+ and -g-), a Java-side GC is kicked, and then the weak global reference is checked to see if it was collected. If it's still alive, a new gref is created around the weak ref (+g+, -w-), otherwise the weak ref is destroyed (-w).

Java instance is created and wrapped by a MCW

```
I/monodroid-gref(27679): +g+ grefc 2211 gwrefc 0 obj-handle 0x4066df10/L -> new-handle 0x4066df10/L from ...
I/monodroid-gref(27679): handle 0x4066df10; key_handle 0x4066df10: Java Type:
`android/graphics/drawable/TransitionDrawable`; MCW type: `Android.Graphics.Drawables.TransitionDrawable`
```

A GC is being performed...

```
I/monodroid-gref(27679): +w+ grefc 1953 gwrefc 259 obj-handle 0x4066df10/G -> new-handle 0xde68f95f/W from
take_weak_global_ref_jni
I/monodroid-gref(27679): -g- grefc 1952 gwrefc 259 handle 0x4066df10/G from take_weak_global_ref_jni
```

Object is still alive, as handle != null

wref turned back into a gref

```
I/monodroid-gref(27679): *try_take_global obj=0x4976f080 -> wref=0xde68f95f handle=0x4066df10
I/monodroid-gref(27679): +g+ grefc 1930 gwrefc 39 obj-handle 0xde68f95f/W -> new-handle 0x4066df10/G from
take_global_ref_jni
I/monodroid-gref(27679): -w- grefc 1930 gwrefc 38 handle 0xde68f95f/W from take_global_ref_jni
```

Object is dead, as handle == null

wref is freed, no new gref created

```
I/monodroid-gref(27679): *try_take_global obj=0x4976f080 -> wref=0xde68f95f handle=0x0
I/monodroid-gref(27679): -w- grefc 1914 gwrefc 296 handle 0xde68f95f/W from take_global_ref_jni
```

There is one "interesting" wrinkle here: on targets running Android prior to 4.0, the gref value is equal to the address of the Java object in the Android runtime's memory. (That is, the GC is a non-moving, conservative, collector, and it's handing out direct references to those objects.) Thus after a +g+, +w+, -g-, +g+, -w- sequence, the resulting gref will have the same value as the original gref value. This makes grepping through logs fairly straightforward.

Android 4.0, however, has a moving collector and no longer hands out direct references to Android runtime VM objects. Consequently, after a +g+, +w+, -g-, +g+, -w- sequence, the gref value *will be different*. If the object survives multiple GCs, it will go by several gref values, making it harder to determine where an instance was actually allocated from.

Querying Programmatically

You can query both the GREF and WREF counts by querying the `JniRuntime` object.

`Java.Interop.JniRuntime.CurrentRuntime.GlobalReferenceCount` - Global Reference Count

`Java.Interop.JniRuntime.CurrentRuntime.WeakGlobalReferenceCount` - Weak Reference Count

Android Debug Logs

The Android Debug Logs may provide additional context regarding any runtime errors you're seeing.

Floating-Point performance is terrible!

Alternatively, "My app runs 10x faster with the Debug build than with the Release build!"

Xamarin.Android supports multiple device ABIs: *armeabi*, *armeabi-v7a*, and *x86*. Device ABIs can be specified within **Project Properties > Application tab > Supported architectures**.

Debug builds use an Android package which provides all ABIs, and thus will use the fastest ABI for the target device.

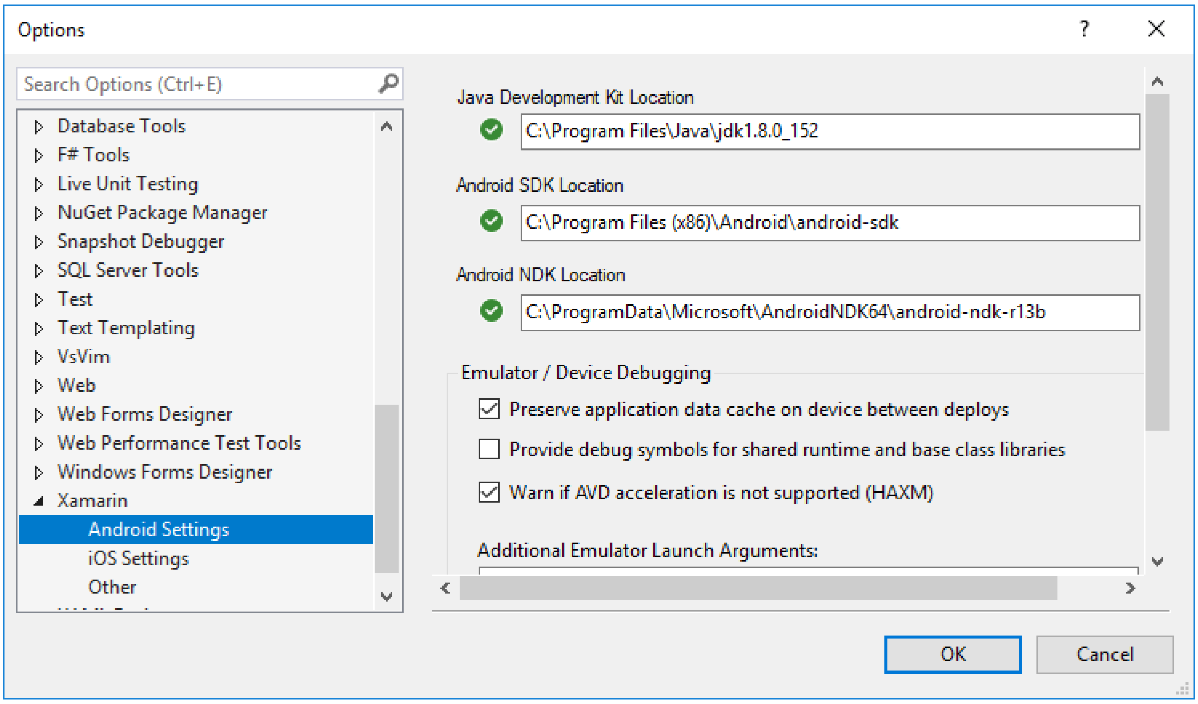
Release builds will only include the ABIs selected in the Project Properties tab. More than one can be selected.

armeabi is the default ABI, and has the broadest device support. *However*, armeabi doesn't support multi-CPU devices and hardware floating-point, amont other things. Consequently, apps using the armeabi Release runtime will be tied to a single core and will be using a soft-float implementation. Both of these can contribute to significantly slower performance for your app.

If your app requires decent floating-point performance (e.g. games), you should enable the *armeabi-v7a* ABI. You may want to only support the *armeabi-v7a* runtime, though this means that older devices which only support *armeabi* will be unable to run your app.

Could not locate Android SDK

There are 2 downloads available from Google for the Android SDK for Windows. If you choose the .exe installer, it will write registry keys that tell Xamarin.Android where it was installed. If you choose the .zip file and unzip it yourself, Xamarin.Android does not know where to look for the SDK. You can tell Xamarin.Android where the SDK is in Visual Studio by going to **Tools > Options > Xamarin > Android Settings**:



IDE does not display target device

Sometimes you will attempt to deploy your application to a device, but the device you want to deploy to isn't shown in the Select Device dialog. This can happen when the Android Debug Bridge decides to go on vacation.

To diagnose this issue, find the adb program, then run:

```
adb devices
```

If your device isn't present, then you need to restart the Android Debug Bridge server so that your device can be found:

```
adb kill-server
adb start-server
```

HTC Sync software may prevent **adb start-server** from working properly. If the **adb start-server** command doesn't print out which port it's starting on, please exit the HTC Sync software and try restarting the adb server.

The specified task executable "keytool" could not be run

This means that your PATH does not contain the directory where the Java SDK's bin directory is located. Check that you followed those steps from the Installation guide.

monodroid.exe or aresgen.exe exited with code 1

To help you debug this problem, go into Visual Studio and change the MSBuild verbosity level, to do this, select: **Tools** > **Options** > **Project** and **Solutions** > **Build** and **Run** > **MSBuild Project Build Output Verbosity** and set this value to **Normal**.

Rebuild, and check Visual Studio's Output pane, which should contain the full error.

There is not enough storage space on the device to deploy the package

This occurs when you don't start the emulator from within Visual Studio. When starting the emulator outside of Visual Studio, you need to pass the `-partition-size 512` options, e.g.

```
emulator -partition-size 512 -avd MonoDroid
```

Ensure you use the correct simulator name, i.e. the name you used when configuring the simulator.

INSTALL_FAILED_INVALID_APK when installing a package

Android package names *must* contain a period ('.'). Edit your package name so that it contains a period.

- Within Visual Studio:
  - Right click your project > Properties
  - Click the Android Manifest tab on the left.
  - Update the Package name field.
    - If you see the message "No AndroidManifest.xml found. Click to add one.", click the link and then update the Package name field.
- Within Visual Studio for Mac:

- Right click your project > Options.
- Navigate to the Build / Android Application section.
- Change the Package name field to contain a '.'.

INSTALL_FAILED_MISSING_SHARED_LIBRARY when installing a package

A "shared library" in this context is *not* a native shared library (*libfoo.so*) file; it is instead a library that must be separately installed on the target device, such as Google Maps.

The Android package specifies which shared libraries are required with the `<uses-library/>` element. If a *required* library is not present on the target device (e.g. `//uses-library/@android:required` is *true*, which is the default), then package installation will fail with *INSTALL_FAILED_MISSING_SHARED_LIBRARY*.

To determine which shared libraries are required, view the *generated* **AndroidManifest.xml** file (e.g. **obj\Debug\android\AndroidManifest.xml**) and look for the `<uses-library/>` elements. `<uses-library/>` elements can be added manually in your project's **Properties\AndroidManifest.xml** file and via the UsesLibraryAttribute custom attribute.

For example, adding an assembly reference to *Mono.Android.GoogleMaps.dll* will implicitly add a `<uses-library/>` for the Google Maps shared library.

INSTALL_FAILED_UPDATE_INCOMPATIBLE when installing a package

Android packages have three requirements:

- They must contain a '.' (see previous entry)
- They must have a unique string package name (hence the reverse-tld convention seen in Android app names, e.g. com.android.chrome for the Chrome app)
- When upgrading packages, the package must have the same signing key.

Thus, imagine this scenario:

1. You build & deploy your app as a Debug app
2. You change the signing key, e.g. to use as a Release app (or because you don't like the default-provided Debug signing key)
3. You install your app without removing it first, e.g. Debug > Start Without Debugging within Visual Studio

When this happens, package installation will fail with a INSTALL_FAILED_UPDATE_INCOMPATIBLE error, because the package name didn't change while the signing key did. The Android Debug Log will also contain a message similar to:

```
E/PackageManager(  146): Package [PackageName] signatures do not match the previously installed version;
ignoring!
```

To fix this error, completely remove the application from your device before re-installing.

INSTALL_FAILED_UID_CHANGED when installing a package

When an Android package is installed, it is assigned a *user id* (UID). *Sometimes*, for currently unknown reasons, when installing over an already installed app, the installation will fail with `INSTALL_FAILED_UID_CHANGED`:

```
ERROR [2015-03-23 11:19:01Z]: ANDROID: Deployment failed
Mono.AndroidTools.InstallFailedException: Failure [INSTALL_FAILED_UID_CHANGED]
   at Mono.AndroidTools.Internal.AdbOutputParsing.CheckInstallSuccess(String output, String packageName)
   at Mono.AndroidTools.AndroidDevice.<>c__DisplayClass2c.<InstallPackage>b__2b(Task`1 t)
   at System.Threading.Tasks.ContinuationTaskFromResultTask`1.InnerInvoke()
   at System.Threading.Tasks.Task.Execute()
```

To work around this issue, *fully uninstall* the Android package, either by installing the app from the Android target's GUI, or using `adb`:

```
$ adb uninstall @PACKAGE_NAME@
```

DO NOT USE `adb uninstall -k`, as this will *preserve* application data, and thus preserve the conflicting UID on the target device.

Release apps fail to launch on device

Does the Android Debug Log output will contain a message similar to:

```
D/AndroidRuntime( 1710): Shutting down VM
W/dalvikvm( 1710): threadid=1: thread exiting with uncaught exception (group=0xb412f180)
E/AndroidRuntime( 1710): FATAL EXCEPTION: main
E/AndroidRuntime( 1710): java.lang.UnsatisfiedLinkError: Couldn't load monodroid: findLibrary returned null
E/AndroidRuntime( 1710):        at java.lang.Runtime.loadLibrary(Runtime.java:365)
```

If so, there are two possible causes for this:

1. The .apk doesn't provide an ABI that the target device supports. For example, the .apk only contains armeabi-v7a binaries, and the target device only supports armeabi.

2. An Android bug. If this is the case, uninstall the app, cross your fingers, and reinstall the app.

To fix (1), edit the Project Options/Properties and add support for the required ABI to the list of Supported ABIs. To determine which ABI you need to add, run the following adb command against your target device:

```
adb shell getprop ro.product.cpu.abi
adb shell getprop ro.product.cpu.abi2
```

The output will contain the primary (and optional secondary) ABIs.

```
$ adb shell getprop | grep ro.product.cpu
[ro.product.cpu.abi2]: [armeabi]
[ro.product.cpu.abi]: [armeabi-v7a]
```

The OutPath property is not set for project “MyApp.csproj”

This generally means you have an HP computer and the environment variable “Platform” has been set to something like MCD or HPD. This conflicts with the MSBuild Platform property that is generally set to “Any CPU” or “x86”. You will need to remove this environment variable from your machine before MSBuild can function:

- Control Panel > System > Advanced > Environment Variables

Restart Visual Studio or Visual Studio for Mac and try to rebuild. Things should now work as expected.

java.lang.ClassCastException: mono.android.runtime.JavaObject cannot be cast to...

Xamarin.Android 4.x doesn't properly marshal nested generic types properly. For example, consider the following C# code using SimpleExpandableListAdapter:

```
// BAD CODE; DO NOT USE
var groupData = new List<IDictionary<string, object>> () {
        new Dictionary<string, object> {
                { "NAME", "Group 1" },
                { "IS_EVEN", "This group is odd" },
        },
};
var childData = new List<IList<IDictionary<string, object>>> () {
        new List<IDictionary<string, object>> {
                new Dictionary<string, object> {
                        { "NAME", "Child 1" },
                        { "IS_EVEN", "This group is odd" },
                },
        },
};
mAdapter = new SimpleExpandableListAdapter (
        this,
        groupData,
        Android.Resource.Layout.SimpleExpandableListItem1,
        new string[] { "NAME", "IS_EVEN" },
        new int[] { Android.Resource.Id.Text1, Android.Resource.Id.Text2 },
        childData,
        Android.Resource.Layout.SimpleExpandableListItem2,
        new string[] { "NAME", "IS_EVEN" },
        new int[] { Android.Resource.Id.Text1, Android.Resource.Id.Text2 }
);
```

The problem is that Xamarin.Android incorrectly marshals nested generic types. The `List<IDictionary<string, object>>` is being marshaled to a java.lang.ArrrayList, but the `ArrayList` is containing `mono.android.runtime.JavaObject` instances (which reference the `Dictionary<string, object>` instances) instead of something that implements java.util.Map, resulting in the following exception:

```
E/AndroidRuntime( 2991): FATAL EXCEPTION: main
E/AndroidRuntime( 2991): java.lang.ClassCastException: mono.android.runtime.JavaObject cannot be cast to
java.util.Map
E/AndroidRuntime( 2991):        at
android.widget.SimpleExpandableListAdapter.getGroupView(SimpleExpandableListAdapter.java:278)
E/AndroidRuntime( 2991):        at
android.widget.ExpandableListConnector.getView(ExpandableListConnector.java:446)
E/AndroidRuntime( 2991):        at android.widget.AbsListView.obtainView(AbsListView.java:2271)
E/AndroidRuntime( 2991):        at android.widget.ListView.makeAndAddView(ListView.java:1769)
E/AndroidRuntime( 2991):        at android.widget.ListView.fillDown(ListView.java:672)
E/AndroidRuntime( 2991):        at android.widget.ListView.fillFromTop(ListView.java:733)
E/AndroidRuntime( 2991):        at android.widget.ListView.layoutChildren(ListView.java:1622)
```

The workaround is to use the provided Java Collection types instead of the `System.Collections.Generic` types for the “inner” types. This will result in appropriate Java types when marshaling the instances. (The following code is more complicated than necessary in order to reduce gref lifetimes. It can be simplified to altering the original code via `s/List/JavaList/g` and `s/Dictionary/JavaDictionary/g` if gref lifetimes aren't a worry.)

```
// insert good code here
using (var groupData = new JavaList<IDictionary<string, object>> ()) {
    using (var groupEntry = new JavaDictionary<string, object> ()) {
        groupEntry.Add ("NAME", "Group 1");
        groupEntry.Add ("IS_EVEN", "This group is odd");
        groupData.Add (groupEntry);
    }
    using (var childData = new JavaList<IList<IDictionary<string, object>>> ()) {
        using (var childEntry = new JavaList<IDictionary<string, object>> ())
        using (var childEntryDict = new JavaDictionary<string, object> ()) {
            childEntryDict.Add ("NAME", "Child 1");
            childEntryDict.Add ("IS_EVEN", "This child is odd.");
            childEntry.Add (childEntryDict);
            childData.Add (childEntry);
        }
        mAdapter = new SimpleExpandableListAdapter (
            this,
            groupData,
            Android.Resource.Layout.SimpleExpandableListItem1,
            new string[] { "NAME", "IS_EVEN" },
            new int[] { Android.Resource.Id.Text1, Android.Resource.Id.Text2 },
            childData,
            Android.Resource.Layout.SimpleExpandableListItem2,
            new string[] { "NAME", "IS_EVEN" },
            new int[] { Android.Resource.Id.Text1, Android.Resource.Id.Text2 }
        );
    }
}
```

This will be fixed in a future release.

Unexpected NullReferenceExceptions

Occasionally the Android Debug Log will mention NullReferenceExceptions that “cannot happen,” or come from Mono for Android runtime code shortly before the app dies:

```
E/mono(15202): Unhandled Exception: System.NullReferenceException: Object reference not set to an instance
of an object
E/mono(15202):   at Java.Lang.Object.GetObject (IntPtr handle, System.Type type, Boolean owned)
E/mono(15202):   at Java.Lang.Object._GetObject[IOnTouchListener] (IntPtr handle, Boolean owned)
E/mono(15202):   at Java.Lang.Object.GetObject[IOnTouchListener] (IntPtr handle, Boolean owned)
E/mono(15202):   at
Android.Views.View+IOnTouchListenerAdapter.n_OnTouch_Landroid_view_View_Landroid_view_MotionEvent_(IntPtr
jnienv, IntPtr native__this, IntPtr native_v, IntPtr native_e)
E/mono(15202):   at (wrapper dynamic-method) object:b039cbb0-15e9-4f47-87ce-442060701362
(intptr,intptr,intptr,intptr)
```

or

```
E/mono    ( 4176): Unhandled Exception:
E/mono    ( 4176): System.NullReferenceException: Object reference not set to an instance of an object
E/mono    ( 4176): at Android.Runtime.JNIEnv.NewString (string)
E/mono    ( 4176): at Android.Util.Log.Info (string,string)
```

This can happen when the Android runtime decides to abort the process, which can happen for any number of reasons, including hitting the target's GREF limit or doing something “wrong” with JNI.

To see if this is the case, check the Android Debug Log for a message from your process similar to:

```
E/dalvikvm(  123): VM aborting
```

Abort due to Global Reference Exhaustion

The Android runtime's JNI layer only supports a limited number of JNI object references to be valid at any given point in time. When this limit is exceeded, things break.

The GREF (*global reference*) limit is 2000 references in the emulator, and ~52000 references on hardware.

You know you're starting to create too many GREFs when you see messages such as this in the Android Debug Log:

```
D/dalvikvm(  602): GREF has increased to 1801
```

When you reach the GREF limit, a message such as the following is printed:

```
D/dalvikvm(  602): GREF has increased to 2001
W/dalvikvm(  602): Last 10 entries in JNI global reference table:
W/dalvikvm(  602):  1991: 0x4057eff8 cls=Landroid/graphics/Point; (20 bytes)
W/dalvikvm(  602):  1992: 0x4057f010 cls=Landroid/graphics/Point; (28 bytes)
W/dalvikvm(  602):  1993: 0x40698e70 cls=Landroid/graphics/Point; (20 bytes)
W/dalvikvm(  602):  1994: 0x40698e88 cls=Landroid/graphics/Point; (20 bytes)
W/dalvikvm(  602):  1995: 0x40698ea0 cls=Landroid/graphics/Point; (28 bytes)
W/dalvikvm(  602):  1996: 0x406981f0 cls=Landroid/graphics/Point; (20 bytes)
W/dalvikvm(  602):  1997: 0x40698208 cls=Landroid/graphics/Point; (20 bytes)
W/dalvikvm(  602):  1998: 0x40698220 cls=Landroid/graphics/Point; (28 bytes)
W/dalvikvm(  602):  1999: 0x406956a8 cls=Landroid/graphics/Point; (20 bytes)
W/dalvikvm(  602):  2000: 0x406956c0 cls=Landroid/graphics/Point; (20 bytes)
W/dalvikvm(  602): JNI global reference table summary (2001 entries):
W/dalvikvm(  602):    51 of Ljava/lang/Class; 164B (41 unique)
W/dalvikvm(  602):    46 of Ljava/lang/Class; 188B (17 unique)
W/dalvikvm(  602):     6 of Ljava/lang/Class; 212B (6 unique)
W/dalvikvm(  602):    11 of Ljava/lang/Class; 236B (7 unique)
W/dalvikvm(  602):     3 of Ljava/lang/Class; 260B (3 unique)
W/dalvikvm(  602):     4 of Ljava/lang/Class; 284B (2 unique)
W/dalvikvm(  602):     8 of Ljava/lang/Class; 308B (6 unique)
W/dalvikvm(  602):     1 of Ljava/lang/Class; 316B
W/dalvikvm(  602):     4 of Ljava/lang/Class; 332B (3 unique)
W/dalvikvm(  602):     1 of Ljava/lang/Class; 356B
W/dalvikvm(  602):     2 of Ljava/lang/Class; 380B (1 unique)
W/dalvikvm(  602):     1 of Ljava/lang/Class; 428B
W/dalvikvm(  602):     1 of Ljava/lang/Class; 452B
W/dalvikvm(  602):     1 of Ljava/lang/Class; 476B
W/dalvikvm(  602):     2 of Ljava/lang/Class; 500B (1 unique)
W/dalvikvm(  602):     1 of Ljava/lang/Class; 548B
W/dalvikvm(  602):     1 of Ljava/lang/Class; 572B
W/dalvikvm(  602):     2 of Ljava/lang/Class; 596B (2 unique)
W/dalvikvm(  602):     1 of Ljava/lang/Class; 692B
W/dalvikvm(  602):     1 of Ljava/lang/Class; 956B
W/dalvikvm(  602):     1 of Ljava/lang/Class; 1004B
W/dalvikvm(  602):     1 of Ljava/lang/Class; 1148B
W/dalvikvm(  602):     2 of Ljava/lang/Class; 1172B (1 unique)
W/dalvikvm(  602):     1 of Ljava/lang/Class; 1316B
W/dalvikvm(  602):     1 of Ljava/lang/Class; 3428B
W/dalvikvm(  602):     1 of Ljava/lang/Class; 3452B
W/dalvikvm(  602):     1 of Ljava/lang/String; 28B
W/dalvikvm(  602):     2 of Ldalvik/system/VMRuntime; 12B (1 unique)
W/dalvikvm(  602):    10 of Ljava/lang/ref/WeakReference; 28B (10 unique)
W/dalvikvm(  602):     1 of Ldalvik/system/PathClassLoader; 44B
W/dalvikvm(  602):  1553 of Landroid/graphics/Point; 20B (1553 unique)
W/dalvikvm(  602):   261 of Landroid/graphics/Point; 28B (261 unique)
W/dalvikvm(  602):     1 of Landroid/view/MotionEvent; 100B
W/dalvikvm(  602):     1 of Landroid/app/ActivityThread$ApplicationThread; 28B
W/dalvikvm(  602):     1 of Landroid/content/ContentProvider$Transport; 28B
W/dalvikvm(  602):     1 of Landroid/view/Surface$CompatibleCanvas; 44B
W/dalvikvm(  602):     1 of Landroid/view/inputmethod/InputMethodManager$ControlledInputConnectionWrapper;
36B
W/dalvikvm(  602):     1 of Landroid/view/ViewRoot$1; 12B
W/dalvikvm(  602):     1 of Landroid/view/ViewRoot$W; 28B
W/dalvikvm(  602):     1 of Landroid/view/inputmethod/InputMethodManager$1; 28B
W/dalvikvm(  602):     1 of Landroid/view/accessibility/AccessibilityManager$1; 28B
W/dalvikvm(  602):     1 of Landroid/widget/LinearLayout$LayoutParams; 44B
W/dalvikvm(  602):     1 of Landroid/widget/LinearLayout; 332B
W/dalvikvm(  602):     2 of Lorg/apache/harmony/xnet/provider/jsse/TrustManagerImpl; 28B (1 unique)
W/dalvikvm(  602):     1 of Landroid/view/SurfaceView$MyWindow; 36B
W/dalvikvm(  602):     1 of Ltouchtest/RenderThread; 92B
W/dalvikvm(  602):     1 of Landroid/view/SurfaceView$3; 12B
W/dalvikvm(  602):     1 of Ltouchtest/DrawingView; 412B
W/dalvikvm(  602):     1 of Ltouchtest/Activity1; 180B
W/dalvikvm(  602): Memory held directly by tracked refs is 75624 bytes
E/dalvikvm(  602): Excessive JNI global references (2001)
E/dalvikvm(  602): VM aborting
```

In the above example (which, incidentally, comes from bug 685215) the problem is that too many Android.Graphics.Point instances are being created; see comment #2 for a list of fixes for this particular bug.

Typically, a useful solution is to find which type has too many instances allocated – Android.Graphics.Point in the above dump – then find where they're created in your source code and dispose of them appropriately (so that their Java-object lifetime is shortened). This is not always appropriate (#685215 is multithreaded, so the trivial solution avoids the Dispose call), but it's the first thing to consider.

You can enable GREF Logging to see when GREFs are created and how many exist.

Abort due to JNI type mismatch

If you hand-roll JNI code, it's possible that the types won't match correctly, e.g. if you try to invoke `java.lang.Runnable.run` on a type that doesn't implement `java.lang.Runnable`. When this occurs, there will be a message similar to this in the Android Debug Log:

```
W/dalvikvm( 123): JNI WARNING: can't call Ljava/Type;;.method on instance of Lanother/java/Type;
W/dalvikvm( 123):              in Lmono/java/lang/RunnableImplementor;.n_run:()V (CallVoidMethodA)
...
E/dalvikvm( 123): VM aborting
```

Dynamic Code Support

Dynamic code does not compile

To use C# dynamic in your application or library, you have to add System.Core.dll, Microsoft.CSharp.dll and Mono.CSharp.dll to your project.

In Release build, MissingMethodException occurs for dynamic code at run time.

- It is likely that your application project does not have references to System.Core.dll, Microsoft.CSharp.dll or Mono.CSharp.dll. Make sure those assemblies are referenced.
  - Keep in mind that dynamic code always costs. If you need efficient code, consider not using dynamic code.
- In the first preview, those assemblies were excluded unless types in each assembly are explicitly used by the application code. See the following for a workaround: http://lists.ximian.com/pipermail/mo...il/009798.html

Projects built with AOT+LLVM crash on x86 devices

When deploying an app built with AOT+LLVM on x86-based devices, you may see an exception error message similar to the following:

```
Assertion: should not be reached at /Users/.../external/mono/mono/mini/tramp-x86.c:124
Fatal signal 6 (SIGABRT), code -6 in tid 4051 (Xamarin.bug56111)
```

This is a known issue – the workaround is to disable LLVM.

Which Android SDK packages should I install?

1/24/2020 • 2 minutes to read • Edit Online

Installing the Android SDK doesn't automatically include all the minimum required packages for developing. While individual developer needs vary, the following packages will generally be required for developing with Xamarin.Android:

Tools

Install the latest tools from the Tools folder in the SDK manager:

- Android SDK Tools
- Android SDK Platform-Tools
- Android SDK Build-Tools

Android Platform(s)

Install the "SDK Platform" for the Android versions you've set as minimum & target.

Examples:

- Target API 23
- Minimum API 23

Only need to install SDK Platform for API 23

- Target API 23
- Minimum API 15

Need to install SDK Platforms for API 15 and 23. Note that you do not need to install the API levels between the minimum and target (even if you are backporting to those API levels).

System Images

These are only required if you want to use the out-of-the-box Android emulators from Google. For more information, see Android Emulator Setup

Extras

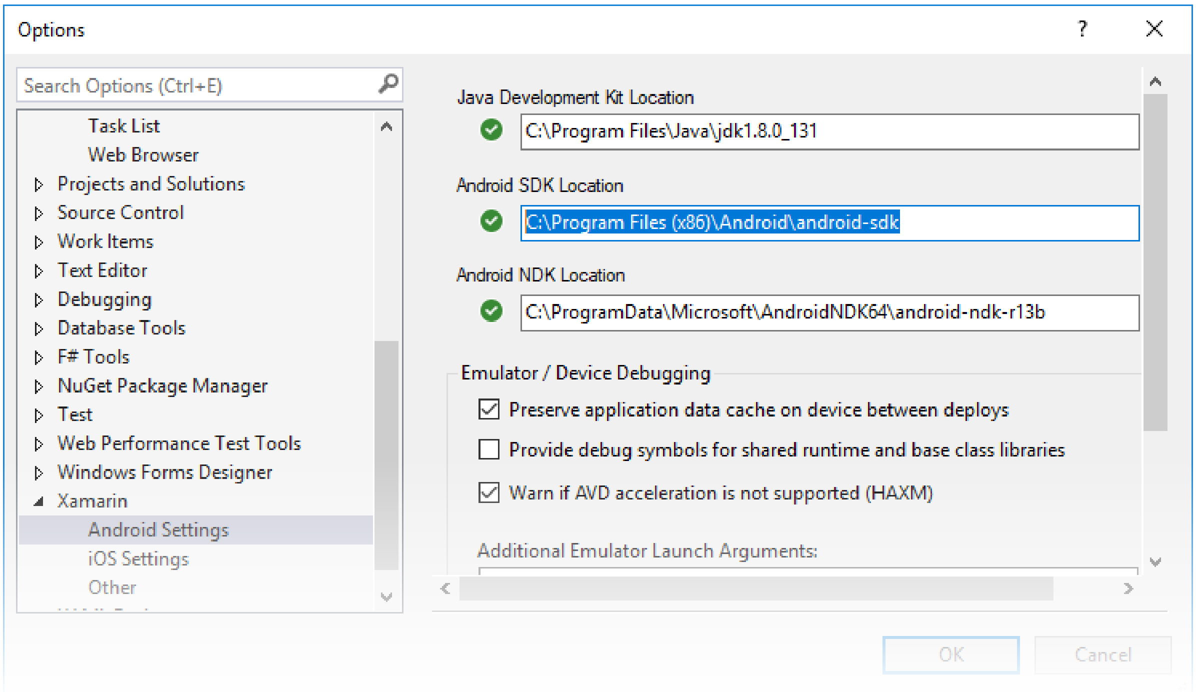
The Android SDK Extras are usually not required; but it is useful to be aware of them since they may be required depending on your use case.

Where can I set my Android SDK locations?

7/8/2021 • 2 minutes to read • Edit Online

- Visual Studio
- Visual Studio for Mac

In Visual Studio, navigate to **Tools > Options > Xamarin > Android Settings** to view and set the Android SDK location:



The default location for each path is as follows:

- Java Development Kit Location:

  C:\Program Files\Java\jdk1.8.0_131

- Android SDK Location:

  C:\Program Files (x86)\Android\android-sdk

- Android NDK Location:

  C:\ProgramData\Microsoft\AndroidNDK64\android-ndk-r13b

Note that the version number of the NDK may vary. For example, instead of **android-ndk-r13b**, it could be an earlier version such as **android-ndk-r10e**.

To set the Android SDK location, enter the full path of the Android SDK directory into the **Android SDK Location** box. You can navigate to the Android SDK location in File Explorer, copy the path from the address bar, and paste this path into the **Android SDK Location** box. For example, if your Android SDK location is at **C:\Users\username\AppData\Local\Android\Sdk**, clear the old path in the **Android SDK Location** box, paste in this path, and click **OK**.

How do I update the Java Development Kit (JDK) version?

7/8/2021 • 2 minutes to read • Edit Online

This article illustrates how to update the Java Development Kit (JDK) version on Windows and Mac.

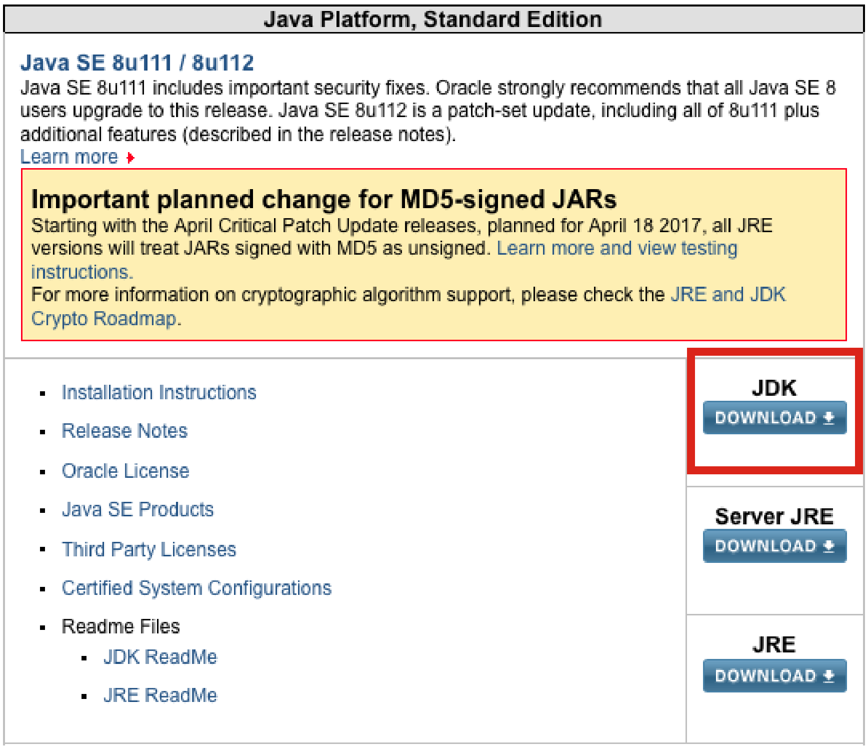
Overview

Xamarin.Android uses the Java Development Kit (JDK) to integrate with the Android SDK for building Android apps and running the Android designer. The latest versions of the Android SDK (API 24 and higher) require JDK 8 (1.8). Alternately, you can install the Microsoft Mobile OpenJDK Preview. The Microsoft Mobile OpenJDK will eventually replace JDK 8 for Xamarin.Android development.

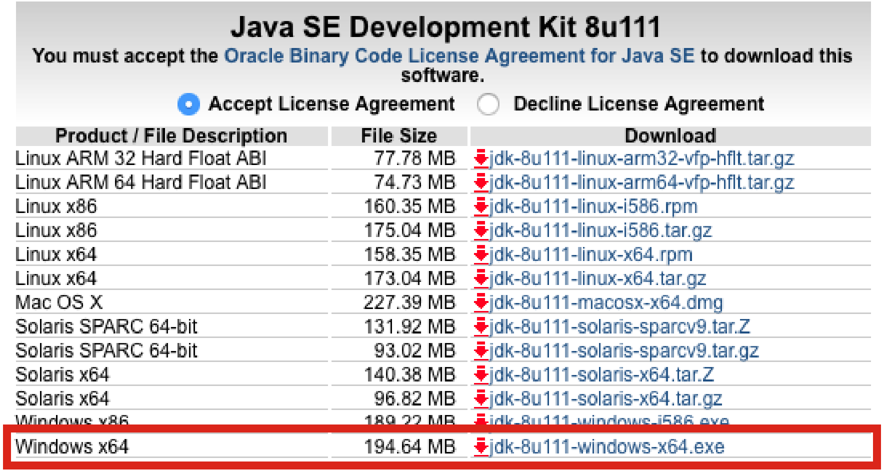
To update to the Microsoft Mobile OpenJDK, see Microsoft Mobile OpenJDK Preview. To update to JDK 8, follow these steps:

- Visual Studio
- Visual Studio for Mac

1. Download JDK 8 (1.8) from the Oracle website:

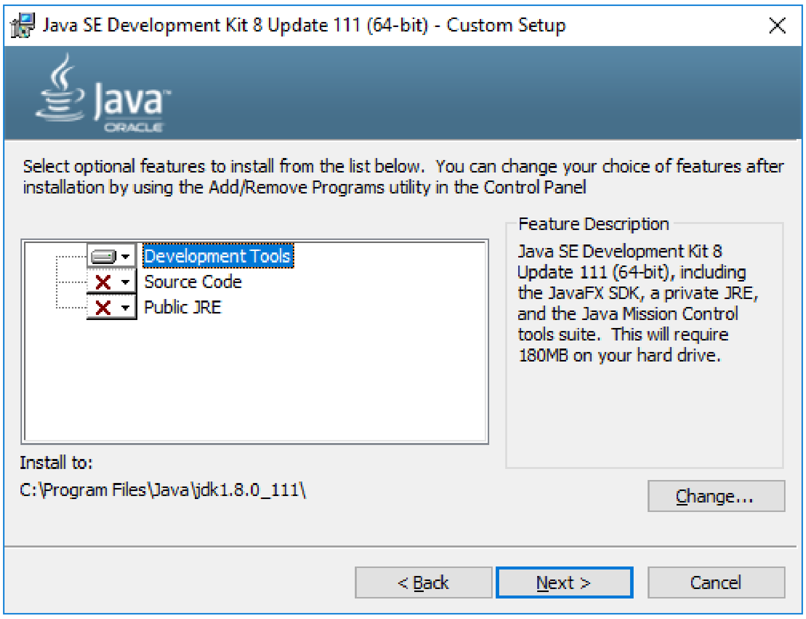


2. Pick the 64-bit version to allow rendering of custom controls in the Xamarin Android designer:

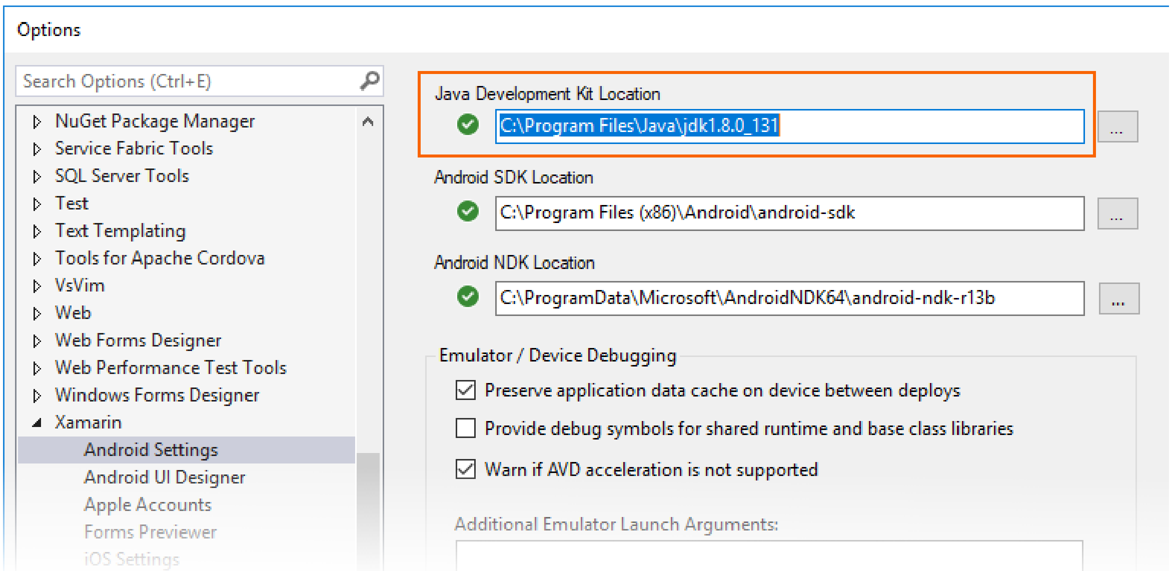


Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	77.78 MB	jdk-8u111-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	74.73 MB	jdk-8u111-linux-arm64-vfp-hflt.tar.gz
Linux x86	160.35 MB	jdk-8u111-linux-i586.rpm
Linux x86	175.04 MB	jdk-8u111-linux-i586.tar.gz
Linux x64	158.35 MB	jdk-8u111-linux-x64.rpm
Linux x64	173.04 MB	jdk-8u111-linux-x64.tar.gz
Mac OS X	227.39 MB	jdk-8u111-macosx-x64.dmg
Solaris SPARC 64-bit	131.92 MB	jdk-8u111-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	93.02 MB	jdk-8u111-solaris-sparcv9.tar.gz
Solaris x64	140.38 MB	jdk-8u111-solaris-x64.tar.Z
Solaris x64	96.82 MB	jdk-8u111-solaris-x64.tar.gz
Windows x86	189.22 MB	jdk-8u111-windows-i586.exe
Windows x64	194.64 MB	jdk-8u111-windows-x64.exe

3. Run the .exe and install the **Development Tools**:



4. Open Visual Studio and update the **Java Development Kit Location** to point to the new JDK under **Tools > Options > Xamarin > Android Settings > Java Development Kit Location**:



Be sure to restart Visual Studio after updating the location.

Xamarin.Android and Java Development Kit 9 or later

9/17/2021 • 2 minutes to read • Edit Online

This article explains how to resolve Java Development Kit (JDK) 9 or later errors in Xamarin.Android.

Overview

Xamarin.Android uses the Java Development Kit (JDK) to integrate with the Android SDK for building Android apps and running the Android designer. The latest versions of the Android SDK (API 24 and higher) require JDK 8 (1.8) or the Microsoft Mobile OpenJDK Preview. **Because the Android SDK tools available from Google are not yet compatible with JDK 9, Xamarin.Android does not work with JDK 9 or later.**

> **NOTE**
>
> To target Android API 31, you need to install JDK 11. Learn more about JDK 11 impacts to Visual Studio here.

JDK Errors

If you try to build a Xamarin.Android project with a version of the JDK later than JDK 8, you will get an explicit error indicating that this version of JDK is not supported. For example:

```
Building with JDK Version `9.0.4` is not supported. Please install JDK version `1.8.0`. See
https://aka.ms/xamarin/jdk9-errors
```

To resolve these errors, you must install JDK 8 (1.8) as explained in How do I update the Java Development Kit (JDK) version?. Alternately, you can install the Microsoft Mobile OpenJDK Preview The Microsoft Mobile OpenJDK will eventually replace JDK 8 for Xamarin.Android development.

Checking the JDK Version

You can check to see which version of Java you have installed by entering the following command (the JDK `bin` directory must be in your `PATH`):

```
java -version
```

If JDK 9 installed, you will see a message like the following:

```
java version "9.0.4"
Java(TM) SE Runtime Environment (build 9.0.4+11)
Java HotSpot(TM) 64-Bit Server VM (build 9.0.4+11, mixed mode)
```

If JDK 9 or later is installed, you must install Java JDK 8 (1.8) or the Microsoft Mobile OpenJDK Preview. For information about how to install JDK 8, see How do I update the Java Development Kit (JDK) version?. For information about how to install the Microsoft Mobile OpenJDK, see Microsoft Mobile OpenJDK Preview.

Note that you do not have to uninstall a later version of the JDK; however, you must ensure that Xamarin is using JDK 8 rather than a later JDK version. In Visual Studio, click **Tools > Options > Xamarin > Android**

Settings. If **Java Development Kit Location** is not set to a JDK 8 location (such as **C:\Program Files\Java\jdk1.8.0_111**), click **Change** and set it to the location where JDK 8 is installed. In Visual Studio for Mac, navigate to **Preferences > Projects > SDK Locations > Android > Java SDK (JDK)** and click **Browse** to update this path.

Known Issues with JDK 9

apksigner

There is a known issue with apksigner and JDK 9 in which the `apksigner.bat` file invokes the `apksigner.jar` with `-Djava.ext.dirs` instead of `-classpath` which JDK 9 expects. It is recommended to use JDK 8 (1.8). For information about how to install JDK 8, see How do I update the Java Development Kit (JDK) version?

If you have installed JDK 9, ensure that the following path is not set on your `PATH` environment variable as it will still point to JDK 9: `C:\ProgramData\Oracle\Java\javapath`. After removing it, `java-version` at a command line should show JDK 8.

How can I manually install the Android Support libraries required by the Xamarin.Android.Support packages?

7/8/2021 • 4 minutes to read • Edit Online

Example steps for Xamarin.Android.Support.v4

- Visual Studio
- Visual Studio for Mac

Download the desired Xamarin.Android.Support NuGet package (for example by installing it with the NuGet package manager).

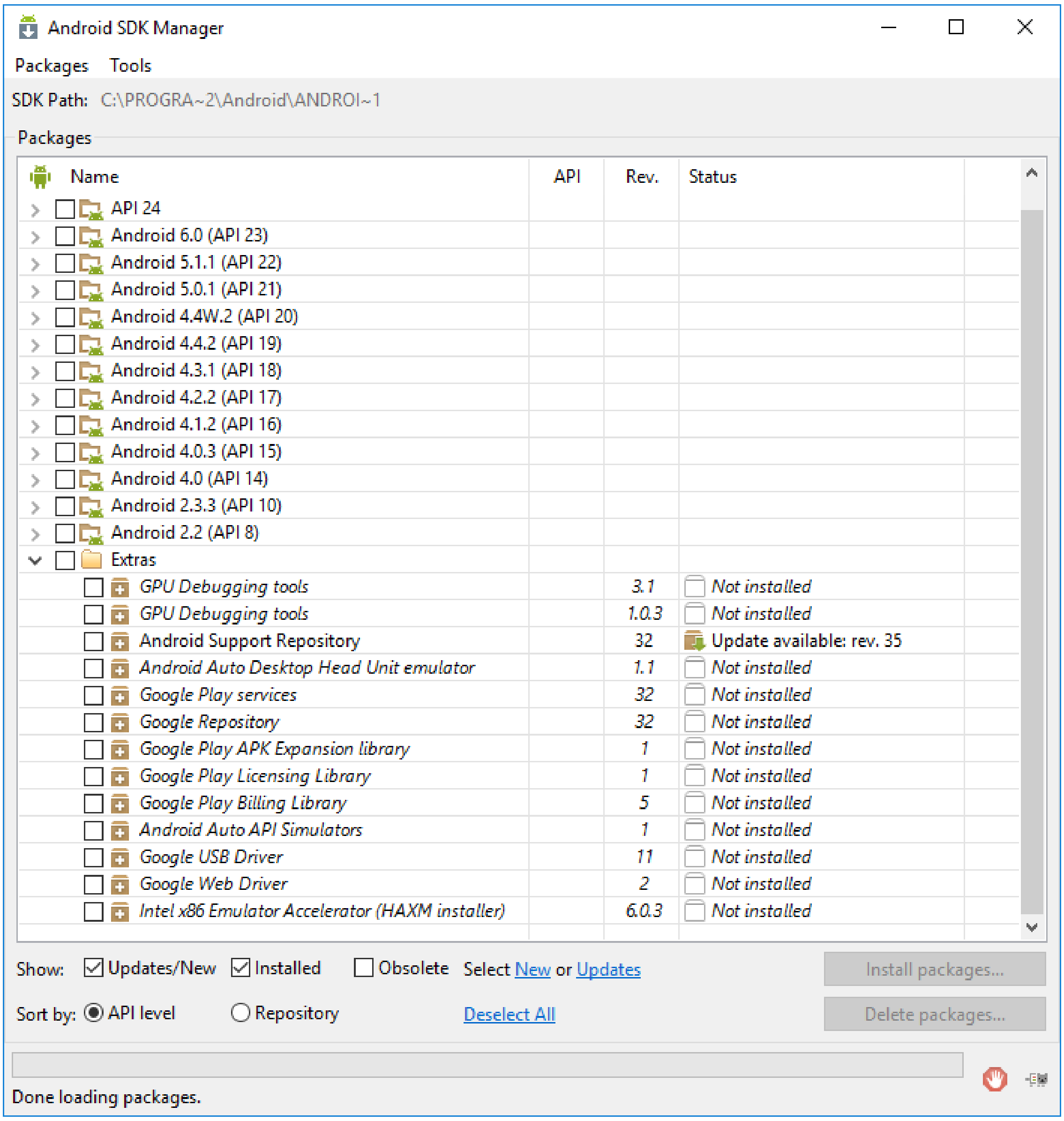
Use `ildasm` to check which version of **android_m2repository.zip** the NuGet package needs:

```
ildasm /caverbal /text /item:Xamarin.Android.Support.v4
packages\Xamarin.Android.Support.v4.23.4.0.1\lib\MonoAndroid403\Xamarin.Android.Support.v4.dll | findstr
SourceUrl
```

Example output:

```
property string 'SourceUrl' = string('https://dl-
ssl.google.com/android/repository/android_m2repository_r32.zip')
property string 'SourceUrl' = string('https://dl-
ssl.google.com/android/repository/android_m2repository_r32.zip')
property string 'SourceUrl' = string('https://dl-
ssl.google.com/android/repository/android_m2repository_r32.zip')
```

Download **android_m2repository.zip** from Google using the URL returned from **ildasm**. Alternately, you can check which version of the *Android Support Repository* you currently have installed in the Android SDK Manager:



If the version matches the one you need for the NuGet package, then you don't have to download anything new. You can instead re-zip the existing **m2repository** directory that is located under **extras\android** in the *SDK Path* (as shown the top of the Android SDK Manager window).

Calculate the MD5 hash of the URL returned from **ildasm**. Format the resulting string to use all uppercase letters and no spaces. For example, adjust the `$url` variable as needed and then run the following 2 lines (based on the original C# code from Xamarin.Android) in PowerShell:

```
$url = "https://dl-ssl.google.com/android/repository/android_m2repository_r32.zip"
(([System.Security.Cryptography.MD5]::Create()).ComputeHash([System.Text.Encoding]::UTF8.GetBytes($url)) | %
{ $_.ToString("X02") }) -join ""
```

Example output:

```
F16A3455987DBAE5783F058F19F7FCDF
```

Copy **android_m2repository.zip** into the **%LOCALAPPDATA%\Xamarin\zips** folder. Rename the file to use the MD5 hash from the previous MD5 hash calculating step. For example:

%LOCALAPPDATA%\Xamarin\zips\F16A3455987DBAE5783F058F19F7FCDF.zip

(Optional) Unzip the file into

%LOCALAPPDATA%\Xamarin\Xamarin.Android.Support.v4\23.4.0.0\content (creating a **content\m2repository** subdirectory). If you skip this step, then the first build that uses the library will take a little longer because it will need to complete this step. The version number for the subdirectory (**23.4.0.0** in this example) is not quite the same as the NuGet package version. You can use `ildasm` to find the correct version number:

```
ildasm /caverbal /text /item:Xamarin.Android.Support.v4
packages\Xamarin.Android.Support.v4.23.4.0.1\lib\MonoAndroid403\Xamarin.Android.Support.v4.dll | findstr
/C:"string 'Version'"
```

Example output:

```
property string 'Version' = string('23.4.0.0')}
property string 'Version' = string('23.4.0.0')}
property string 'Version' = string('23.4.0.0')}
```

Additional references

- Bug 43245 – Inaccurate "Download failed. Please download {0} and put it to the {1} directory." and "Please install package: '{0}' available in SDK installer" error messages related to Xamarin.Android.Support packages

Next Steps

This document discusses the current behavior as of August 2016. The technique described in this document is not part of the stable testing suite for Xamarin, so it could break in the future.

For further assistance, to contact us, or if this issue remains even after utilizing the above information, please see What support options are available for Xamarin? for information on contact options, suggestions, as well as how to file a new bug if needed.

What USB drivers do I need to debug Android on Windows?

11/2/2020 • 2 minutes to read • Edit Online

Finding USB Drivers

To debug on an Android device when developing in Windows; you need to install a compatible USB driver. The Android SDK Manager includes the "Google USB Driver" by default, which adds support for Nexus devices as described here: https://developer.android.com/sdk/win-usb.html

Other devices require USB drivers specifically published by the device manufacturer. Some links for the most common manufacturers are included in this guide: https://developer.android.com/tools/extras/oem-usb.html

Alternatives

Depending on the manfacturer, it can be difficult to track down the exact USB driver needed. Some alternatives for testing Android apps developed in Windows including using an Android emulator or using external testing services. Some of these include:

- App Center Test - Cloud Testing services run on hundreds of real Android devices.
- Visual Studio Emulator for Android
- Debugging on the Android Emulator

Is it possible to connect to Android emulators running on a Mac from a Windows VM?

11/2/2020 • 3 minutes to read • Edit Online

To connect to the Android Emulator running on a Mac from a Windows virtual machine, use the following steps:

> **NOTE**
> We recommend using an Android Emulator that does not include the Google Play Store.

1. Start the emulator on the Mac.
2. Kill the `adb` server on the Mac:

```
adb kill-server
```

3. Note that the emulator is listening on 2 TCP ports on the loopback network interface:

```
lsof -iTCP -sTCP:LISTEN -P | grep 'emulator\|qemu'

emulator6 94105 macuser   20u  IPv4 0xa8dacfb1d4a1b51f      0t0  TCP localhost:5555 (LISTEN)
emulator6 94105 macuser   21u  IPv4 0xa8dacfb1d845a51f      0t0  TCP localhost:5554 (LISTEN)
```

The odd-numbered port is the one used to connect to `adb`. See also https://developer.android.com/tools/devices/emulator.html#emulatornetworking.

4. *Option 1*: Use `nc` to forward inbound TCP packets received externally on port 5555 (or any other port you like) to the odd-numbered port on the loopback interface (**127.0.0.1 5555** in this example), and to forward the outbound packets back the other way:

```
cd /tmp
mkfifo backpipe
nc -kl 5555 0<backpipe | nc 127.0.0.1 5555 > backpipe
```

As long as the `nc` commands stay running in a Terminal window, the packets will be forwarded as expected. You can type Control-C in the Terminal window to quit the `nc` commands once you're done using the emulator.

(Option 1 is usually easier than Option 2, especially if **System Preferences > Security & Privacy > Firewall** is switched on.)

Option 2: Use `pfctl` to redirect TCP packets from port `5555` (or any other port you like) on the Shared Networking interface to the odd-numbered port on the loopback interface (`127.0.0.1:5555` in this example):

```
sed '/rdr-anchor/a rdr pass on vmnet8 inet proto tcp from any to any port 5555 -> 127.0.0.1 port
5555' /etc/pf.conf | sudo pfctl -ef -
```

This command sets up port forwarding using the `pf packet filter` system service. The line breaks are important. Be sure to keep them intact when copy-pasting. You will also need to adjust the interface name from *vmnet8* if you're using Parallels. `vmnet8` is the name of the special *NAT device* for the *Shared Networking* mode in VMWare Fusion. The appropriate network interface in Parallels is likely vnic0.

5. Connect to the emulator from the Windows machine:

```
C:\> adb connect ip-address-of-the-mac:5555
```

   Replace "ip-address-of-the-mac" with the IP address of the Mac, for example as listed by `ifconfig vmnet8 | grep 'inet '`. If needed, replace `5555` with the other port you like from step 4. (Note: one way to get command-line access to `adb` is via **Tools > Android > Android Adb Command Prompt** in Visual Studio.)

Alternate technique using `ssh`

If you have enabled *Remote Login* on the Mac, then you can use `ssh` port forwarding to connect to the emulator.

1. Install an SSH client on Windows. One option is to install Git for Windows. The `ssh` command will then be available in the **Git Bash** command prompt.

2. Follow steps 1-3 from above to start the emulator, kill the `adb` server on the Mac, and identify the emulator ports.

3. Run `ssh` on Windows to set up two-way port forwarding between a local port on Windows (`localhost:15555` in this example) and the odd-numbered emulator port on the Mac's loopback interface (`127.0.0.1:5555` in this example):

```
C:\> ssh -L localhost:15555:127.0.0.1:5555 mac-username@ip-address-of-the-mac
```

   Replace `mac-username` with your Mac username as listed by `whoami`. Replace `ip-address-of-the-mac` with the IP address of the Mac.

4. Connect to the emulator using the local port on Windows:

```
C:\> adb connect localhost:15555
```

   (Note: one easy way to get command-line access to `adb` is via **Tools > Android > Android Adb Command Prompt** in Visual Studio.)

A small caution: if you use port `5555` for the local port, `adb` will think that the emulator is running locally on Windows. This doesn't cause any trouble in Visual Studio, but in Visual Studio for Mac it causes the app to exit immediately after launch.

Alternate technique using `adb -H` is not yet supported

In theory, another approach would be to use `adb`'s built-in capability to connect to an `adb` server running on a remote machine (see for example https://stackoverflow.com/a/18551325). But the Xamarin.Android IDE extensions do not currently provide a way to configure that option.

Contact information

This document discusses the current behavior as of March, 2016. The technique described in this document is not part of the stable testing suite for Xamarin, so it could break in the future.

If you notice that the technique no longer works, or if you notice any other mistakes in the document, feel free to add to the discussion on the following forum thread: http://forums.xamarin.com/discussion/33702/android-emulator-from-host-device-inside-windows-vm. Thanks!

How do I automate an Android NUnit Test project?

11/2/2020 • 2 minutes to read • Edit Online

> **NOTE**
>
> This guide explains how to automate an Android NUnit test project, not a Xamarin.UITest project. Xamarin.UITest guides can be found here.

When you create a **Unit Test App (Android)** project in Visual Studio (or **Android Unit Test** project in Visual Studio for Mac), this project will not automatically run your tests by default. To run NUnit tests on a target device, you can create an Android.App.Instrumentation subclass that is started by using the following command:

```
adb shell am instrument
```

The following steps explain this process:

1. Create a new file called **TestInstrumentation.cs**:

   ```
   using System;
   using System.Reflection;
   using Android.App;
   using Android.Content;
   using Android.Runtime;
   using Xamarin.Android.NUnitLite;

   namespace App.Tests {

       [Instrumentation(Name="app.tests.TestInstrumentation")]
       public class TestInstrumentation : TestSuiteInstrumentation {

           public TestInstrumentation (IntPtr handle, JniHandleOwnership transfer) : base (handle,
   transfer)
           {
           }

           protected override void AddTests ()
           {
               AddTest (Assembly.GetExecutingAssembly ());
           }
       }
   }
   ```

   In this file, `Xamarin.Android.NUnitLite.TestSuiteInstrumentation` (from **Xamarin.Android.NUnitLite.dll**) is subclassed to create `TestInstrumentation`.

2. Implement the `TestInstrumentation` constructor and the `AddTests` method. The `AddTests` method controls which tests are actually executed.

3. Modify the `.csproj` file to add **TestInstrumentation.cs**. For example:

```
<?xml version="1.0" encoding="utf-8"?>
<Project DefaultTargets="Build" ToolsVersion="4.0"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    ...
    <ItemGroup>
        <Compile Include="TestInstrumentation.cs" />
    </ItemGroup>
    <Target Name="RunTests" DependsOnTargets="_ValidateAndroidPackageProperties">
        <Exec Command="&quot;$(_AndroidPlatformToolsDirectory)adb&quot; $(AdbTarget) $(AdbOptions)
shell am instrument -w $(_AndroidPackage)/app.tests.TestInstrumentation" />
    </Target>
    ...
</Project>
```

4. Deploy your application in debug or release mode, then stop it.

5. Use the following command to run the unit tests. Replace `PACKAGE_NAME` with the app's package name (the package name can be found in the app's `/manifest/@package` attribute located in **AndroidManifest.xml**):

```
adb shell am instrument -w PACKAGE_NAME/app.tests.TestInstrumentation
```

6. Optionally, you can modify the `.csproj` file to add the `RunTests` MSBuild target. This makes it possible to invoke the unit tests with a command like the following:

```
msbuild /t:RunTests Project.csproj
```

(Note that using this new target is not required; the earlier `adb` command can be used instead of `msbuild`.)

For more information about using the `adb shell am instrument` command to run unit tests, see the Android Developer Running tests with ADB topic.

> **NOTE**
>
> With the Xamarin.Android 5.0 release, the default package names for Android Callable Wrappers will be based on the MD5SUM of the assembly-qualified name of the type being exported. This allows the same fully-qualified name to be provided from two different assemblies and not get a packaging error. So make sure that you use the `Name` property on the `Instrumentation` attribute to generate a readable ACW/class name.

The ACW name must be used in the `adb` command above. Renaming/refactoring the C# class will thus require modifying the `RunTests` command to use the correct ACW name.

Why can't my Android release build connect to the Internet?

10/28/2019 • 2 minutes to read • Edit Online

Cause

The most common cause of this issue is that the **INTERNET** permission is automatically included in a debug build, but must be set manually for a release build. This is because the Internet permission is used to allow a debugger to attach to the process, as described for "DebugSymbols" here.

Fix

To resolve the issue, you can require the Internet permission in the Android Manifest. This can be done either through the manifest editor or the manifest's sourcecode:

- Fix in Editor: In your Android project, go to **Properties -> AndroidManifest.xml -> Required Permissions** and check **Internet**
- Fix in Sourcecode: Open the AndroidManifest in a source editor and add the permission tag inside the `<Manifest>` tags:

  ```
  <Manifest>
  ...
  <uses-permission android:name="android.permission.INTERNET" />
  </Manifest>
  ```

Smarter Xamarin Android Support v4 / v13 NuGet Packages

10/28/2019 • 2 minutes to read • Edit Online

About the Android Support Libraries

Google has created support libraries to make new features available to older versions of Android. In general, Support Libraries are given a version number in their name, which is the lowest Android API Level they are compatible with (eg: Support-v4 can only be used on API Level 4 and higher. More info in this Stack Overflow discussion).

Two of the support libraries: `Support-v4` and `Support-v13` can not be used together in the same app, that is, they are mutually exclusive. This is because `Support-v13` actually contains all of the types and implementation of `Support-v4`. If you try and reference both in the same project you will encounter duplicate type errors.

Problems with Referencing

Since `Support-v4` has become so popular, a lot of 3rd party libraries now depend on it. They could have chosen to depend on Support-v13 instead, but it's more common to depend on *v4* since that gives any apps using these 3rd party libraries the option of supporting API levels all the way down to 4.

If a Xamarin 3rd party library references the `Xamarin.Android.Support.v4.dll` binding to `Support-v4`, any app that uses this library must also reference `Xamarin.Android.Support.v4.dll`. This becomes a problem when the same app also wants to use some of the functionality from the `Xamarin.Android.Support.v13.dll` binding to `Support-v13`. If you reference both bindings, you will encounter duplicate type errors.

Type-Forwarded v4 Binding Assembly

To get around this problem, we have created a special `Xamarin.Android.Support.v4.dll` assembly which has no implementation, but simply `[assembly: TypeForwardedTo (..)]` attributes which forward all of the `Support-v4` types to the implementation within the `Xamarin.Android.Support.v13.dll` assembly.

This means a developer can reference this *type-forwarded* assembly in their app which will satisfy the reference to `Xamarin.Android.Support.v4.dll` by any 3rd party libraries, while still allowing `Xamarin.Android.Support.v13.dll` to be used in the app.

NuGet Assistance

While a developer could manually add the correct references necessary, we are able to use NuGet to help choose the right assembly (either the normal *v4* binding or the type-forwarded *v4* assembly) when the NuGet package is installed.

So, the `Xamarin.Android.Support.v4` NuGet package now contains the following logic:

If your app is targeting API Level 13 (Gingerbread 3.2) or higher:

- `Xamarin.Android.Support.v13` NuGet will automatically be added as a dependency
- The *type-forwarded* `Xamarin.Android.Support.v4.dll` will be referenced in the project

If your app is targeting anything lower than API Level 13, you will get the normal

`Xamarin.Android.Support.v4.dll` binding referenced in your project.

Do I have to use Support-v13?

If your app is targeting API Level 13 or higher and you choose to use the `Xamarin Android Support-v4` NuGet package, then the `Xamarin Android Support v13` NuGet package is a required dependency.

We feel the very minor increase in app size (the two .jar files differ by 17kb) is well worth the compatibility and fewer headaches it results in.

If you are adamant about using `Support-v4` in an app that targets API Level 13 or higher, you can always manually download the `.nupkg`, extract it, and reference the assembly.

How do I resolve a PathTooLongException error?

11/2/2020 • 2 minutes to read • Edit Online

Cause

Generated path names in a Xamarin.Android project can be quite long. For example, a path like the following could be generated during a build:

C:\Some\Directory\Solution\Project\obj\Debug\library_projects\Xamarin.Forms.Platform.Android\library_project_imports\assets

On Windows (where the maximum length for a path is 260 characters), a **PathTooLongException** could be produced while building the project if a generated path exceeds the maximum length.

Fix

The `UseShortFileNames` MSBuild property is set to `True` to circumvent this error by default. When this property is set to `True`, the build process uses shorter path names to reduce the likelihood of producing a **PathTooLongException**. For example, when `UseShortFileNames` is set to `True`, the above path is shortened to path that is similar to the following:

C:\Some\Directory\Solution\Project\obj\Debug\lp\1\jl\assets

To set this property manually, add the following MSBuild property to the project **.csproj** file:

```
<PropertyGroup>
    <UseShortFileNames>True</UseShortFileNames>
</PropertyGroup>
```

If setting this flag does not fix the **PathTooLongException** error, another approach is to specify a common intermediate output root for projects in your solution by setting `IntermediateOutputPath` in the project **.csproj** file. Try to use a relatively short path. For example:

```
<PropertyGroup>
    <IntermediateOutputPath>C:\Projects\MyApp</IntermediateOutputPath>
</PropertyGroup>
```

For more information about setting build properties, see Build Process.

What version of Xamarin.Android added Lollipop support?

11/2/2020 • 2 minutes to read • Edit Online

NOTE

This guide was originally written for the Android L preview.

- Xamarin.Android 4.17 added Android L Preview support.
- Xamarin.Android 4.20 added Android Lollipop support.

Xamarin only actively supports the current stable release of the Xamarin tools. The information below is provided "as-is" for older versions of the tools. For the latest information on Xamarin releases, please check the release notes.

"Missing android.jar for API Level 21" in Android L Preview

- Visual Studio
- Visual Studio for Mac

The following error message (or similar) may show up:

```
Error 1 Could not find android.jar for API Level 21.
```

This message means that the Android SDK platform for API Level 21 is not installed. Either install it in the Android SDK Manager (**Tools > Open Android SDK Manager...**), or change your Xamarin.Android project to target an API version that is installed.

There are a few workarounds for this issue:

1. Change your project so that it targets API 19 or lower.
2. Rename your android-21 folder from android-21 to android-L. (At best, this should only be used as a temporary fix, and it might not work very well at all.)

   %LOCALAPPDATA%\Android\android-sdk\platforms\android-21

3. Temporarily downgrade back to the Android API Level 21 "L" preview [1]:

   a. Delete the **%LOCALAPPDATA%\Android\android-sdk\platforms\android-21**
   b. Extract [1] into **C:\Users\<username>\AppData\Local\Android\android-sdk\platforms** to create an **android-L** folder.

[1] - https://dl-ssl.google.com/android/repository/android-L_r04.zip

Android.Support.v7.AppCompat - No resource found that matches the given name: attr 'android:actionModeShareDrawable'

1/7/2020 • 2 minutes to read • Edit Online

1. Make sure you download the latest extras as well as the Android 5.0 (API 21) SDK via the Android SDK Manager.
2. Ensure that you are compiling your application with compileSdkVersion set to 21. You can optionally set the targetSdkVersion to 21 as well.
3. If you require a previous version such as API 19, please download the respective version found on the NuGet page:

https://www.nuget.org/packages/Xamarin.Android.Support.v7.AppCompat/

> **NOTE**
> If you manually install this via Package Manager Console, make sure you also install the same version of Xamarin.Android.Support.v4

https://www.nuget.org/packages/Xamarin.Android.Support.v4/

Stack Overflow Reference: https://stackoverflow.com/questions/26431676/appcompat-v721-0-0-no-resource-found-that-matches-the-given-name-attr-andro

See Also

- Which Android SDK packages should I install?

Adjusting Java memory parameters for the Android designer

11/2/2020 • 2 minutes to read • Edit Online

The default memory parameters that are used when starting the `java` process for the Android designer might be incompatible with some system configurations.

Starting with Xamarin Studio 5.7.2.7 (and later, Visual Studio for Mac) and Visual Studio Tools for Xamarin 3.9.344, these settings can be customized on a per-project basis.

New Android designer properties and corresponding Java options

The following property names correspond to the indicated java command-line option

- **AndroidDesignerJavaRendererMinMemory** -Xms
- **AndroidDesignerJavaRendererMaxMemory** -Xmx
- **AndroidDesignerJavaRendererPermSize** -XX:MaxPermSize

- Visual Studio
- Visual Studio for Mac

1. Open your solution in Visual Studio.
2. Select each Android project one-by-one in the Solution Explorer and click Show All Files twice on each project. You can skip projects that do not contain any `.axml` layout files. This step will ensure that each project directory contains a `.csproj.user` file.
3. Quit Visual Studio.
4. Locate the `.csproj.user` file for each of the projects from step 2.
5. Edit each `.csproj.user` file in a text editor.
6. Add any or all of the new Android designer memory properties within a `<PropertyGroup>` element. You can use an existing `<PropertyGroup>` or create a new one. Here's a complete example `.csproj.user` file that includes all 3 attributes set to their default values:

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="12.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
   <PropertyGroup>
     <ProjectView>ProjectFiles</ProjectView>
   </PropertyGroup>
   <PropertyGroup>
     <AndroidDesignerJavaRendererMinMemory>128m</AndroidDesignerJavaRendererMinMemory>
     <AndroidDesignerJavaRendererMaxMemory>750m</AndroidDesignerJavaRendererMaxMemory>
     <AndroidDesignerJavaRendererPermSize>350m</AndroidDesignerJavaRendererPermSize>
   </PropertyGroup>
</Project>
```

7. Save and close all of the updated `.csproj.user` files.
8. Restart Visual Studio and reopen your solution.

My Android Resource.designer.cs file will not update

10/28/2019 • 2 minutes to read • Edit Online

NOTE

This issue has been resolved in Xamarin Studio 5.1.4 and later versions. However, if the issue occurs in Visual Studio for Mac, please file a new bug with your full versioning information and full build log output.

A bug in Xamarin.Studio 5.1 previously corrupted .csproj files by partially or completely deleting the xml code in the .csproj file. This would cause important parts of the Android build system (such as updating the Android Resource.designer.cs) to fail. As of the 5.1.4 stable release on July 15th, this bug has been fixed; but in many cases the project file has to be repaired manually, as described below.

Two possible approaches to fixing up the project file

Either:

1. Create a brand new Xamarin.Android application project, set all the project properties to match your old project, and add all of your resources, source files, etc. back into the project.

   OR

2. Make a backup copy of your original project's .csproj file, then open it in a text editor, and add back in the missing elements from a cleanly generated .csproj file.

If this does not solve the problem

After experimenting with these elements, you may notice that after adding back the elements and rebuilding the project, the Resource.designer.cs file would update, but then you might still have to close and re-open the solution to get code completion to recognize the new types contained in Resource.designer.cs.

Resolving Library Installation Errors

7/8/2021 • 5 minutes to read • Edit Online

In some cases, you may get errors while installing Android support libraries. This guide provides workarounds for some common errors.

Overview

While building a Xamarin.Android app project, you may get build errors when Visual Studio or Visual Studio for Mac attempt to download and install dependency libraries. Many of these errors are caused by network connectivity issues, file corruption, or versioning problems. This guide describes the most common support library installation errors and provides the steps to work around these issues and get your app project building again.

Errors While Downloading m2Repository

You may see **m2repository** errors when referencing a NuGet package of the Android Support Libraries or Google Play services. The error message resembles the following:

```
Download failed. Please download https://dl-ssl.google.com/android/repository/android_m2repository_r16.zip
and extract it to the C:\Users\mgm\AppData\Local\Xamarin\Android.Support.v4\22.2.1\content directory.
```

This example is for **android_m2repository_r16**, but you may see this same error message for a different version such as **android_m2repository_r18** or **android_m2repository_r25**.

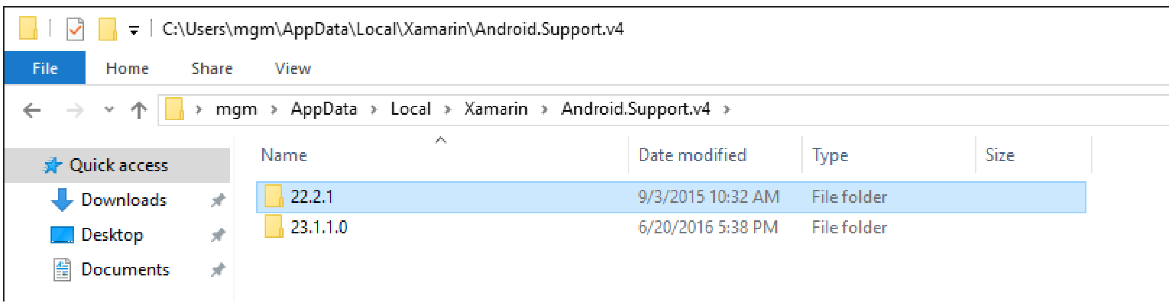
Automatic Recovery from m2repository Errors

Often, this issue can be remedied by deleting the problematic library and rebuilding according to these steps:

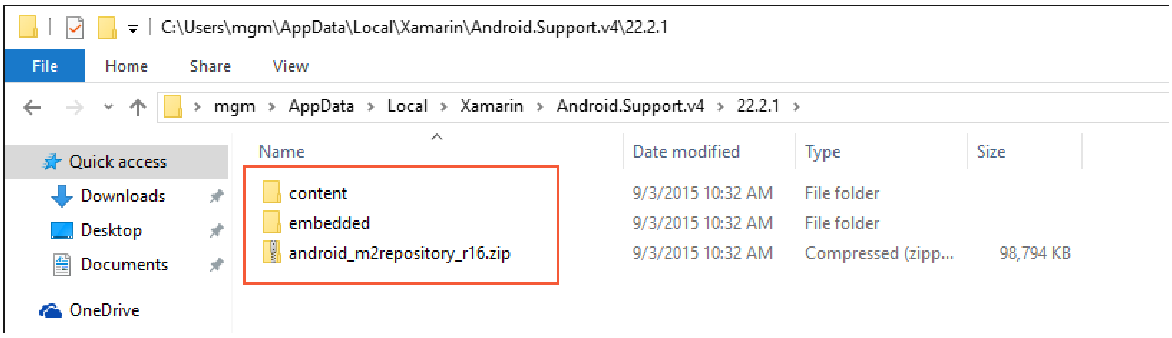
1. Navigate to the support library directory on your computer:

   - On Windows, support libraries are located at **C:\Users*username*\AppData\Local\Xamarin**.
   - On Mac OS X, support libraries are located at **/Users/*username*/.local/share/Xamarin**.

2. Locate the library and version folder corresponding to the error message. For example, the library and version folder for the above error message is located at **Android.Support.v4\22.2.1**:



3. Delete the contents of the version folder. Be sure to remove the **.zip** file as well as the **content** and **embedded** subdirectories within this folder. For the example error message shown above, the files and subdirectories shown in this screenshot (**content**, **embedded**, and **android_m2repository_r16.zip**) are to be deleted:



Note that it is important to delete the *entire* contents of this folder. Although this folder may initially contain the "missing" **android_m2repository_r16.zip** file, this file may have been partially downloaded or corrupted.

4. Rebuild the project – doing so will cause the build process to re-download the missing library.

In most cases, these steps will resolve the build error and allow you to continue. If deleting this library does not resolve the build error, you must manually download and install the **android_m2repository_r_nn_.zip** file as described in the next section.

Manually Downloading m2repository

If you have tried using the automatic recovery steps above and still have build errors, you can manually download the **android_m2repository_r_nn_.zip** file (using a web browser) and install it according to the following steps. This procedure is also useful if you do not have internet access on your development computer but you are able to download the archive using a different computer.

1. Download the **android_m2repository_r_nn_.zip** file that corresponds to the error message – links are provided in the following list (along with the corresponding MD5 hash of each link's URL):

   - android_m2repository_r33.zip – 5FB756A25962361D17BBE99C3B3FCC44
   - android_m2repository_r32.zip – F16A3455987DBAE5783F058F19F7FCDF
   - android_m2repository_r31.zip – 99A8907CE2324316E754A95E4C2D786E
   - android_m2repository_r30.zip – 05AD180B8BDC7C21D6BCB94DDE7F2C8F
   - android_m2repository_r29.zip – 2A3A8A6D6826EF6CC653030E7D695C41
   - android_m2repository_r28.zip – 17BE247580748F1EDB72E9F374AA0223
   - android_m2repository_r27.zip – C9FD4FCD69D7D12B1D9DF076B7BE4E1C
   - android_m2repository_r26.zip – 8157FC1C311BB36420C1D8992AF54A4D
   - android_m2repository_r25.zip – 0B3F1796C97C707339FB13AE8507AF50
   - android_m2repository_r24.zip – 8E3C9EC713781EDFE1EFBC5974136BEA
   - android_m2repository_r23.zip – D5BB66B3640FD9B9C6362C9DB5AB0FE7
   - android_m2repository_r22.zip – 96659D653BDE0FAEDB818170891F2BB0
   - android_m2repository_r21.zip – CD3223F2EFE068A26682B9E9C4B6FBB5
   - android_m2repository_r20.zip – 650E58DF02DB1A832386FA4A2DE46B1A
   - android_m2repository_r19.zip – 263B062D6EFAA8AEE39E9460B8A5851A
   - android_m2repository_r18.zip – 25947AD38DCB4865ABEB61522FAFDA0E
   - android_m2repository_r17.zip – 49054774F44AE5F35A6BA9D3C117EFD8

- android_m2repository_r16.zip – 0595E577D19D31708195A83087881EE6

If the **m2repository** archive is not shown in this table, you can create the download URL by prepending `https://dl-ssl.google.com/android/repository/` to the name of the **m2repository** to download. For example, use **https://dl-ssl.google.com/android/repository/android_m2repository_r10.zip** to download **android_m2repository_r10.zip**.

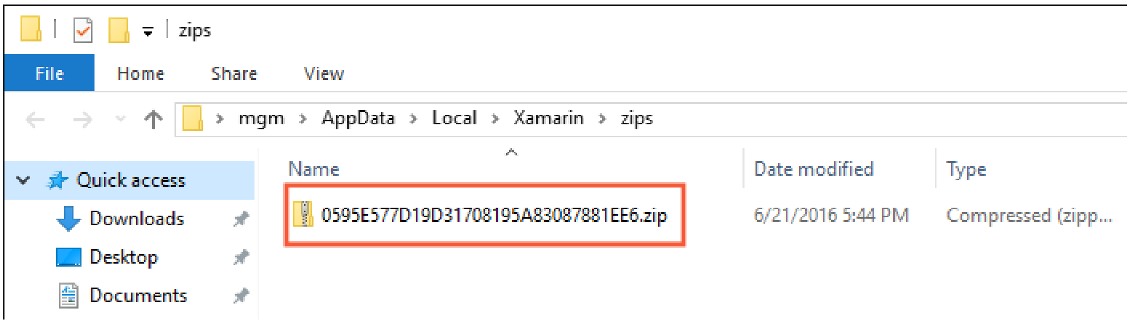
2. Rename the file to the corresponding MD5 hash of the download URL as shown in the above table. For example, if you downloaded **android_m2repository_r25.zip**, rename it to **0B3F1796C97C707339FB13AE8507AF50.zip**. If the MD5 hash for the download URL of the downloaded file is not shown in the table, you can use an online MD5 generator to convert the URL to an MD5 hash string.

3. Copy the file to the Xamarin **zips** folder:

   - On Windows, this folder is located at **C:\Users*username*\AppData\Local\Xamarin\zips**.

   - On Mac OS X, this folder is located at **/Users/*username*/.local/share/Xamarin/zips**.

   For example, the following screenshot illustrates the result when **android_m2repository_r16.zip** is downloaded and renamed to the MD5 hash of its download URL on Windows:



If this procedure does not resolve the build error, you must manually download the **android_m2repository_r_nn_.zip** file, unzip it, and install its contents as described in the next section.

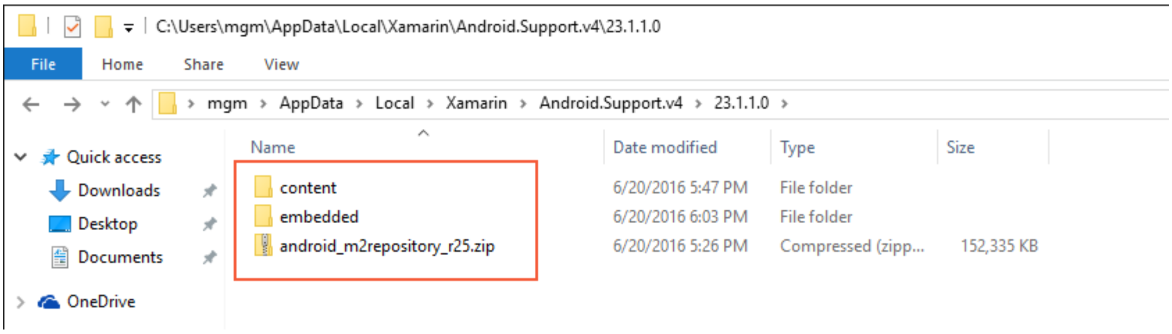
Manually Downloading and Installing m2repository Files

The fully manual process for recovering from **m2repository** errors entails downloading the **android_m2repository_r_nn_.zip** file (using a web browser), unzipping it, and copying its contents to the support library directory on your computer. In the following example, we'll recover from this error message:

```
Unzipping failed. Please download https://dl-ssl.google.com/android/repository/android_m2repository_r25.zip
and extract it to the C:\Users\mgm\AppData\Local\Xamarin\Android.Support.v4\23.1.1\content directory.
```

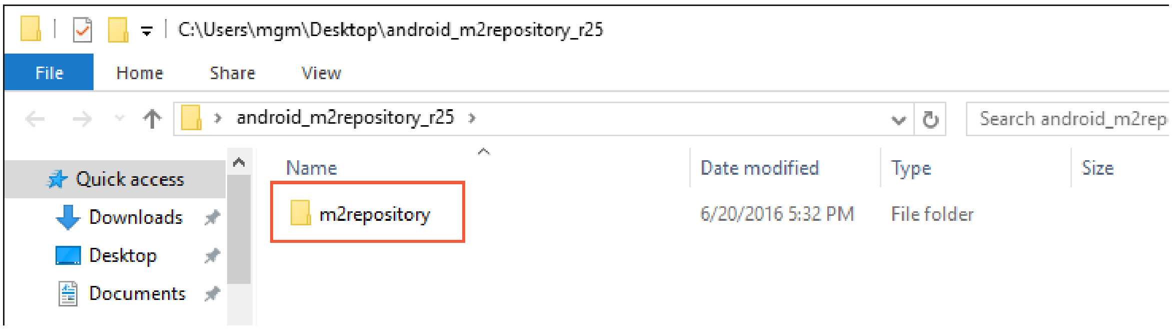
Use the following steps to download **m2repository** and install its contents:

1. Delete the contents of the library folder corresponding to the error message. For example, in the above error message you would delete the contents of **C:\Users*username*\AppData\Local\Xamarin\Android.Support.v4\23.1.1.0**. As described earlier, you must delete the entire contents of this directory:

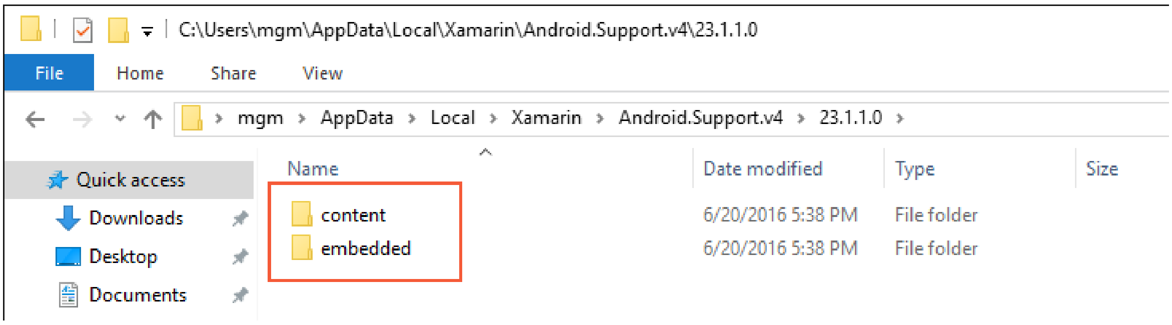


2. Download the **android_m2repository_r_nn_.zip** file from Google that corresponds to the error message (see the table in the previous section for links).

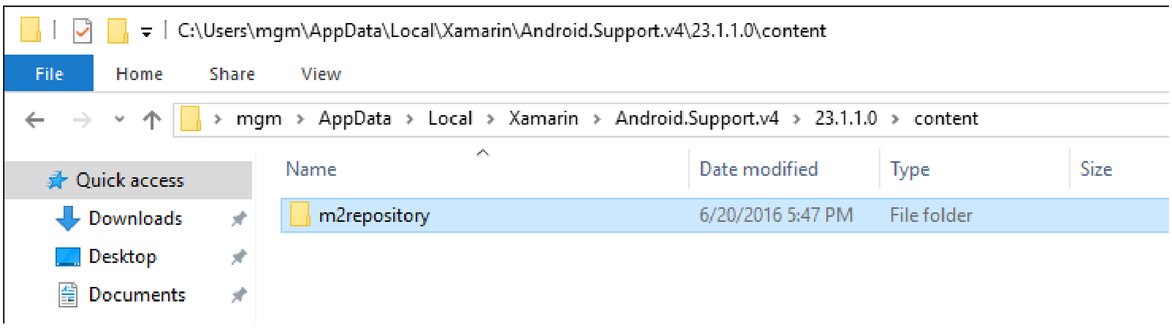
3. Extract this **.zip** archive to any location (such as the Desktop). This should create a directory that corresponds to the name of the **.zip** archive. Within this directory, you should find a subdirectory called **m2repository**:



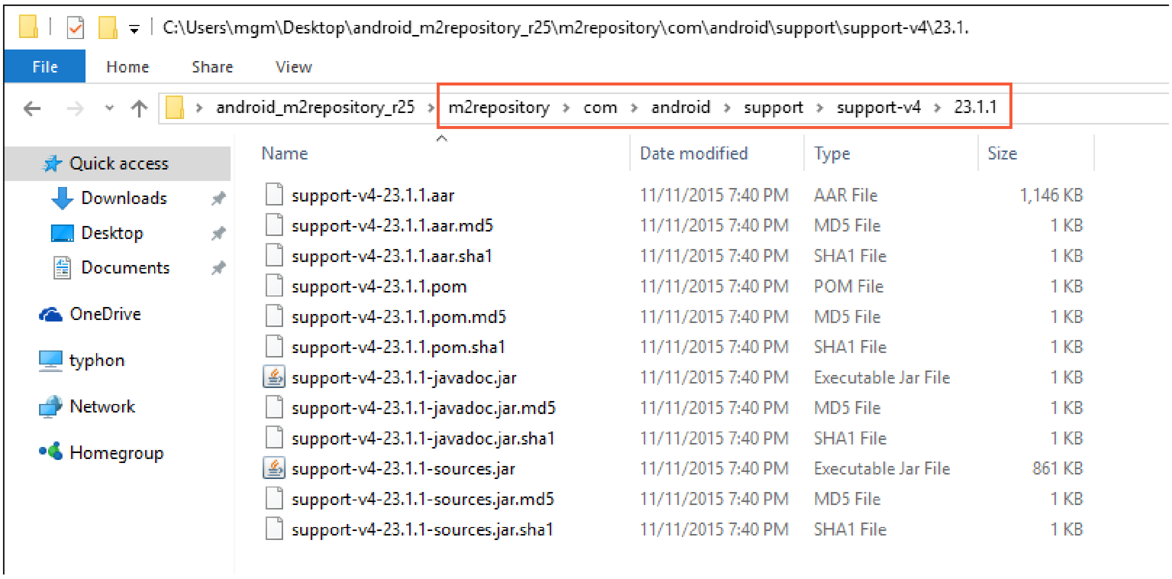
4. In the versioned library directory that you purged in step 1, re-create the **content** and **embedded** subdirectories. For example, the following screenshot illustrates **content** and **embedded** subdirectories being created in the **23.1.1.0** folder for **android_m2repository_r25.zip**:



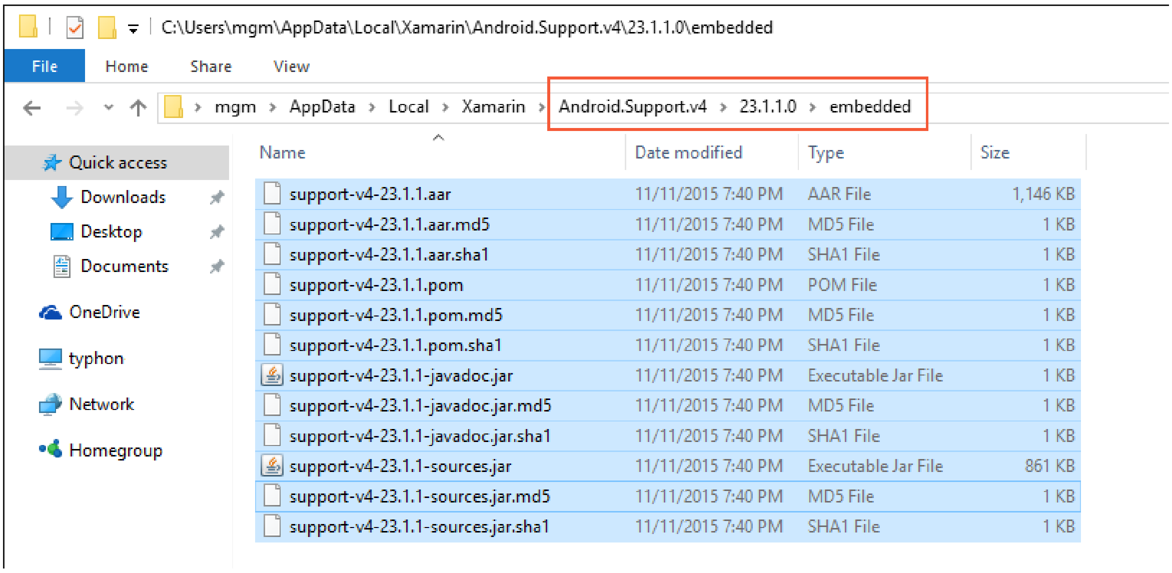
5. Copy **m2repository** from the extracted **.zip** into the **content** directory that you created in the previous step:



6. In the extracted **.zip** directory, browse to **m2repository\com\android\support\support-v4** and open the folder corresponding the version number created above (in this example, **23.1.1**):



7. Copy all of the files in this folder to the **embedded** directory created in step 4:



8. Verify that all files are copied over. The **embedded** directory should now contain files such as **.jar**, **.aar**, and **.pom**.

9. Unzip the contents of any extracted **.aar** files to the **embedded** directory. On Windows, append a **.zip** extension to the **.aar** file, open it, and copy the contents to the **embedded** directory. On macOS, unzip the **.aar** file by using the **unzip** command in the Terminal (for example, **unzip file.aar**).

At this point, you have manually installed the missing components and your project should build without errors. If not, verify that you have downloaded the **m2repository .zip** archive version that corresponds exactly to the version in the error message, and verify that you have installed its contents in the correct locations as described in the above steps.

Summary

This article explained how to recover from common errors that can take place during the automatic download and installation of dependency libraries. It described how to delete the problematic library and rebuild the project as a way to re-download and re-install the library. It described how to download the library and install it in the **zips** folder. It also described a more involved procedure for manually downloading and installing the necessary files as a way to work around issues that cannot be resolved via automatic means.

Changes to the Android SDK Tooling

10/28/2019 • 2 minutes to read • Edit Online

Changes to how the Android SDK manages the installed API levels and AVDs.

Changes to Android SDK Tooling

In recent versions of the SDK Tools for Android, Google has removed the existing AVD and SDK managers in favor of new CLI (Command Line Interface) tooling. The **android** program has been removed and the Google GUI (Graphical User Interface) managers in Visual Studio for Mac and older versions of Visual Studio Tools for Xamarin will no longer work past version 25.2.5 of Android SDK Tools. For example, attempting to use the **android** program via the command line will result in an error message like the following:

```
The "android" command is deprecated.
For manual SDK, AVD, and project management, please use Android Studio.
For command-line tools, use tools\bin\sdkmanager.bat
and tools\bin\avdmanager.bat
```

The following sections explain how to manage the Android SDK and Android Virtual Devices using Android SDK 25.3.0 and later.

UI Tools

Visual Studio and Visual Studio for Mac now provide Xamarin replacements for the discontinued Google GUI-based managers:

- To download Android SDK tools, platforms, and other components that you need for developing Xamarin.Android apps, use the Xamarin Android SDK Manager instead of the legacy Google SDK Manager.
- To create and configure Android Virtual Devices, use the Android Device Manager instead of the legacy Google Emulator Manager.

These tools are functionally equivalent to the Google GUI-based managers they replace.

CLI Tools

Alternately, you can use CLI tools to manage and update your emulators and Android SDK. The following programs now make up the command line interface for the Android SDK tools:

sdkmanager

Added In: Android SDK Tools 25.2.3 (November, 2016) and higher.

There is a new program called **sdkmanager** in the **tools/bin** folder of your Android SDK. This tool is used to maintain the Android SDK at the command line. For more information about using this tool, see sdkmanager.

avdmanager

Added In: Android SDK Tools 25.3.0 (March, 2017) and higher.

There is a new program called **avdmanager** in the **tools/bin** folder of your Android SDK. This tool is used to maintain the AVDs for the Android Emulator. For more information about using this tool, see avdmanager.

Downgrading

You can downgrade your **Android SDK Tools** version by installing a previous version of the Android SDK from the Android Developer website.

Using the old GUI

You can still use the original GUI by running the **android** program inside your **tools** folder as long as you are on **Android SDK Tools** version **25.2.5** or lower.

Related Links

- Android SDK Setup
- Android Device Manager
- Understanding Android API levels
- SDK Tools Release Notes (Google)
- sdkmanager
- avdmanager

Android Wear

7/8/2021 • 2 minutes to read • Edit Online

Android Wear is a version of Android that is designed for wearable devices such as smart watches. This section includes instructions on how to install and configure tools required for Wear development, a step-by-step walkthrough for creating your first Wear device, and a list of samples that you can refer to for creating your own Wear apps.

Getting Started

Introduces Android Wear, describes how to install and configure your computer for Wear development, and provides steps to help you create and run your first Android Wear app on an emulator or Wear device.

User Interface

Explains Android Wear-specific controls and provides links to samples that demonstrate how to use these controls.

Platform Features

Documents in this section cover features specific to Android Wear. Here you'll find a topic that describes how to create a WatchFace.

Screen Sizes

Preview and optimize your user interface for the available screen sizes.

Deployment & Testing

Explains how to deploy your Android Wear app to an Android Wear device or to Android emulator configured for Wear. It also includes debugging tips and information for how to set up a Bluetooth connection between your development computer and an Android device.

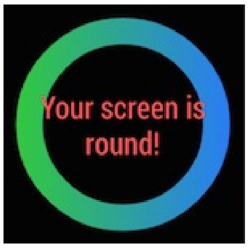
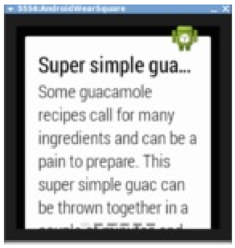
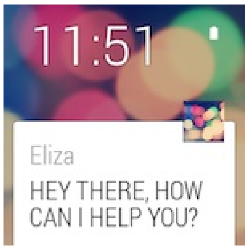
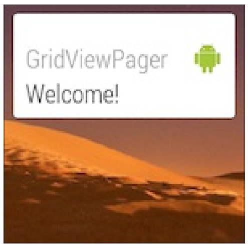
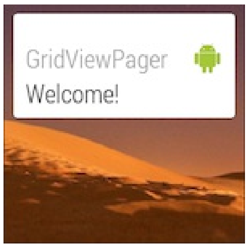
Wear APIs

The Android Developer site provides detailed information about key Wear APIs such as Wearable Activity, Intents, Authentication, Complications, Complications Rendering, Notifications, Views, and WatchFace.

Samples

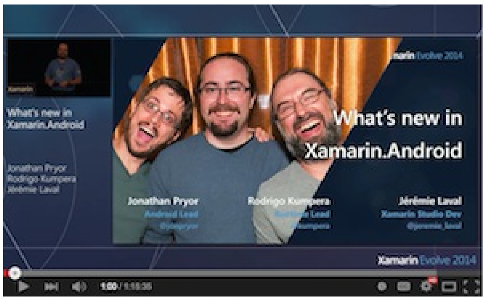
You can find a number of samples using Android Wear (or go directly to github).

SAMPLE	DESCRIPTION	SCREENSHOT

SAMPLE	DESCRIPTION	SCREENSHOT
SkeletonWear	A simple example of the basics of wearable projects, including GridViewPager and interactive notifications.	
WatchViewStub	A simple demo of the WatchViewStub control that detects screen shape and automatically loads the correct layout. See how WatchViewStub works in the **Resources/layout/main_activity.xml** layout.	
RecipeAssistant	Demonstration of Wear notification pages, in the form of recipe steps. Notifications are created in RecipeService.cs.	
ElizaChat	Fun sample of interacting with a "personal assistant" called Eliza, using Wear interactive notifications to create a conversation using canned responses.	
GridViewPager	GridViewPager implements the 2D navigation pattern, where the user swipes vertically and then horizontally to navigate through options and content.	
WatchFace	WatchFace is a custom watch face with analog-style hour, minute, and second hands. This sample demonstrates how to create a watch face service that draws the current time and handles ambient mode and visibility change events. It includes a broadcast receiver that listens for time zone changes and automatically updates the time accordingly.	

Videos

Check out these video links that discuss Xamarin.Android with Wear support:

DESCRIPTION	SCREENSHOT
Android L and So Much More – The Android L Developer Preview introduced a plethora of new APIs for developers to take advantage of, including Material Design, notifications, and new animations, to name a few.	
C# is in my Ears and in my Eyes: Google Glass and Android Wear – Wearable computing might seem like something from the future (or an Inspector Gadget episode), but many people are already embracing the future today! C# developers know this and already have the tools and skills to harness the power of wearable devices (from Evolve 2014).	
What's new in Xamarin.Android – Android L, Android Wear, Android TV, Android Auto, Material Design, and ART; what does this mean to you as a Xamarin developer? from Evolve 2014.	

Get Started with Android Wear

10/28/2019 • 2 minutes to read • Edit Online

The guides in this section introduce Android Wear, describe how to install and configure your computer for Wear development, and provide steps to help you create and run your first Android Wear app.

Introduction to Wear

Provides a basic overview of Android Wear, describes its key features, lists some of the more popular Android Wear devices, and provides links to essential Google Android Wear documentation for further reading.

Setup & Installation

Walks through the installation steps and configuration details required to prepare your computer and devices for Android Wear development.

Hello, Wear

This walkthrough provides step-by-step instructions for creating a small Android Wear project that handles button clicks and displays a click counter on the Wear device.

Introduction to Android Wear

7/8/2021 • 8 minutes to read • Edit Online

With the introduction of Google's Android Wear, you are no longer restricted to just phones and tablets when it comes to developing great Android apps. Xamarin.Android's support for Android Wear makes it possible for you to run C# code on your wrist! This introduction provides a basic overview of Android Wear, describes its key features, and offers an overview of the features available in Android Wear 2.0. It lists some of the more popular Android Wear devices, and it provides links to essential Google Android Wear documentation for further reading.

Overview

Android Wear runs on a variety of devices, including the first-generation Motorola 360, LG's G watch, and the Samsung Gear Live. A second generation, including Sony's SmartWatch 3, has also been released with additional capabilities including built-in GPS and offline music playback. For Android Wear 2.0, Google has teamed up with LG for two new watches: the LG Watch Sport and the LG Watch Style.

Xamarin.Android 5.0 and later supports Android Wear through our Android 4.4W (API 20) support and a NuGet package that adds additional Wear-specific UI controls. Xamarin.Android 5.0 and later also includes functionality for packaging your Wear apps. NuGet packages are also available for Android Wear 2.0 as described later in this guide.

Android Wear Basics

Android Wear has a user interface paradigm that differs from that of Android handheld apps. The first wave of Wear apps were designed to extend a companion handheld app in some way, but beginning with Android Wear 2.0, Wear apps can be used standalone. When you deploy a Wear app, it is packaged with a companion handheld app. Because most Wear apps depend upon a handheld companion app, they need some way to communicate with handheld apps. The following sections describe these usage scenarios and outline the essential Android Wear features.

Usage Scenarios

The first version of Android Wear was focused primarily on extending current handheld applications with enhanced notifications and syncing data between the handheld app and the wearable app. Therefore, these scenarios are relatively straightforward to implement.

Wearable Notifications

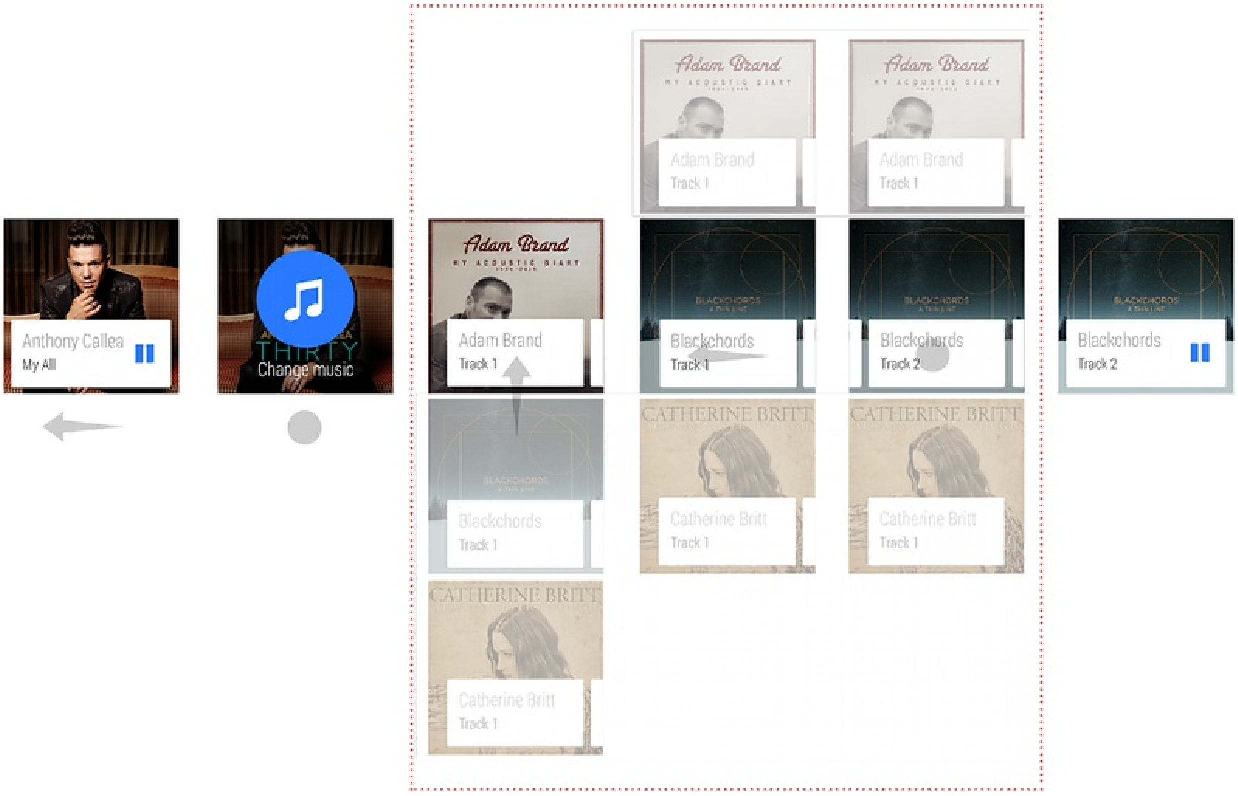
The simplest way to support Android Wear is to take advantage of the shared nature of notifications between the handheld and the wearable device. By using the support v4 notification API and the `WearableExtender` class (available in the Xamarin Android Support Library), you can tap into the native features of the platform, like inbox style cards or voice input. The RecipeAssistant sample provides example code that demonstrates how to send a list of notifications to an Android Wear device.

Companion Applications

Another strategy is to create a complete application that runs natively on the wearable device and pairs with a companion handheld app. A good example of this approach is the Quiz sample app, which demonstrates how to create a quiz that runs on a handheld device and asks quiz questions on the wearable device.

User Interface

The primary navigation pattern for Wear is a series of cards arranged vertically. Each of these cards can have associated actions that are layered out on the same row. The `GridViewPager` class provides this functionality; it adheres to the same adapter concept as `ListView`. You typically associate the `GridViewPager` with a `FragmentGridPagerAdaptor` (or `GridPagerAdaptor`) that lets you represent each row and column cells as a `Fragment`:



Wear also makes use of action buttons that consist of a big colored circle with small description text underneath it (as illustrated above). The GridViewPager sample demonstrates how to use `GridViewPager` and `GridPagerAdapter` in a Wear app.

Android Wear 2.0 adds a navigation drawer, an action drawer, and inline action buttons to the Wear user interface. For more about Android Wear 2.0 user interface elements, see the Android Anatomy topic.

Communications

Android Wear provides two different communication APIs to facilitate communications between wearable apps and companion handheld apps:

Data API – This API is similar to a synchronized data store between the wearable device and the handheld device. Android takes care of propagating changes between wearable and handheld when it is optimal to do so. When the wearable is out of range, it queues synchronization for a later time. The main entry point for this API is `WearableClass.DataApi`. For more information about this API, see the Android Syncing Data Items topic.

Message API – This API makes it possible for you to use a lower level communications path: a small payload is sent one-way without synchronization between the handheld and wearable apps. The main entry point for this API is `WearableClass.MessageApi`. For more information about this API, see the Android Sending and Receiving

Messages topic.

You can choose to register callbacks for receiving those messages via each of the API listener interfaces or, alternatively, implement a service in your app that derives from `WearableListenerService`. This service will be automatically instantiated by Android Wear. The FindMyPhone sample illustrates how to implement a `WearableListenerService`.

Deployment

Each wearable app is deployed with its own APK file embedded inside the main application APK. This packaging is handled automatically in Xamarin.Android 5.0 and later, but must be performed manually for versions of Xamarin.Android earlier than version 5.0. Working with Packaging explains deployment in more detail.

Going Further

The best way to become familiar with Android Wear is to build and test your first app. The following list provides a recommended reading order to help you get up to speed quickly:

1. Setup & Installation provides detailed instructions for installing and configuring your development environment for building Xamarin.Android Wear apps.

2. After you have installed the required packages and configured an emulator or device, see Hello, Wear for step-by-step instructions that explain how to create a small Android Wear project that handles button clicks and displays a click counter on the Wear device.

3. Deployment & Testing provides more detailed information about configuring and deploying to emulators and devices, including instructions on how to deploy your app to a Wear device via Bluetooth.

4. Working with Screen Sizes explains how to preview and optimize your user interface for the various available screen sizes on Wear devices.

5. Working with Packaging describes the steps for manually packaging Wear apps for distribution on Google Play.

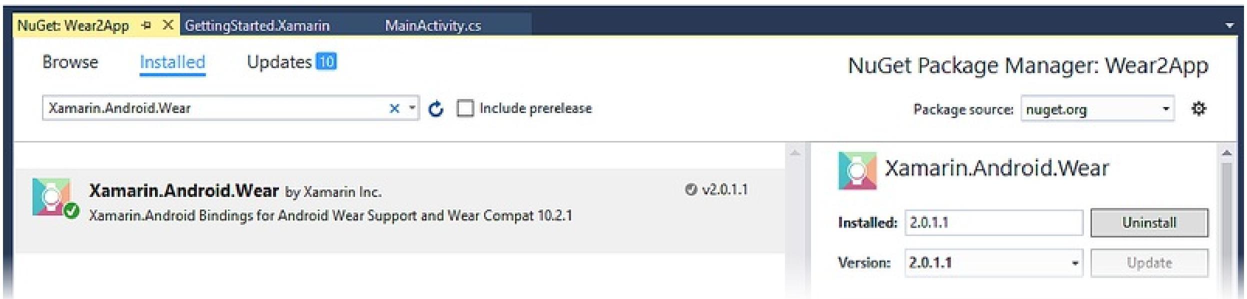
After you have created your first Wear app, you may want to try building a custom watch face for Android Wear. Creating a Watch Face provides step-by-step instructions and example code for developing a stripped down digital watch face service, followed by more code that enhances it to an analog-style watch face with extra features.

Android Wear 2.0

Android Wear 2.0 introduces a variety of new features and capabilities, such as *complications*, curved layouts, navigation and action drawers, and expanded notifications. Also, Wear 2.0 makes it possible for you to build standalone apps that work independently of handheld apps. The new *wrist gestures* capability enables one-handed interactions with your app. The following sections highlight these features and provide links to help you get started with using them in your app.

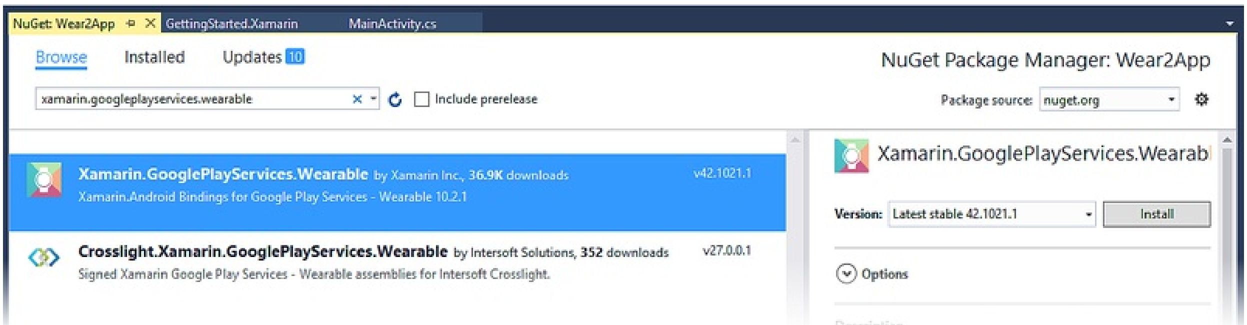
Install Wear 2.0 Packages

To build a Wear 2.0 app with Xamarin.Android, you must add the **Xamarin.Android.Wear v2.0** package to your project (click the **Browse tab**):



This NuGet package contains bindings for both the Android Support Wearable and Wear Compat libraries.

In addition to **Xamarin.Android.Wear**, we recommend that you install the **Xamarin.GooglePlayServices.Wearable** NuGet:



Key Features of Wear 2.0

Android Wear 2.0 is the biggest update to Android Wear since its initial launch in 2014. The following sections highlight the key features of Android Wear 2.0, and links are provided to help you get started using these new features in your app.

Complications

Complications are small watch face widgets that you can see at a glance without having to swipe the watch face. Complications are similar to desktop-style dashboard widgets; they display information such as the weather, battery life, calendar events, and fitness app statistics:

For more about complications, see the Android Watch Face Complications topic.

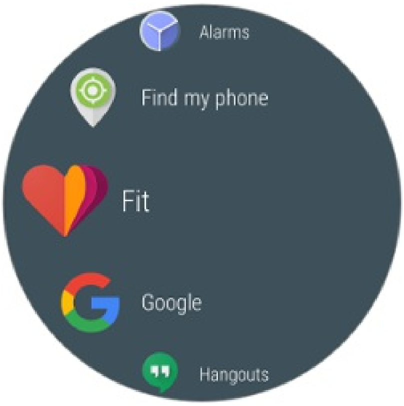
Navigation and Action Drawers

Two new drawers are included in Wear 2.0. The *navigation drawer*, which appears at the top of the screen, allows users to navigate between app views (as shown on the left below). The *action drawer*, which appears at the bottom of the screen (as shown on the right), allows users to choose from a list of actions.

For more information about these two new interactive drawers, see the Android Wear Navigation and Actions topic.

Curved Layouts

Wear 2.0 introduces new features for displaying curved layouts on round Wear devices. Specifically, the new `WearableRecyclerView` class is optimized for displaying a list of vertical items on round displays:



`WearableRecyclerView` extends the `RecyclerView` class to support curved layouts and circular scrolling gestures. For more information, see the Android WearableRecyclerView API documentation.

Standalone Apps

Android Wear 2.0 apps can work independently of handheld apps. This means that, for example, a smart watch can continue to offer full functionality even if the companion handheld device is turned off or far away from the wearable device. For more information about this feature, see the Android Standalone Apps topic.

Wrist Gestures

Wrist gestures make it possible for users to interact with your app without using the touch screen – users can respond to the app with a single hand. Two wrist gestures are supported:

- Flick wrist out
- Flick wrist in

For more information, see the Android Wrist Gestures topic.

There are many more Wear 2.0 features such as inline actions, smart reply, remote input, expanded notifications, and a new bridging mode for notifications. For more information about the new Wear 2.0 features, see the Android API Overview.

Devices

Here are some examples of the devices that can run Android Wear:

- Motorola 360
- LG G Watch

- LG G Watch R
- Samsung Gear Live
- Sony SmartWatch 3
- ASUS ZenWatch

Further Reading

Check out Google's Android Wear documentation:

- About Android Wear
- Android Wear App Design
- android.support.wearable library
- Android Wear 2.0

Summary

This introduction provided an overview of Android Wear. It outlined the basic features of Android Wear and included a overview of the features introduced in Android Wear 2.0. It provided links to essential reading to help developers get started with Xamarin.Android Wear development, and it listed examples of some of the Android Wear devices currently on the market.

Related Links

- Installation and Setup
- Getting Started

Install and setup Wear OS on Xamarin.Android

7/8/2021 • 2 minutes to read • Edit Online

This article walks through the installation steps and configuration details required to prepare your computer and devices for Android Wear development. By the end of this article, you'll have a working Xamarin.Android Wear installation integrated into Visual Studio for Mac and/or Microsoft Visual Studio, and you'll be ready to start building your first Xamarin.Android Wear application.

Requirements

The following is required to create Xamarin-based Android Wear apps:

- **Visual Studio or Visual Studio for Mac** – Visual Studio 2017 Community or later is required.
- **Xamarin.Android** – Xamarin.Android 4.17 or later must be installed and configured with either Visual Studio or Visual Studio for Mac.
- **Android SDK** - Android SDK 5.0.1 (API 21) or later must be installed via the Android SDK Manager.
- **Java Developer Kit** – Xamarin Android development requires JDK 1.8 if you are developing for API level 24 or greater (JDK 1.8 also supports API levels earlier than 24).

You can continue to use JDK 1.7 if you are developing specifically for API level 23 or earlier.

> **IMPORTANT**
>
> Xamarin.Android does not support JDK 9.

Installation

After you have installed Xamarin.Android, perform the following steps so that you're ready to build and test Android Wear apps:

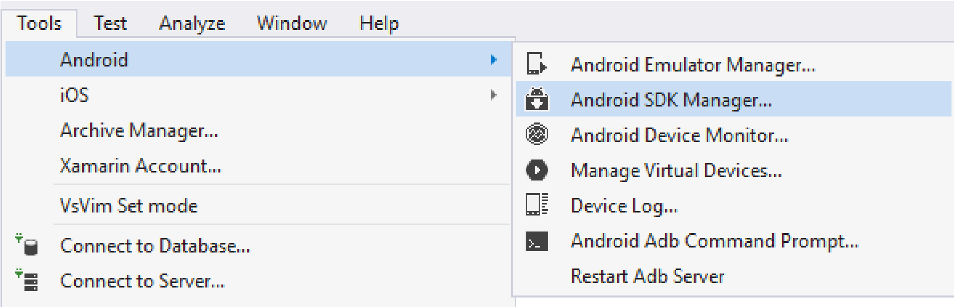
1. Install the required Android SDK and tools.
2. Configure a test device.
3. Create your first Android Wear app.

These steps are described in the following sections.

Install Android SDK and tools

Launch the **Android SDK Manager**:

- Visual Studio
- Visual Studio for Mac

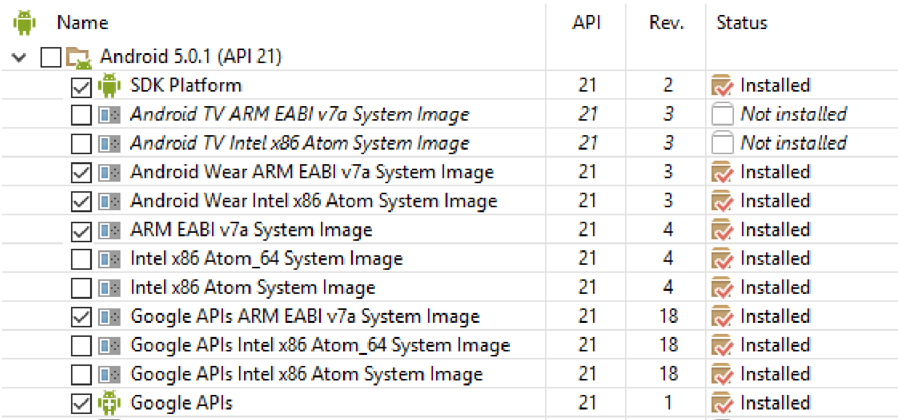


Ensure that you have the following Android SDK and tools installed:

- Android SDK Tools v 24.0.0 or higher, and
- Android 4.4W (API20), or
- Android 5.0.1 (API21) or higher.

If you do not have the latest SDK and tools installed, download the required SDK tools *and* the API bits (you may need to scroll a bit to find them – the API selection is shown below):

- Visual Studio
- Visual Studio for Mac



Name	API	Rev.	Status
Android 5.0.1 (API 21)			
SDK Platform	21	2	Installed
Android TV ARM EABI v7a System Image	*21*	*3*	*Not installed*
Android TV Intel x86 Atom System Image	*21*	*3*	*Not installed*
Android Wear ARM EABI v7a System Image	21	3	Installed
Android Wear Intel x86 Atom System Image	21	3	Installed
ARM EABI v7a System Image	21	4	Installed
Intel x86 Atom_64 System Image	21	4	Installed
Intel x86 Atom System Image	21	4	Installed
Google APIs ARM EABI v7a System Image	21	18	Installed
Google APIs Intel x86 Atom_64 System Image	21	18	Installed
Google APIs Intel x86 Atom System Image	21	18	Installed
Google APIs	21	1	Installed

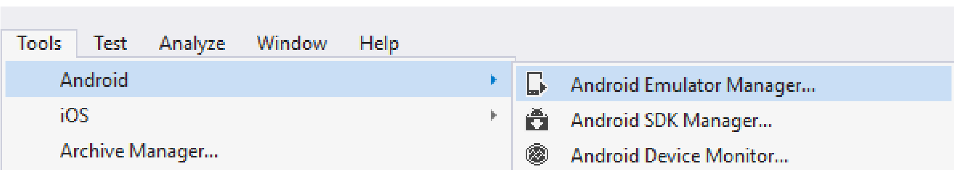
Configuration

Before you can use test your app, you must configure an Android Wear emulator or an actual Android Wear device.

Android Wear Emulator

Before you can use an Android Wear emulator, you must configure an Android Wear Android Virtual Device (AVD) using the **Google Emulator Manager**:

- Visual Studio
- Visual Studio for Mac



For more information about setting up an Android Wear emulator, see Debug Android Wear on an Emulator.

Android Wear Device

If you have an Android Wear device such as an Android Wear Smartwatch, You can debug the app on this device instead of using an emulator. For information about developing with a Wear device, see Debug on a Wear

Device.

Create Your First Android Wear App

Follow the Hello, Wear instructions to build your first watch app.

Packaging Your App

Android wear applications are always distributed with a companion Android phone app.

When you add your Android Wear application as a reference to your main Android application it is automatically assumed to be an Android Wear project and will generate all necessary XML and metadata for you. In addition, it will verify that package and version numbers match so you can easily ship your apps to Google Play.

To learn more about packaging Wear apps, see Working with Packaging.

Related Links

- SkeletonWear (sample)

Hello, Wear

7/8/2021 • 3 minutes to read • Edit Online

Create your first Android Wear app and run it on a Wear emulator or device. This walkthrough provides step-by-step instructions for creating a small Android Wear project that handles button clicks and displays a click counter on the Wear device. It explains how to debug the app using a Wear emulator or a Wear device that is connected via Bluetooth to an Android phone. It also provides a set of debugging tips for Android Wear.

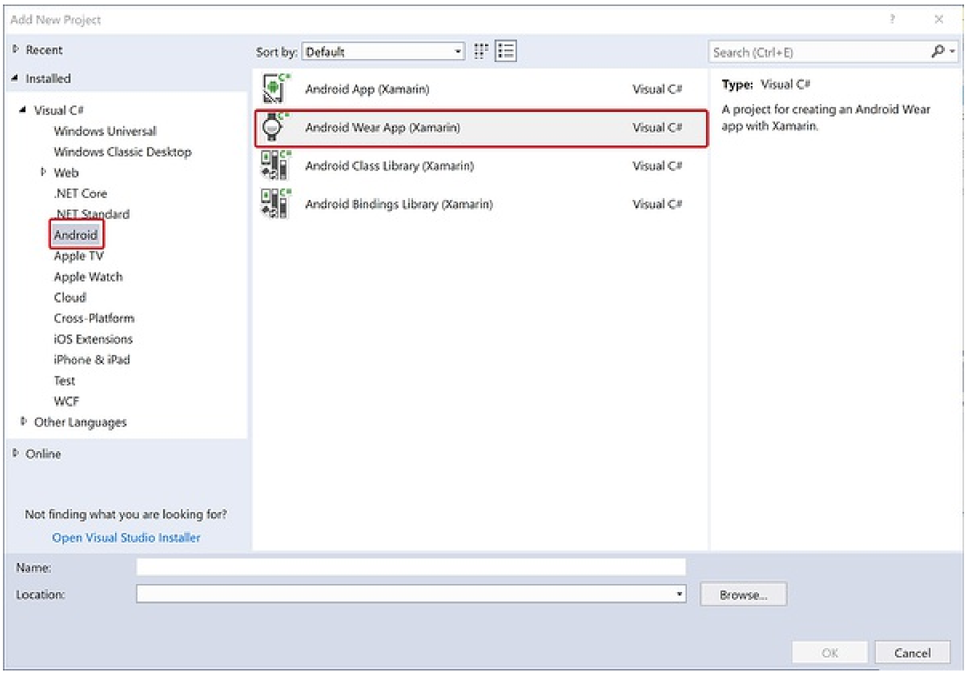
Your first Wear app

Follow these steps to create your first Xamarin.Android Wear app:

1. Create a new Android project

Create a new **Android Wear Application**:

- Visual Studio
- Visual Studio for Mac

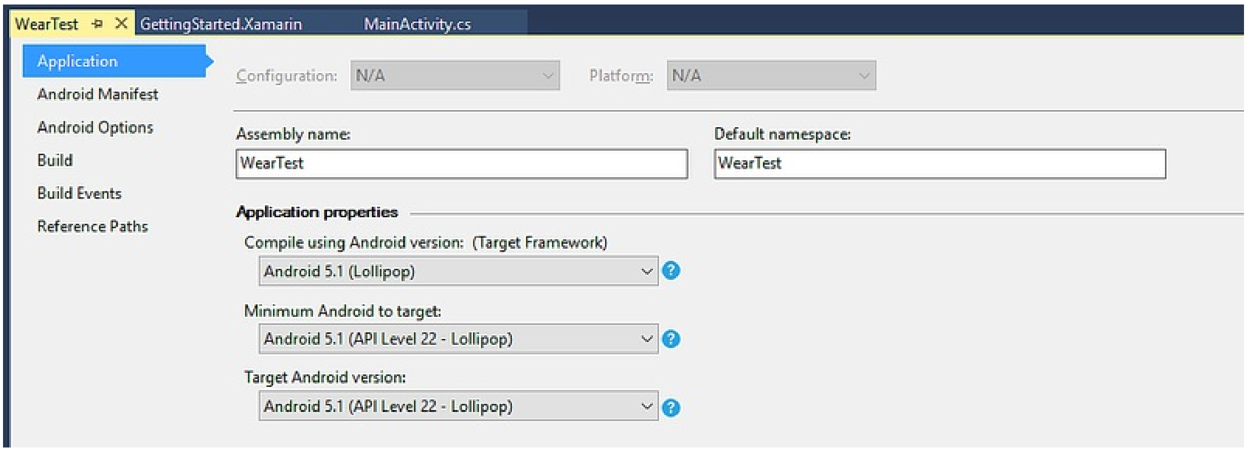


This template automatically includes the **Xamarin Android Wearable Library** NuGet (and dependencies) so you'll have access to Wear-specific widgets. If you don't see the Wear template, review the Installation and Setup guide to double-check that you have installed a supported Android SDK.

2. Choose the correct Target Framework

- Visual Studio
- Visual Studio for Mac

Ensure that **Minimum Android to target** is set to **Android 5.0 (Lollipop)** or later:



For more information on setting the target framework, see Understanding Android API Levels.

3. Edit the Main.axml layout

Configure the layout to contain a `TextView` and a `Button` for the sample:

```xml
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="match_parent">
  <ScrollView
    android:id="@+id/scroll"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:background="#000000"
    android:fillViewport="true">
    <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:orientation="vertical">
      <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="2dp"
        android:text="Main Activity"
        android:textSize="36sp"
        android:textColor="#006600" />
      <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="2dp"
        android:textColor="#cccccc"
        android:id="@+id/result" />
      <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:onClick="showNotification"
        android:text="Click Me!"
        android:id="@+id/click_button" />
    </LinearLayout>
  </ScrollView>
</FrameLayout>
```

4. Edit the MainActivity.cs source

Add the code to increment a counter and display it whenever the button is clicked:

```csharp
[Activity (Label = "WearTest", MainLauncher = true, Icon = "@drawable/icon")]
public class MainActivity : Activity
{
  int count = 1;

  protected override void OnCreate (Bundle bundle)
  {
    base.OnCreate (bundle);

    SetContentView (Resource.Layout.Main);

    Button button = FindViewById<Button> (Resource.Id.click_button);
    TextView text = FindViewById<TextView> (Resource.Id.result);

    button.Click += delegate {
      text.Text = string.Format ("{0} clicks!", count++);
    };
  }
}
```

5. Setup an Emulator or Device

The next step is set up an emulator or device to deploy and run the app. If you are not yet familiar with the process of deploying and running Xamarin.Android apps in general, see the Hello, Android Quickstart.

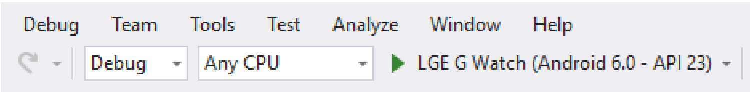
If you do not have an Android Wear device such as an Android Wear Smartwatch, You can run the app on an emulator. For information about debugging Wear apps on an emulator, see Debug Android Wear on an Emulator.

If you have an Android Wear device such as an Android Wear Smartwatch, You can run the app on the device instead of using an emulator. For more information about debugging on a Wear device, see Debug on a Wear Device.

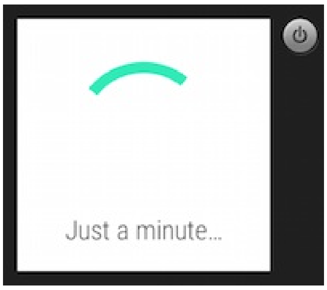
6. Run the Android Wear app

The Android Wear device should appear in the device pulldown menu. Be sure to choose the correct Android Wear device or AVD before you start debugging. After selecting the device, click the Play button to deploy the app to the emulator or device.

- Visual Studio
- Visual Studio for Mac



You may see a **Just a minute...** message (or some other interstitial screen) at first:



If you are using a watch emulator, it can take a while to start up the app. When you are using Bluetooth, it takes more time to deploy the app than it would over USB. (For example, it takes about 5 minutes to deploy this app to an LG G Watch that is Bluetooth-connected to a Nexus 5 phone.)

After the app successfully deploys, the screen of the Wear device should display a screen like the following:

Tap the **CLICK ME!** button on the face of the Wear device and see the count increment with each tap:

Next Steps

Check out the Wear samples including Android Wear apps with companion Phone apps.

When you are ready to distribute your app, see Working with Packaging.

Related Links

- Click Me App (sample)

User Interfaces for Wear OS with Xamarin.Android

11/2/2020 • 2 minutes to read • Edit Online

The following sections explain the various tools and building blocks that are used to compose user interfaces in Android Wear apps.

Controls

Explains Android Wear-specific controls and provides links to samples that demonstrate how to use these controls.

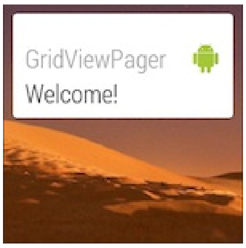
Android Wear Controls

7/8/2021 • 2 minutes to read • Edit Online

Android Wear apps can use many of the same controls already in use for regular Android apps, including `Button`, `TextView`, and image drawables. Layout controls including `ScrollView`, `LinearLayout`, and `RelativateLayout` can also be used.

This page links to the Android-Wear-specific controls from the wearable UI library available in Xamarin projects via the Wearable Support NuGet package. These controls include the following:

- **GridViewPager** – Create a two-dimensional navigation interface where the user scrolls down then across to make a selection (for more information, see GridViewPager):



Other important controls for Wear apps include:

- `BoxInsetLayout` (see working with screen sizes),
- `WatchViewStub` (see working with screen sizes),
- `CardFrame` (see Android Creating Cards),
- `CardScrollView` (see Android Creating Cards),
- `WearableListView` (see Android Create Lists).

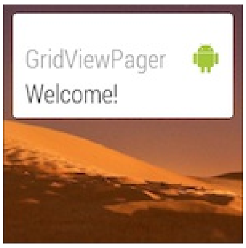
Related Links

- Android.Support.Wearable docs

GridViewPager

7/8/2021 • 2 minutes to read • Edit Online

The GridViewPager sample demonstrates how to implement the 2D picker navigation pattern for Android Wear.



First add the Xamarin Android Wear Support NuGet package to your project.

The layout XML looks like this:

```
<android.support.wearable.view.GridViewPager xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:keepScreenOn="true" />
```

Create a `GridPagerAdapter` (or subclass such as `FragmentGridPagerAdapter` to supply views to display as the user navigates.

The sample adapter shows how to implement the required methods, including overrides for `RowCount`, `GetColumnCount`, `GetBackground`, and `GetFragment`

Wire up the adapter as shown:

```
pager.Adapter = new SimpleGridPagerAdapter (this, FragmentManager);
```

Related Links

- Google's 2D Picker doc
- android.support.wearable docs
- GridViewPager (sample)

Wear OS Platform Features with Xamarin.Android

11/2/2020 • 2 minutes to read • Edit Online

Documents in this section cover features specific to Android Wear. Here you'll find a topic that describes how to create a WatchFace.

Creating a Watch Face

A step-by-step walkthrough for implementing a custom watch face service for Android Wear. Instructions are provided for building a stripped down digital watch face service, and then more code is added to create an analog-style watch face with extra features.

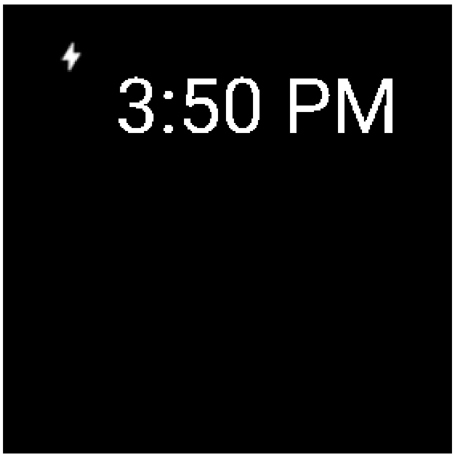
Creating a Watch Face

7/8/2021 • 14 minutes to read • Edit Online

This guide explains how to implement a custom watch face service for Android Wear 1.0. Step-by-step instructions are provided for building a stripped down digital watch face service, followed by more code to create an analog-style watch face.

Overview

In this walkthrough, a basic watch face service is created to illustrate the essentials of creating a custom Android Wear 1.0 watch face. The initial watch face service displays a simple digital watch that displays the current time in hours and minutes:



After this digital watch face is developed and tested, more code is added to upgrade it to a more sophisticated analog watch face with three hands:

Watch face services are bundled and installed as part of a Wear 1.0 app. In the following examples, `MainActivity` contains nothing more than the code from the Wear 1.0 app template so that the watch face service can be packaged and deployed to the smart watch as part of the app. In effect, this app will serve purely as a vehicle for getting the watch face service loaded into the Wear 1.0 device (or emulator) for debugging and testing.

Requirements

To implement a watch face service, the following is required:

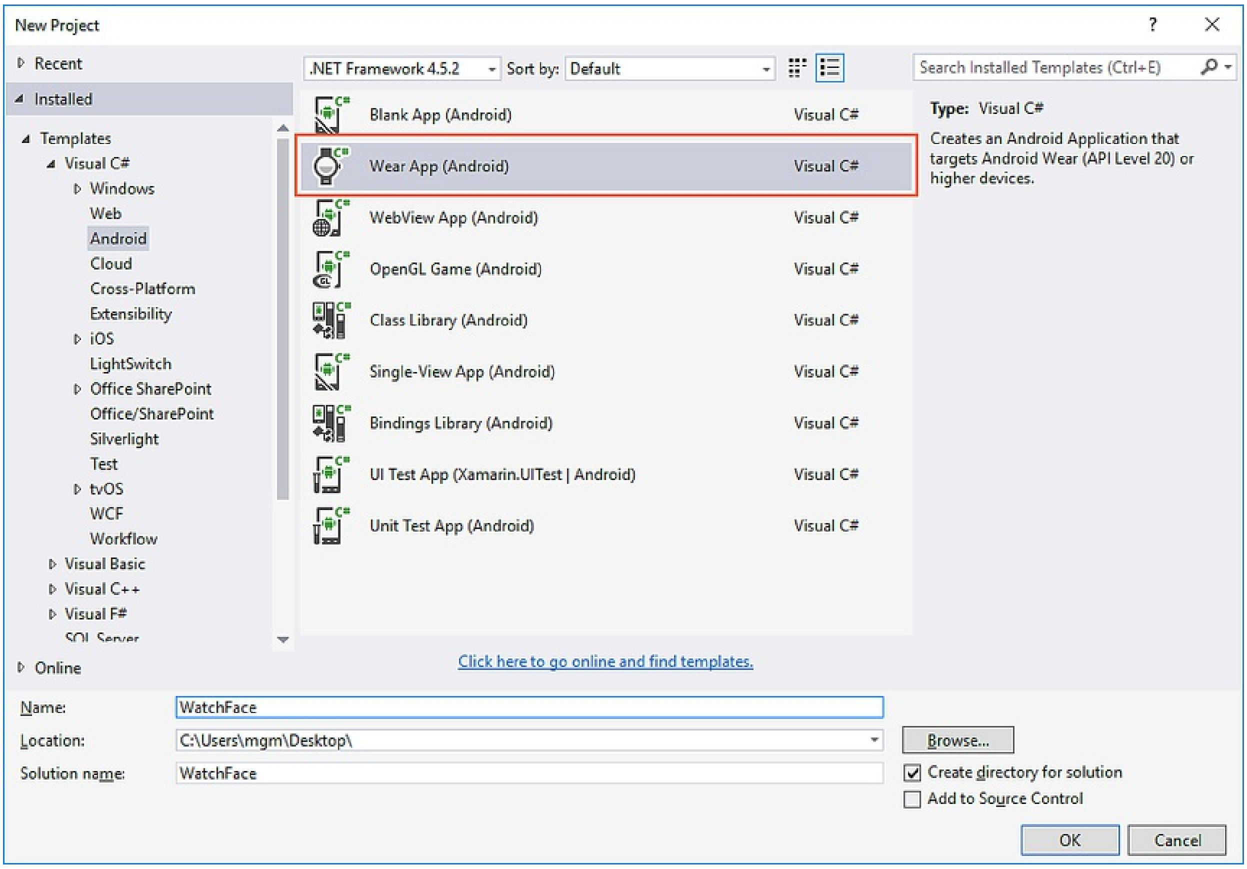
- Android 5.0 (API level 21) or higher on the Wear device or emulator.
- The Xamarin Android Wear Support Libraries must be added to the Xamarin.Android project.

Although Android 5.0 is the minimum API level for implementing a watch face service, Android 5.1 or later is recommended. Android Wear devices running Android 5.1 (API 22) or higher allow Wear apps to control what's displayed on the screen while the device is in low-power *ambient* mode. When the device leaves low-power *ambient* mode, it is in *interactive* mode. For more about these modes, see Keeping Your App Visible.

Start an App Project

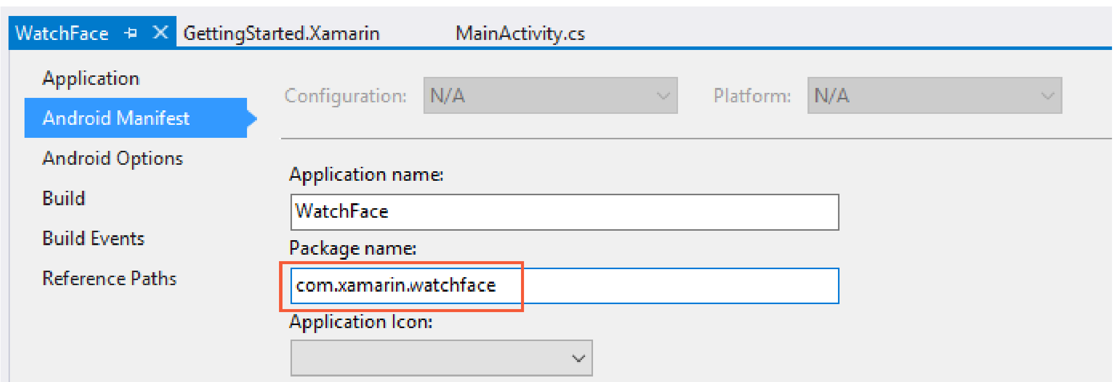
Create a new Android Wear 1.0 project called **WatchFace** (for more information about creating new Xamarin.Android projects, see Hello, Android):

- Visual Studio
- Visual Studio for Mac



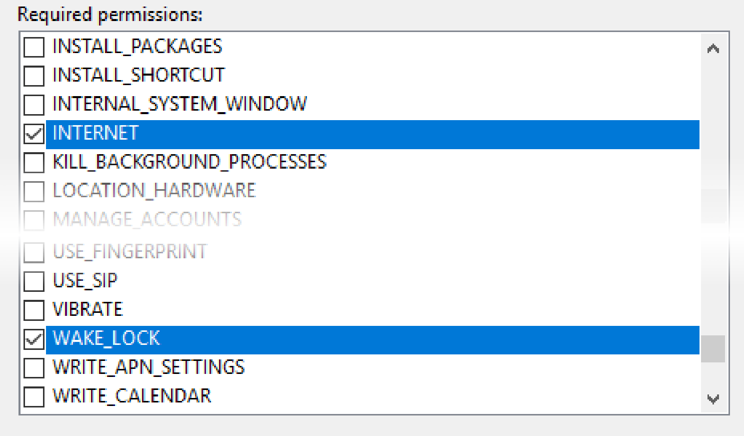
Set the package name to `com.xamarin.watchface`:

- Visual Studio
- Visual Studio for Mac



- Visual Studio
- Visual Studio for Mac

In addition, scroll down and enable the **INTERNET** and **WAKE_LOCK** permissions:

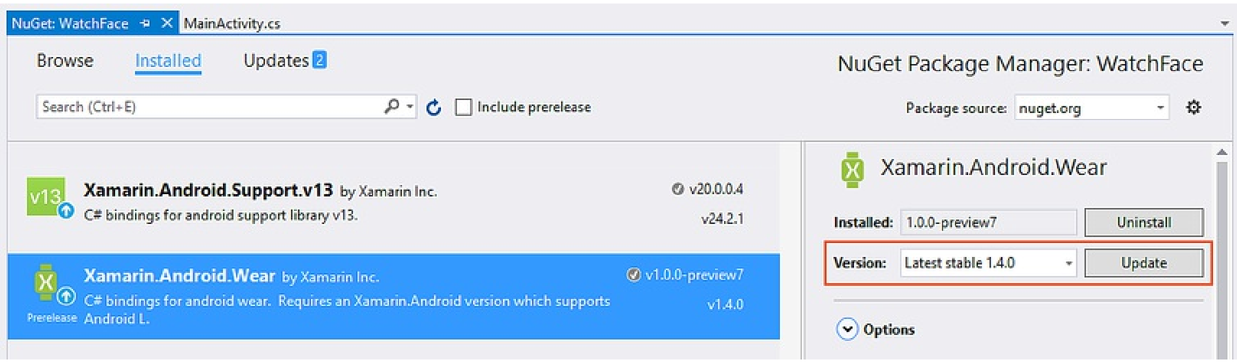


Next, download preview.png – this will be added to the **drawables** folder later in this walkthrough.

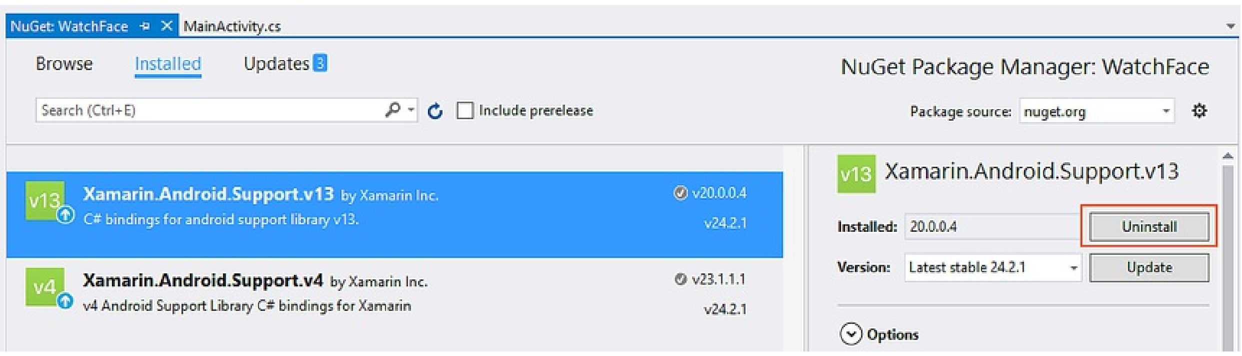
Add the Xamarin.Android Wear Package

- Visual Studio
- Visual Studio for Mac

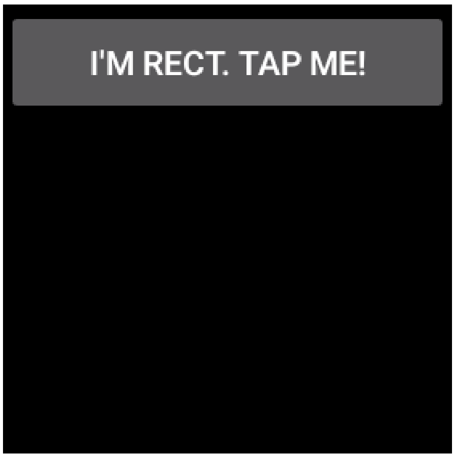
Start the NuGet Package Manager (in Visual Studio, right-click **References** in the **Solution Explorer** and select **Manage NuGet Packages** ...). Update the project to the latest stable version of **Xamarin.Android.Wear**:



Next, if **Xamarin.Android.Support.v13** is installed, uninstall it:



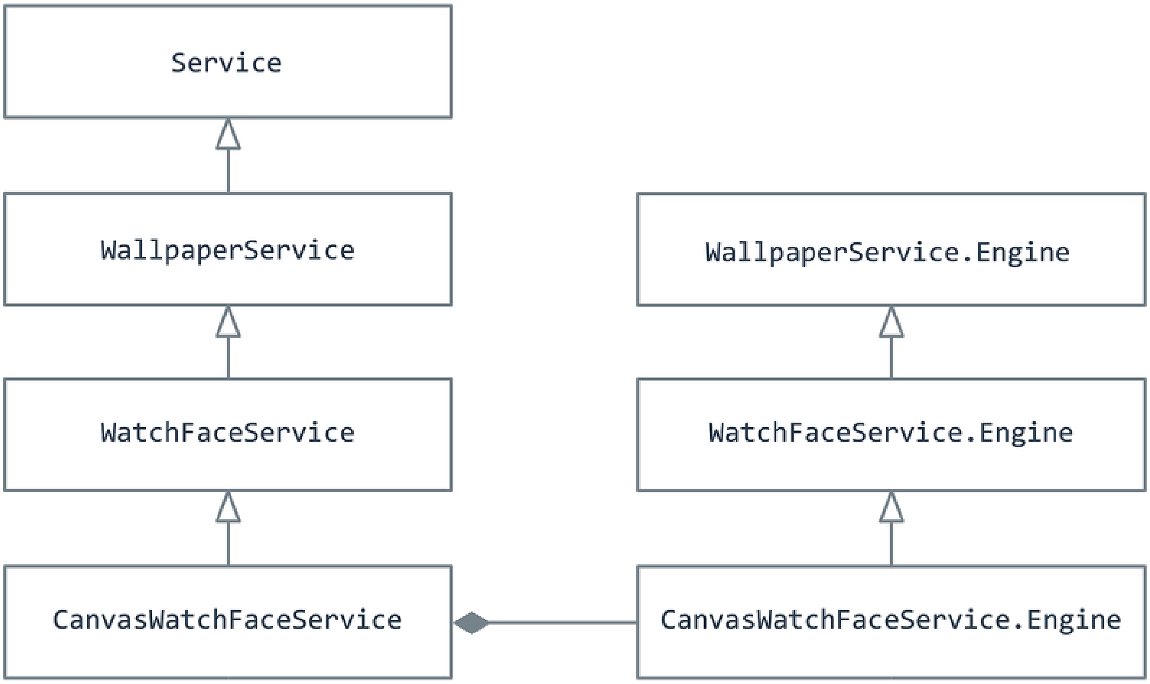
Build and run the app on a Wear device or emulator (for more information about how to do this, see the Getting Started guide). You should see the following app screen on the Wear device:



At this point, the basic Wear app does not have watch face functionality because it does not yet provide a watch face service implementation. This service will be added next.

CanvasWatchFaceService

Android Wear implements watch faces via the `CanvasWatchFaceService` class. `CanvasWatchFaceService` is derived from `WatchFaceService`, which itself is derived from `WallpaperService` as shown in the following diagram:



`CanvasWatchFaceService` includes a nested `CanvasWatchFaceService.Engine`; it instantiates a `CanvasWatchFaceService.Engine` object that does the actual work of drawing the watch face. `CanvasWatchFaceService.Engine` is derived from `WallpaperService.Engine` as shown in the above diagram.

Not shown in this diagram is a `Canvas` that `CanvasWatchFaceService` uses for drawing the watch face – this `Canvas` is passed in via the `OnDraw` method as described below.

In the following sections, a custom watch face service will be created by following these steps:

1. Define a class called `MyWatchFaceService` that is derived from `CanvasWatchFaceService`.

2. Within `MyWatchFaceService`, create a nested class called `MyWatchFaceEngine` that is derived from `CanvasWatchFaceService.Engine`.

3. In `MyWatchFaceService`, implement a `CreateEngine` method that instantiates `MyWatchFaceEngine` and returns it.

4. In `MyWatchFaceEngine`, implement the `OnCreate` method to create the watch face style and perform any other initialization tasks.

5. Implement the `OnDraw` method of `MyWatchFaceEngine`. This method is called whenever the watch face needs to be redrawn (i.e. *invalidated*). `OnDraw` is the method that draws (and redraws) watch face elements such as hour, minute, and second hands.

6. Implement the `OnTimeTick` method of `MyWatchFaceEngine`. `OnTimeTick` is called at least once per minute (in both ambient and interactive modes) or when the date/time has changed.

For more information about `CanvasWatchFaceService`, see the Android CanvasWatchFaceService API documentation. Similarly, CanvasWatchFaceService.Engine explains the actual implementation of the watch face.

Add the CanvasWatchFaceService

- Visual Studio
- Visual Studio for Mac

Add a new file called **MyWatchFaceService.cs** (in Visual Studio, right-click **WatchFace** in the **Solution Explorer**, click **Add > New Item...**, and select **Class**).

Replace the contents of this file with the following code:

```
using System;
using Android.Views;
using Android.Support.Wearable.Watchface;
using Android.Service.Wallpaper;
using Android.Graphics;

namespace WatchFace
{
    class MyWatchFaceService : CanvasWatchFaceService
    {
        public override WallpaperService.Engine OnCreateEngine()
        {
            return new MyWatchFaceEngine(this);
        }

        public class MyWatchFaceEngine : CanvasWatchFaceService.Engine
        {
            CanvasWatchFaceService owner;
            public MyWatchFaceEngine (CanvasWatchFaceService owner) : base(owner)
            {
                this.owner = owner;
            }
        }
    }
}
```

`MyWatchFaceService` (derived from `CanvasWatchFaceService`) is the "main program" of the watch face. `MyWatchFaceService` implements only one method, `OnCreateEngine`, which instantiates and returns a `MyWatchFaceEngine` object (`MyWatchFaceEngine` is derived from `CanvasWatchFaceService.Engine`). The instantiated `MyWatchFaceEngine` object must be returned as a `WallpaperService.Engine`. The encapsulating `MyWatchFaceService` object is passed into the constructor.

`MyWatchFaceEngine` is the actual watch face implementation – it contains the code that draws the watch face. It also handles system events such as screen changes (ambient/interactive modes, screen turning off, etc.).

Implement the Engine OnCreate method

The `OnCreate` method initializes the watch face. Add the following field to `MyWatchFaceEngine`:

```
Paint hoursPaint;
```

This `Paint` object will be used to draw the current time on the watch face. Next, add the following method to `MyWatchFaceEngine`:

```
public override void OnCreate(ISurfaceHolder holder)
{
    base.OnCreate (holder);

    SetWatchFaceStyle (new WatchFaceStyle.Builder(owner)
        .SetCardPeekMode (WatchFaceStyle.PeekModeShort)
        .SetBackgroundVisibility (WatchFaceStyle.BackgroundVisibilityInterruptive)
        .SetShowSystemUiTime (false)
        .Build ());

    hoursPaint = new Paint();
    hoursPaint.Color = Color.White;
    hoursPaint.TextSize = 48f;
}
```

`OnCreate` is called shortly after `MyWatchFaceEngine` is started. It sets up the `WatchFaceStyle` (which controls how the Wear device interacts with the user) and instantiates the `Paint` object that will be used to display the time.

The call to `SetWatchFaceStyle` does the following:

1. Sets *peek mode* to `PeekModeShort`, which causes notifications to appear as small "peek" cards on the display.
2. Sets the background visibility to `Interruptive`, which causes the background of a peek card to be shown only briefly if it represents an interruptive notification.
3. Disables the default system UI time from being drawn on the watch face so that the custom watch face can display the time instead.

For more information about these and other watch face style options, see the Android WatchFaceStyle.Builder API documentation.

After `SetWatchFaceStyle` completes, `OnCreate` instantiates the `Paint` object (`hoursPaint`) and sets its color to white and its text size to 48 pixels (TextSize must be specified in pixels).

Implement the Engine OnDraw method

The `OnDraw` method is perhaps the most important `CanvasWatchFaceService.Engine` method – it is the method that actually draws watch face elements such as digits and clock face hands. In the following example, it draws a time string on the watch face. Add the following method to `MyWatchFaceEngine`:

```
public override void OnDraw (Canvas canvas, Rect frame)
{
    var str = DateTime.Now.ToString ("h:mm tt");
    canvas.DrawText (str,
        (float)(frame.Left + 70),
        (float)(frame.Top  + 80), hoursPaint);
}
```

When Android calls `OnDraw`, it passes in a `Canvas` instance and the bounds in which the face can be drawn. In the above code example, `DateTime` is used to calculate the current time in hours and minutes (in 12-hour format). The resulting time string is drawn on the canvas by using the `Canvas.DrawText` method. The string will appear 70 pixels over from the left edge and 80 pixels down from the top edge.

For more information about the `OnDraw` method, see the Android onDraw API documentation.

Implement the Engine OnTimeTick method

Android periodically calls the `OnTimeTick` method to update the time shown by the watch face. It is called at least once per minute (in both ambient and interactive modes), or when the date/time or timezone have changed. Add the following method to `MyWatchFaceEngine`:

```
public override void OnTimeTick()
{
    Invalidate();
}
```

This implementation of `OnTimeTick` simply calls `Invalidate`. The `Invalidate` method schedules `OnDraw` to redraw the watch face.

For more information about the `OnTimeTick` method, see the Android onTimeTick API documentation.

Register the CanvasWatchFaceService

`MyWatchFaceService` must be registered in the **AndroidManifest.xml** of the associated Wear app. To do this, add the following XML to the `<application>` section:

```
<service
    android:name="watchface.MyWatchFaceService"
    android:label="Xamarin Sample"
    android:allowEmbedded="true"
    android:taskAffinity=""
    android:permission="android.permission.BIND_WALLPAPER">
    <meta-data
        android:name="android.service.wallpaper"
        android:resource="@xml/watch_face" />
    <meta-data
        android:name="com.google.android.wearable.watchface.preview"
        android:resource="@drawable/preview" />
    <intent-filter>
        <action android:name="android.service.wallpaper.WallpaperService" />
        <category android:name="com.google.android.wearable.watchface.category.WATCH_FACE" />
    </intent-filter>
</service>
```

This XML does the following:

1. Sets the `android.permission.BIND_WALLPAPER` permission. This permission gives the watch face service permission to change the system wallpaper on the device. Note that this permission must be set in the `<service>` section rather than in the outer `<application>` section.

2. Defines a `watch_face` resource. This resource is a short XML file that declares a `wallpaper` resource (this file will be created in the next section).

3. Declares a drawable image called `preview` that will be displayed by the watch picker selection screen.

4. Includes an `intent-filter` to let Android know that `MyWatchFaceService` will be displaying a watch face.

That completes the code for the basic `WatchFace` example. The next step is to add the necessary resources.

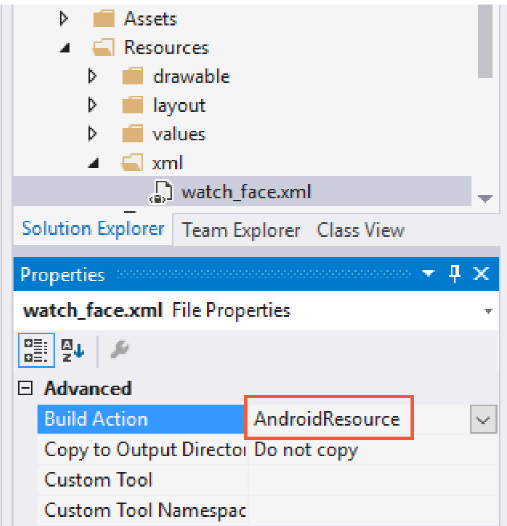
Add resource files

Before you can run the watch service, you must add the **watch_face** resource and the preview image. First, create a new XML file at **Resources/xml/watch_face.xml** and replace its contents with the following XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<wallpaper xmlns:android="http://schemas.android.com/apk/res/android" />
```

Set this file's build action to **AndroidResource**:

- Visual Studio
- Visual Studio for Mac



This resource file defines a simple `wallpaper` element that will be used for the watch face.

If you have not yet done so, download preview.png. Install it at **Resources/drawable/preview.png**. Be sure to add this file to the `WatchFace` project. This preview image is displayed to the user in the watch face picker on the Wear device. To create a preview image for your own watch face, you can take a screenshot of the watch face while it is running. (For more about getting screenshots from Wear devices, see Taking screenshots).

Try it!

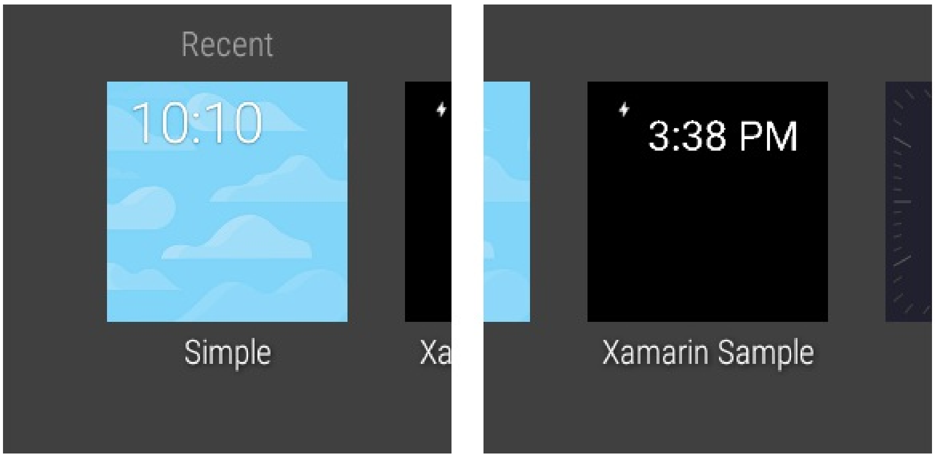
Build and deploy the app to the Wear device. You should see the Wear app screen appear as before. Do the following to enable the new watch face:

1. Swipe to the right until you see the background of the watch screen.

2. Touch and hold anywhere on the background of the screen for two seconds.

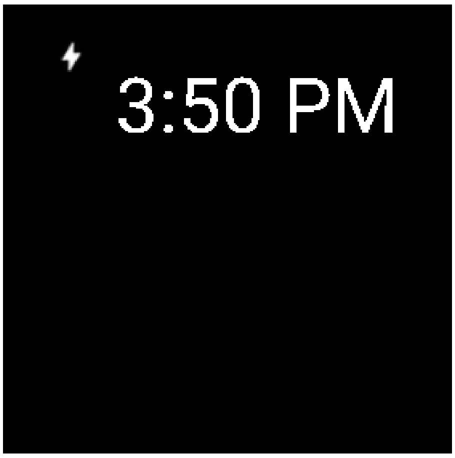
3. Swipe from left to right to browse through the various watch faces.

4. Select the **Xamarin Sample** watch face (shown on the right):



5. Tap the **Xamarin Sample** watch face to select it.

This changes the watch face of the Wear device to use the custom watch face service implemented so far:



This is a relatively crude watch face because the app implementation is so minimal (for example, it doesn't include a watch face background and it doesn't call `Paint` anti-alias methods to improve the appearance). However, it does implement the bare-bones functionality that is required to create a custom watch face.

In the next section, this watch face will be upgraded to a more sophisticated implementation.

Upgrading the watch face

In the remainder of this walkthrough, `MyWatchFaceService` is upgraded to display an analog-style watch face and it is extended to support more features. The following capabilities will be added to create the upgraded watch face:

1. Indicates the time with analog hour, minute, and second hands.

2. Reacts to changes in visibility.

3. Responds to changes between ambient mode and interactive mode.

4. Reads the properties of the underlying Wear device.

5. Automatically updates the time when a time zone change takes place.

Before implementing the code changes below, download drawable.zip, unzip it, and move the unzipped .png files to **Resources/drawable** (overwrite the previous **preview.png**). Add the new .png files to the `WatchFace` project.

Update Engine features

The next step is upgrade **MyWatchFaceService.cs** to an implementation that draws an analog watch face and supports new features. Replace the contents of **MyWatchFaceService.cs** with the analog version of the watch face code in MyWatchFaceService.cs (you can cut and paste this source into the existing **MyWatchFaceService.cs**).

This version of **MyWatchFaceService.cs** adds more code to the existing methods and includes additional overridden methods to add more functionality. The following sections provide a guided tour of the source code.

OnCreate

The updated **OnCreate** method configures the watch face style as before, but it includes some additional steps:

1. Sets the background image to the **xamarin_background** resource that resides in **Resources/drawable-hdpi/xamarin_background.png**.

2. Initializes `Paint` objects for drawing the hour hand, minute hand, and second hand.

3. Initializes a `Paint` object for drawing the hour ticks around the edge of the watch face.

4. Creates a timer that calls the `Invalidate` (redraw) method so that the second hand will be redrawn every second. Note that this timer is necessary because `OnTimeTick` calls `Invalidate` only once every minute.

This example includes only one **xamarin_background.png** image; however, you may want to create a different background image for each screen density that your custom watch face will support.

OnDraw

The updated **OnDraw** method draws an analog-style watch face using the following steps:

1. Gets the current time, which is now maintained in a `time` object.

2. Determines the bounds of the drawing surface and its center.

3. Draws the background, scaled to fit the device when the background is drawn.

4. Draws twelve *ticks* around the face of the clock (corresponding to the hours on the clock face).

5. Calculates the angle, rotation, and length for each watch hand.

6. Draws each hand on the watch surface. Note that the second hand is not drawn if the watch is in ambient mode.

OnPropertiesChanged

This method is called to inform `MyWatchFaceEngine` about the properties of the Wear device (such as low-bit ambient mode and burn-in protection). In `MyWatchFaceEngine`, this method only checks for low bit ambient mode (in low bit ambient mode, the screen supports fewer bits for each color).

For more information about this method, see the Android onPropertiesChanged API documentation.

OnAmbientModeChanged

This method is called when the Wear device enters or exits ambient mode. In the `MyWatchFaceEngine` implementation, the watch face disables anti-aliasing when it is in ambient mode.

For more information about this method, see the Android onAmbientModeChanged API documentation.

OnVisibilityChanged

This method is called whenever the watch becomes visible or hidden. In `MyWatchFaceEngine`, this method registers/unregisters the time zone receiver (described below) according to the visibility state.

For more information about this method, see the Android onVisibilityChanged API documentation.

Time zone feature

The new **MyWatchFaceService.cs** also includes functionality to update the current time whenever the time zone changes (such as while traveling across time zones). Near the end of **MyWatchFaceService.cs**, a time zone change `BroadcastReceiver` is defined that handles timezone-changed Intent objects:

```
public class TimeZoneReceiver: BroadcastReceiver
{
    public Action<Intent> Receive { get; set; }
    public override void OnReceive (Context context, Intent intent)
    {
        if (Receive != null)
            Receive (intent);
    }
}
```

The `RegisterTimezoneReceiver` and `UnregisterTimezoneReceiver` methods are called by the `OnVisibilityChanged` method. `UnregisterTimezoneReceiver` is called when the visibility state of the watch face is changed to hidden. When the watch face is visible again, `RegisterTimezoneReceiver` is called (see the `OnVisibilityChanged` method).

The engine `RegisterTimezoneReceiver` method declares a handler for this time zone receiver's `Receive` event; this handler updates the `time` object with the new time whenever a time zone is crossed:

```
timeZoneReceiver = new TimeZoneReceiver ();
timeZoneReceiver.Receive = (intent) => {
    time.Clear (intent.GetStringExtra ("time-zone"));
    time.SetToNow ();
};
```

An intent filter is created and registered for the time zone receiver:

```
IntentFilter filter = new IntentFilter(Intent.ActionTimezoneChanged);
Application.Context.RegisterReceiver (timeZoneReceiver, filter);
```

The `UnregisterTimezoneReceiver` method unregisters the time zone receiver:

```
Application.Context.UnregisterReceiver (timeZoneReceiver);
```

Run the improved watch face

Build and deploy the app to the Wear device again. Select the watch face from the watch face picker as before. The preview in the watch picker is shown on the left, and the new watch face is shown on the right:

In this screenshot, the second hand is moving once per second. When you run this code on a Wear device, the second hand disappears when the watch enters ambient mode.

Summary

In this walkthrough, a custom Android Wear 1.0 watchface was implemented and tested. The `CanvasWatchFaceService` and `CanvasWatchFaceService.Engine` classes were introduced, and the essential methods of the engine class were implemented to create a simple digital watch face. This implementation was updated with more functionality to create an analog watch face, and additional methods were implemented to handle changes in visibility, ambient mode, and differences in device properties. Finally, a time zone broadcast receiver was implemented so that the watch automatically updates the time when a time zone is crossed.

Related Links

- Creating Watch Faces
- WatchFace sample
- WatchFaceService.Engine

Working with Screen Sizes

7/8/2021 • 2 minutes to read • Edit Online

Android Wear devices can have either a rectangular or a round display, which can also be different sizes.

Identifying Screen Type

The Wear support library provides some controls that help you detect and adapt to different screen shapes, such as `WatchViewStub` and `BoxInsetLayout`.

Be aware that some of the other support library controls (such as `GridViewPager`) *automatically* detect screen shape themselves and shouldn't be added as children of the controls described below.

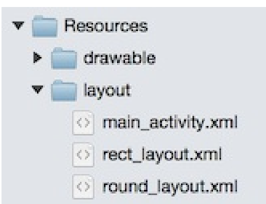
WatchViewStub

See the WatchViewStub sample to see how to detect screen type and display a different layout for each type.

The main layout file contains a `android.support.wearable.view.WatchViewStub` which references different layouts for rectangular and round screens using the `app:rectLayout` and `app:roundLayout` attributes:

```
<android.support.wearable.view.WatchViewStub
    xmlns:app="http://schemas.android.com/apk/res-auto"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:id="@+id/stub"
  app:rectLayout="@layout/rect_layout"
  app:roundLayout="@layout/round_layout" />
```

The solution contains different layouts for each style which will be selected at run-time:



BoxInsetLayout

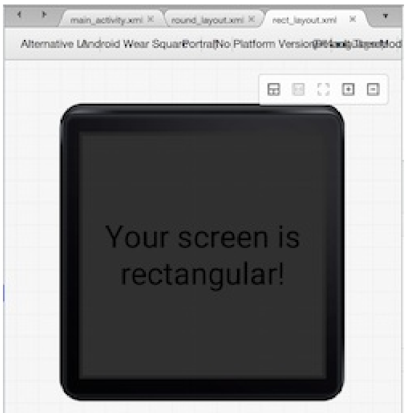
Rather than build different layouts for each screen type, you can also create a single view that adapts to rectangular or round screens.

This Google example shows how to use the `BoxInsetLayout` to use the same layout on both rectangular and round screens.

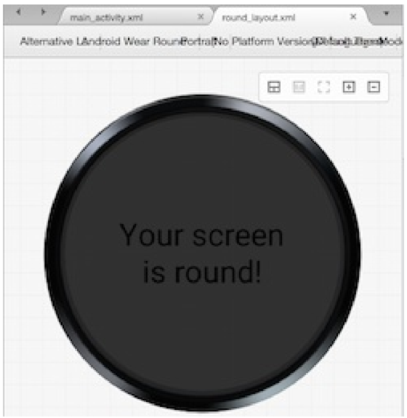
Wear UI Designer

The Xamarin Android Designer supports both rectangular and round screens:

The design surface in rectangular style is shown here:



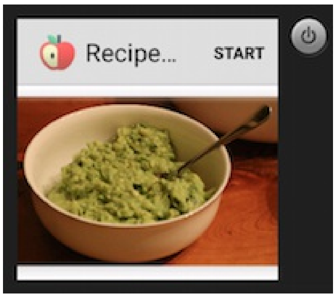
The design surface in round style is shown here:



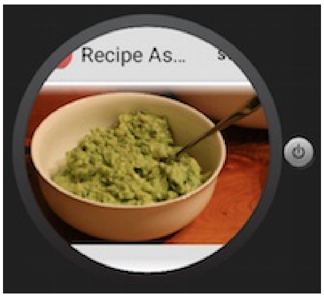
Wear Simulator

The **Google Emulator Manager** contains device definitions for both screen types. You can create rectangular and round emulators to test your app.

The emulator will render like this for a rectangular screen:



It will render like this for a round screen:



Video

Fullscreen apps for Android Wear from developers.google.com.

Deployment and Testing of Wear OS Apps

11/2/2020 • 2 minutes to read • Edit Online

This section explains how to test your Android Wear app on an Android Wear device (or on an Android emulator configured for Wear). It also includes debugging tips and information for how to set up a Bluetooth connection between your development computer and an Android device. When your app is ready, the last topic explains how to prepare your app for deployment.

Debug Android Wear on an Emulator

How to debug a Xamarin.Android Wear application on the Android SDK emulator.

Debug on a Wear Device

How to configure an Android device so that Xamarin.Android Wear applications can be deployed to it directly from either Visual Studio or Visual Studio for Mac.

Packaging Wear Apps

How to package Xamarin.Android Wear apps for distribution on Google Play.

Debug Android Wear on an Emulator

7/8/2021 • 2 minutes to read • Edit Online

These articles explain how to debug a Xamarin.Android Wear application on an emulator.

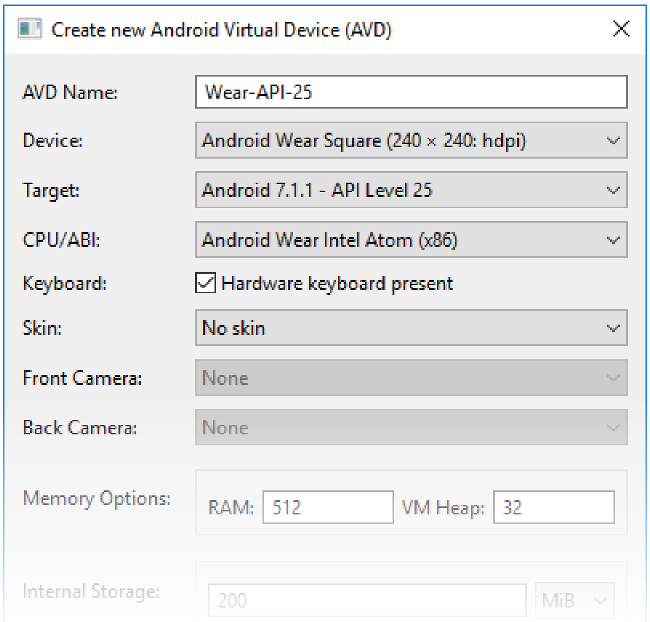
Debug Wear on Emulator Overview

Developing Android Wear applications requires running the application, either on physical hardware or using an emulator or simulator. Using hardware is the best approach, but not always the most practical. In many cases, it can be simpler and more cost effective to simulate/emulate Android Wear hardware using an emulator as described below. If you are not yet familiar with the process of deploying and running Android Wear apps, see Hello, Wear.

Configure the Android Emulator

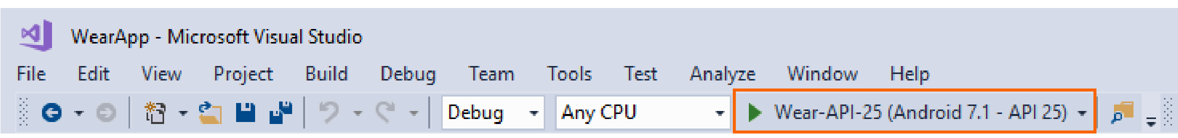
To run your Wear app on an emulator, you must install the Android SDK Android Emulator and configure it for Android Wear. For overall Android SDK Emulator installation and configuration information, see Android Emulator Setup.

When you create a Wear virtual device, select an Android Wear device profile (such as **Android Wear Square**). For improved performance, use the Wear **x86** CPU/ABI as seen in this example:



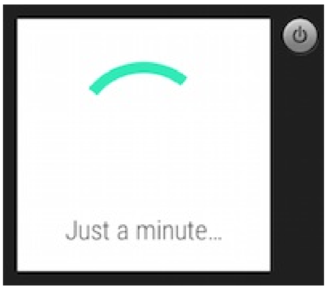
Launch The Wear Virtual Device

After you have created an Android Wear virtual device, you can choose it from the device pull-down menu in the IDE before you start debugging. If your virtual device is not available in the device pull-down, verify that your project is an Android *Wear* app project (not an Android app project) and that its target API level is set to the same API level as the virtual device. For example:



After the Android emulator starts, Xamarin.Android will deploy the Wear app to the emulator. The emulator runs the app with the configured virtual device image.

Don't be surprised if you see this (or another interstitial screen) at first. The watch emulator can take a while to start up:



The emulator may be left running; it is not necessary to shut it down and restart it each time the app is run.

Summary

This guide explained how to configure the Android Emulator for Wear development and launch a Wear virtual device for debugging.

Debug on a Wear Device

11/2/2020 • 3 minutes to read • Edit Online

This article explains how to debug a Xamarin.Android Wear application on a Wear device.

Overview

If you have an Android Wear device such as an Android Wear Smartwatch, You can run the app on the device instead of using an emulator. (If you are not yet familiar with the process of deploying and running Android Wear apps, see Hello, Wear.)

Prepare The Wear Device:

Use the following steps to enable debugging on the Android Wear device:

1. Open the **Settings** menu on the Android Wear device.
2. Scroll to the bottom of the menu and tap **About**.
3. Tap the build number 7 times.
4. On the **Settings** menu, tap **Developer Options**.
5. Confirm that **ADB debugging** is enabled.

Debugging over USB

If your Wear device has a USB port, you can connect the Wear device to your computer, deploy to it, and run/debug the app as you would using an Android phone (for more information, see Debug on a Device).

Debugging over Bluetooth

If your Wear device does not have a USB port, you can deploy the app to the Wear device over Bluetooth by routing the app's debug output to an Android phone that is connected to your computer.

Prepare Your Phone

Use the following steps to prepare your phone for making Bluetooth connections to the Wear device:

1. If you have not already done so, set up your phone for Xamarin.Android development as explained in Set Up Device for Development.
2. Download and install the free Android Wear app from the Google Play Store.

Connect The Device

Use the following steps to connect your Wear device to your Phone:

1. On the phone that will act as Bluetooth intermediary (configured above), start the Android Wear app.
2. Tap the **Settings** icon.
3. Enable **Debugging over Bluetooth**. You should see the following status displayed on the screen of the Android Wear app:

```
Host: disconnected
Target: connected
```

4. Connect the phone to your computer over USB. On your computer, enter the following commands:

```
adb forward tcp:4444 localabstract:/adb-hub
adb connect 127.0.0.1:4444
```

If port 4444 is not available, you can use any other available port to which you have access.

> **NOTE**
>
> If you restart Visual Studio or Visual Studio for Mac, you must run these commands again to setup a connection to the Wear device.

5. When the Wear device prompts you, confirm that you are allowing **ADB Debugging**. In the Android Wear app, you should see the status change to:

```
Host: connected
Target: connected
```

6. After you complete the above steps, running `adb devices` shows the status of both the phone and the Android Wear device:

```
List of devices attached
127.0.0.1:4444    device
019ad61df0a69399  device
```

At this point, you can deploy your app to the Wear device.

Taking screenshots

You can take a screenshot of the Wear device by entering the following command:

```
adb -s 127.0.0.1:4444 shell screencap -p /sdcard/DCIM/screencap.png
```

Copy the screenshot to your computer by entering the following command:

```
adb -s 127.0.0.1:4444 pull /sdcard/DCIM/screencap.png
```

Delete the screenshot on the device by entering the following command:

```
adb -s 127.0.0.1:4444 shell rm /sdcard/DCIM/screencap.png
```

Uninstalling an app

You can uninstall an app from the wear device by entering the following command:

```
adb -s 127.0.0.1:4444 uninstall <package name>
```

For example, to remove the app with the package name `com.xamarin.weartest`, enter the following command:

```
adb -s 127.0.0.1:4444 uninstall com.xamarin.weartest
```

For more information about debugging Android Wear devices over Bluetooth, see Debugging over Bluetooth.

Debugging a Wear app with a companion phone app

Android Wear apps are packaged with a companion Android phone app for distribution on Google Play (for more information, see Working with Packaging). However, you still develop the Wear app and its companion app separately. When you release your app through the Google Play Store, the Wear app will be packaged with the companion app and automatically installed if possible.

To debug the Wear app with a companion app:

1. Build and deploy the companion app to the phone.

2. Right-click the Wear project and set it as the default start project.

3. Deploy the Wear project to the wearable device.

4. Run and debug the Wear app on the device.

Summary

This article explained how to configure an Android Wear device for Wear debug from Visual Studio via Bluetooth, and how to debug a Wear app with a companion phone app. It also provided common debugging tips for debugging a Wear app via Bluetooth.

Packaging Wear Apps

11/2/2020 • 3 minutes to read • Edit Online

WARNING

The following docs and sample projects may no longer be maintained. As of Xamarin.Android 11.1, automatically packaging an Android Wear application within an Android handheld application is no longer supported. It is recommended to distribute Android Wear applications as standalone applications instead.

Android Wear 1.0 apps are packaged with a full Android app for distribution on Google Play.

Android Wear 2.0 apps can be submitted to Google Play as standalone applications.

Automatic Packaging

Starting with Xamarin Android 5.0, your Wear app is automatically packaged as a resource in your Handheld app when you create a project reference from the Handheld project to the Wear project. You can use the following steps to create this association:

- Visual Studio
- Visual Studio for Mac

1. If your Wear app is not already part of your Handheld solution, right-click the solution node and select **Add > Add Existing Project...**.
2. Navigate to the **.csproj** file of your Wear app, select it, and click **Open**. The Wear app project should now be visible in your Handheld solution.
3. Right-click the **References** node and select **Add Reference**.
4. In the **Reference Manager** dialog, enable your Wear project (click to add a check mark), then click **OK**.
5. Change the package name for your Wear project so that it matches the package name of the Handheld project (the package name can be changed under **Properties > Android Manifest**).

Note that you will get an **XA5211** error if the package name of the Wear app does not match the package name of the Handheld app. For example:

```
Error XA5211: Embedded wear app package name differs from handheld
app package name (com.companyname.mywearapp != com.companyname.myapp). (XA5211)
```

To correct this error, change the package name of the Wear app so that it matches the package name of the Handheld app.

When you click **Build > Build All**, this association triggers automatic packaging of the Wear project into the main Handheld (Phone) project. The Wear app is automatically built and included as a resource in the Handheld app.

The assembly that the Wear app project generates is not used as an assembly reference in the Handheld (Phone) project. Instead, the build process does the following:

- Verifies that the package names match.

- Generates XML and adds it to the Handheld project to associate it with the Wear app. For example:

```
<!-- Handheld (Phone) Project.csproj -->
<ProjectReference Include="..\MyWearApp\MyWearApp.csproj">
    <Project>{D80E1FEF-653B-448C-B2AA-609C74E88340}</Project>
    <Name>MyWearApp</Name>
    <IsAppExtension>True</IsAppExtension>
</ProjectReference>
```

- Adds the Wear app as a **raw** resource to the Handheld project.

Manual Packaging

You can write Android Wear apps in Xamarin.Android before version 5.0, but you must follow these manual packaging instructions to distribute the app:

1. Ensure that your Wearable project and Handheld (Phone) projects have the same version number and package name.

2. Manually build the Wearable project as a **Release** build.

3. Manually add the release **.APK** from step (2) into the **Resources/raw** directory of the Handheld (Phone) project.

4. Manually add a new XML resource **Resources/xml/wearable_app_desc.xml** in the Handheld project which refers to Wearable **APK** from step (3):

```
<wearableApp package="wearable.app.package.name">
    <versionCode>1</versionCode>
    <versionName>1.0</versionName>
    <rawPathResId>NAME_OF_APK_FROM_STEP_3</rawPathResId>
</wearableApp>
```

5. Manually add a `<meta-data />` element to the Handheld project's **AndroidManifest.xml** `<application>` element that refers to the new XML resource:

```
<meta-data android:name="com.google.android.wearable.beta.app"
    android:resource="@xml/wearable_app_desc"/>
```

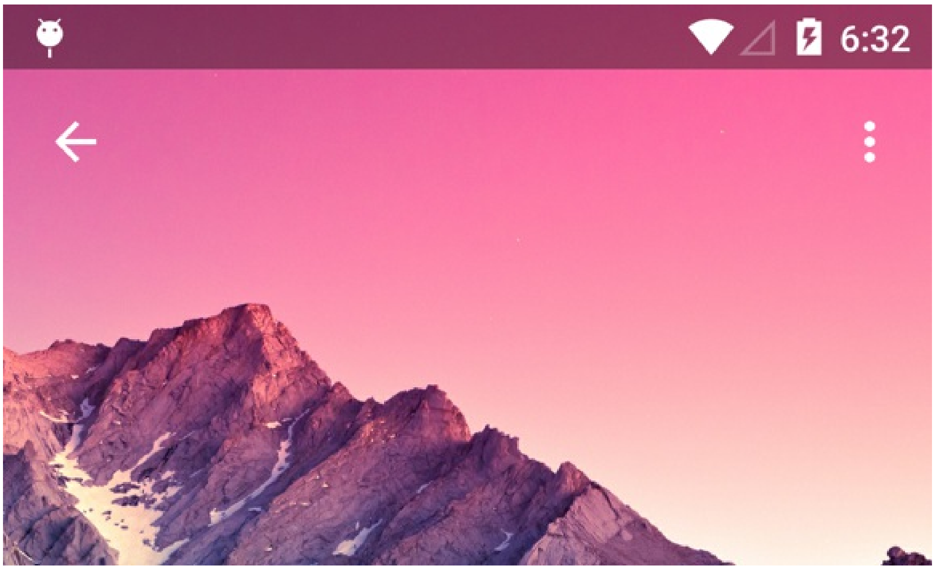
See also the Android Developer site's manual packging instructions.

Xamarin.Android samples

7/8/2021 • 2 minutes to read • Edit Online

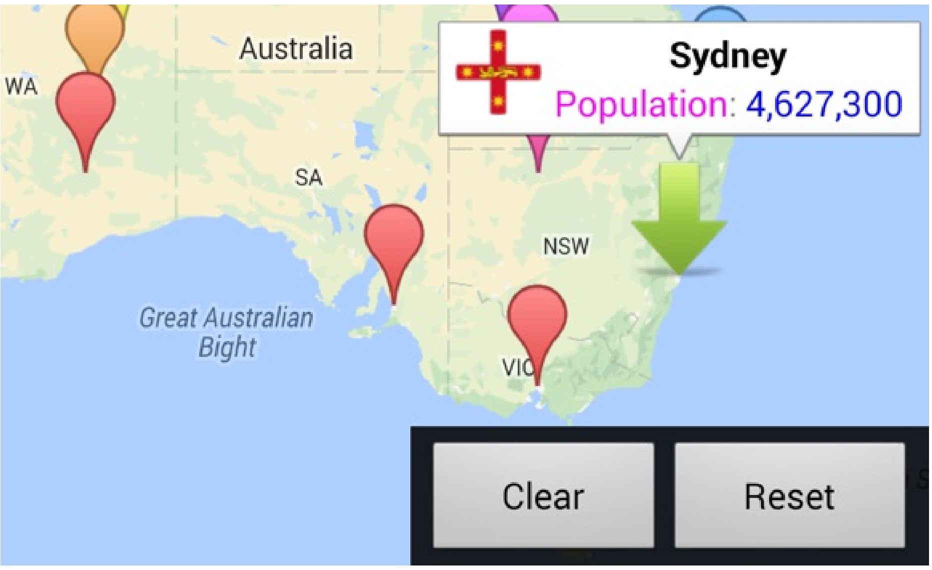
These Xamarin Android sample apps and code demos can help you get started building mobile apps with C# and Xamarin.

All Xamarin.Android samples



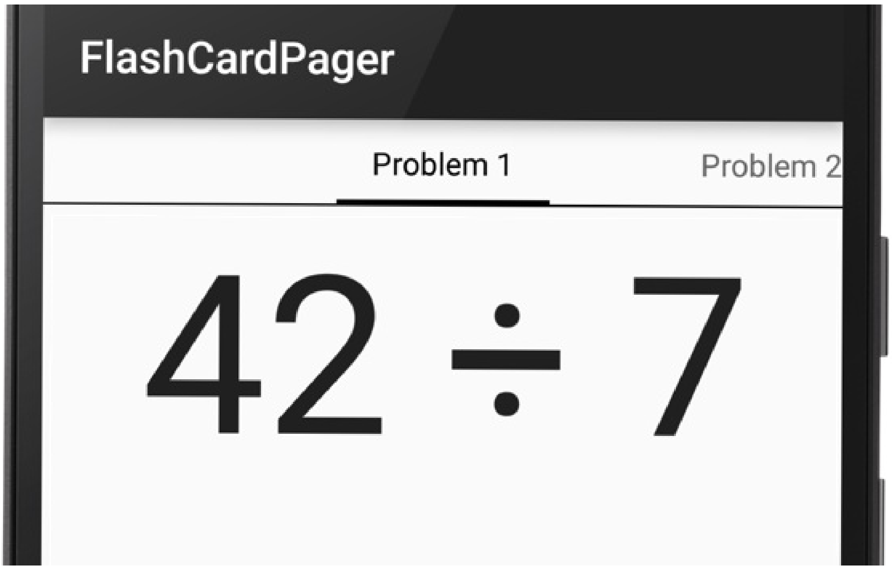
Material Design

This sample demonstrates the new Material Design APIs introduced in Android Lollipop.



Google Play Services

This solution uses the Xamarin Google Play Services NuGet to demonstrate a few uses of the maps API.

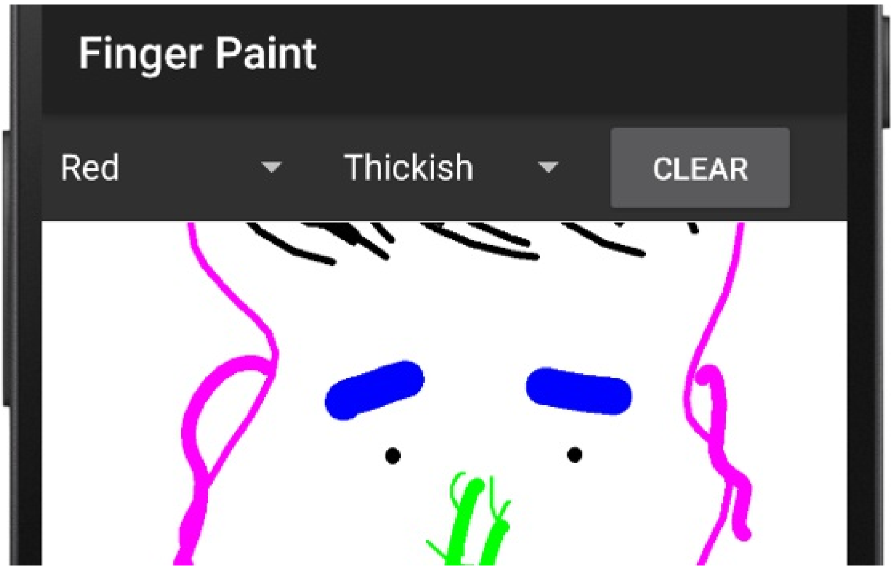


Flash Card Pager

This sample demonstrates how to use ViewPager and PagerTabStrip together to implement an app that presents a series of math problems on flash cards.

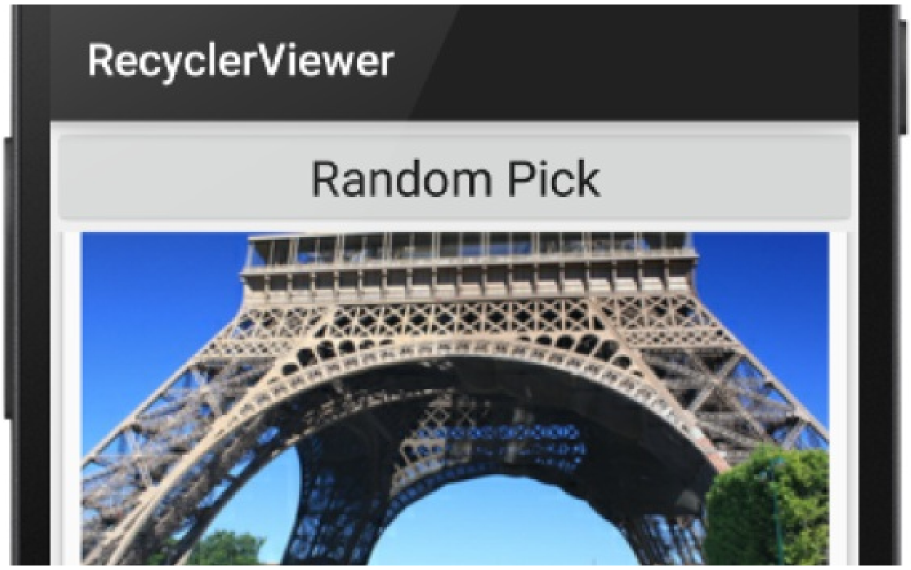
Fragments

Fragments are self-contained, modular components that are used to help address the complexity of writing applications that may run on screens of different sizes.



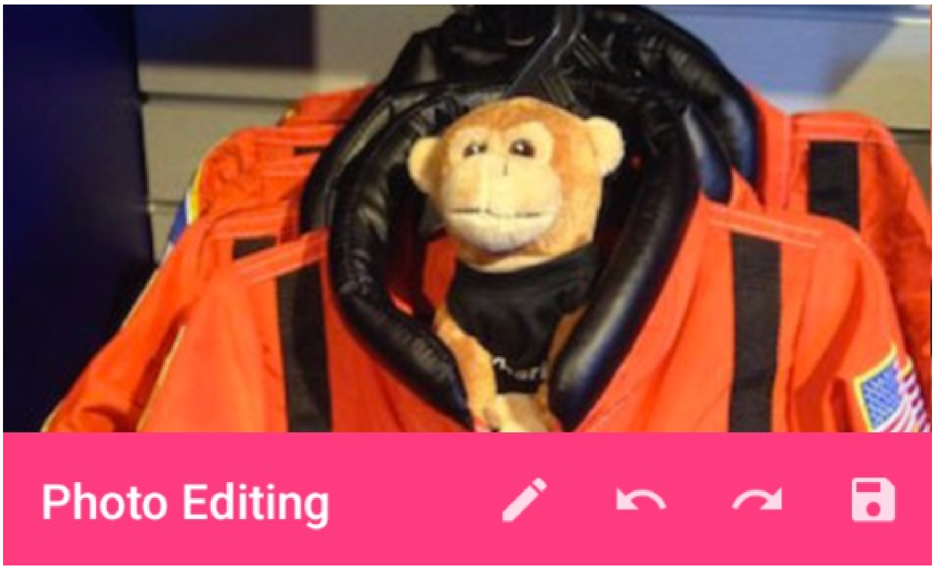
Finger Paint

Colorful finger-painting app using multi-touch tracking on Android.



RecyclerViewer

Use this sample to learn how to use the new CardView and RecyclerView widgets introduced with Android 5.0 Lollipop.



Toolbar

Android sample replacing the ActionBar with the new ToolBar in Android 5.0 Lollipop.

WatchFace

How to implement a custom Android Wear watch face.

All samples

For the complete set of Xamarin Android sample apps and code demos see All Xamarin.Android samples.