Android
Apprentice

FIRST EDITION

Beginning Android development with Kotlin 1.2

By Darryl **Bayliss** & Tom **Blankenship**

Android Apprentice

Darryl Bayliss & Tom Blankenship

Copyright ©2018 Razeware LLC.

Dedications

"To Rachael. For putting up with me whilst I spent nights and weekends behind my laptop writing. To my family, for being there for me always."

— *Darryl Bayliss*

"To my wife Tracy, for her love and understanding while I worked many late nights and weekends. To my son Austin for his support and many corrections to my rough drafts. To my daughter Alaina, for her beautiful smiles and encouragement. To my parents who bought my first Atari computer and set me on this great journey."

— *Tom Blankenship*

About the authors
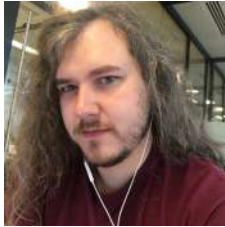
Darryl Bayliss is an author of this book. Darryl is a Software Engineer from Liverpool, focusing on Mobile Development. Away from programming he is usually reading, getting up to no good or playing some fantastical video game involving magic and dragons. Feel free to say hello on Twitter over at @dazindustries

Tom Blankenship is an author on this book. Tom has been addicted to coding since he was a young teenager, writing his first programs on Atari home computers. He currently runs his own software development company focused on native iOS and Android app development. He enjoys playing tennis, guitar, and drums, and spending time with his wife and two children.

About the editors

Namrata Bandekar is a tech editor of this book. Namrata is a Mobile Developer doing native Android and iOS development. Apart from developing apps, she is passionate about traveling, food and hiking with her dog. You can reach out to her on Twitter at @NamrataCodes.

Vijay Sharma is a tech editor of this book. Vijay is a husband, a father and a senior mobile engineer. Based out of Canada's capital, Vijay has worked on dozens of apps for both Android and iOS. When not in front of his laptop, you can find him in front of a TV, behind a book, or chasing after his kids. You can reach out to him on Twitter: @v_sharm

Ellen Shapiro is a tech editor on this book. Ellen is an iOS developer for Bakken & Bæck's Amsterdam office who also occasionally writes Android apps. She is working in her spare time to help bring songwriting app Hum to life. She's also developed several independent applications through her personal company, Designated Nerd Software. When she's not writing code, she's usually tweeting about it at @designatednerd

Chris Belanger is an editor of this book. Chris is the Editor in Chief at raywenderlich.com. He was a developer for nearly 20 years in various fields from e-health to aerial surveillance to industrial controls. If there are words to wrangle or a paragraph to ponder, he's on the case. When he kicks back, you can usually find Chris with guitar in hand, looking for the nearest beach. Twitter: @crispytwit.

Tammy Coron is an editor of this book. Tammy is an independent creative professional and the host of Roundabout: Creative Chaos. She's also the co-founder of Day Of The Indie and the founder of Just Write Code. For more information visit TammyCoron.com.

Eric Soto is the final pass editor of this book. Eric is a Professional Software Engineer and certified Agile-Scrum Master focusing on Apple iOS & Android Apps, NodeJS and APIs. He is based in Palm Beach Florida but works with clients all across the US including many major brands. During his 30 year career, Eric has been able to work with REST APIs, web applications, server back-end systems, automated infrastructure deployments and more. Follow Eric on Twitter: @ericwastaken

Table of Contents: Overview

Table of Contents: Extended

Introduction

This book is your introduction to building great apps in Android, using the Kotlin language. Whether you still consider yourself a novice programmer, or have extensive experience programming for iOS or other platforms, this is the book for you!

It's not our aim to teach you all the ins and outs of Android development or the Kotlin language; they are huge concepts on their own and there is no way we can cover everything. Fortunately, you really just need to master the essential building blocks of Kotlin and Android to start creating apps. As you work on more apps, you'll find the foundations you learn in this book will give you the knowledge you need to easily figure out more complicated details on your own.

The most important thing you'll learn is how to think like a programmer. That will help you approach any programming task, whether it's a game, a utility, a mobile app that uses web service, or anything else you can imagine.

If you're looking for more background on the Kotlin language, we recommend our book, the Kotlin Apprentice, which goes into depth on the Kotlin language itself:

- https://store.raywenderlich.com/products/kotlin-apprentice

Book license

By purchasing *Android Apprentice*, you have the following license:

- You are allowed to use and/or modify the source code in *Android Apprentice* in as many apps as you want, with no attribution required.

- You are allowed to use and/or modify all art, images and designs that are included in *Android Apprentice* in as many apps as you want, but must include this attribution line somewhere inside your app: "Artwork/images/designs: from *Android Apprentice*, available at www.raywenderlich.com".

- The source code included in *Android Apprentice* is for your personal use only. You are NOT allowed to distribute or sell the source code in *Android Apprentice* without prior authorization.

- This book is for your personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, coworkers or students; they would need to purchase their own copies.

All materials provided with this book are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the properties of their respective owners.

Book source code and forums

This book comes with the source code for the starter and completed projects for each chapter. These resources are shipped with the digital edition you downloaded from store.raywenderlich.com.

We've also set up an official forum for the book at forums.raywenderlich.com. This is a great place to ask questions about the book or to submit any errors you may find.

About the cover

The leafbird is a tropical bird found mainly throughout the southern Indian and Asian subcontinents. Their predominant color is green, but the many various species of leafbird sport various swaths of color, including orange, yellow, blue and black.

The Android OS is a lot like the leafbird; although leafbirds all share common characteristics and coloring, all species have a slightly different appearance and behavior. As you begin to develop for Android and experience what's become known as the fragmentation problem, you'll see that each "species" or version of Android has its own little quirks as well!

Section I: Your First Android App

This is your introduction to creating apps in Android. This section will take you step-by-step through installing Android Studio and working inside the IDE and visual designer while you build **TimeFighter**, a simple game that uses many common Android components.

Chapter 1: Setting Up Android Studio

Chapter 2: Layouts

Chapter 3: Activities

Chapter 4: Debugging

Chapter 5: Prettifying The App

Chapter 1: Setting Up Android Studio

By Darryl Bayliss

To create that killer Android App, you'll need to install the tools that you need as a young apprentice. Android development happens inside **Android Studio**, a customized IDE, based on IntelliJ, that gives you a powerful set of tools to work with.

In this chapter you'll learn:

- How to set up Android Studio on your machine.

- How to set up a physical and emulated device for development.

- How to run an app on a device.

Getting started

Open up your favorite web browser and navigate to https://developer.android.com/studio/#downloads.

Android Studio downloads

Platform	Android Studio package	Size	SHA-256 checksum
Windows (64-bit)	android-studio-ide-181.5056338-windows.exe Recommended	927 MB	6ee509f3391757fe87cc5c1e4970a0228fc1ad6ca34a8b31c0a28926179353a9
	android-studio-ide-181.5056338-windows.zip No .exe installer	1001 MB	21aebb3a7fab4931b830ec40d836d6945eabb4f32acf1b52fae148d33599fd7c
Windows (32-bit)	android-studio-ide-181.5056338-windows32.zip No .exe installer	1000 MB	3a61a587c90e358ab15d076d0306550564ad4cc5a8aa1dd0c22c6afd092e976a
Mac	android-studio-ide-181.5056338-mac.dmg	989 MB	b8d2b7add6a7c776d16a8e48bd35c3e2bba18b4717131d7b9a00fa416ebe4480
Linux	android-studio-ide-181.5056338-linux.zip	1007 MB	b9ec0d44f2feaafe1e3fbd1ed696bf325f9e05cfb6c1ace84dbf87ae249efa84

You have a number of download options, as Android Studio can run on a variety of Operating Systems. **Click** the package for the operating system your computer uses.

> **Note**: This chapter assumes that your computer is running macOS; however, as Android Studio supports Windows and Linux, we'll provide instructions for those operating systems as well.

The downloads page will pop up a terms and conditions screen.

Check the checkbox at the bottom of the screen if you agree to the terms and then **Click** the Download Android Studio button.

As your computer begins to download Android Studio, your browser will show a pop-up window with some suggestions on what to do next.
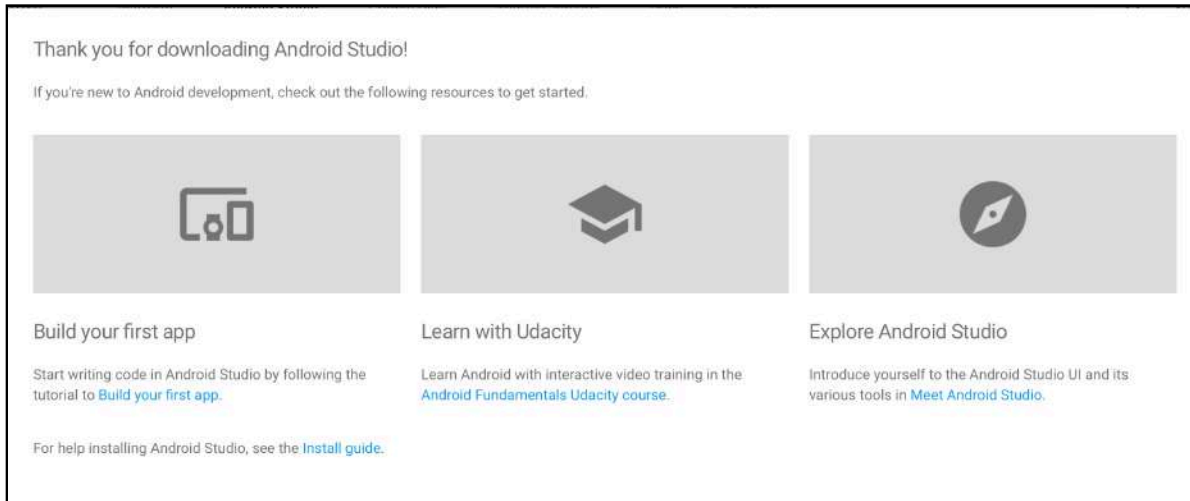
Click the blue install guide link at the bottom of the pop-up. The browser will open up a new page containing setup videos for Android Studio for each operating system.
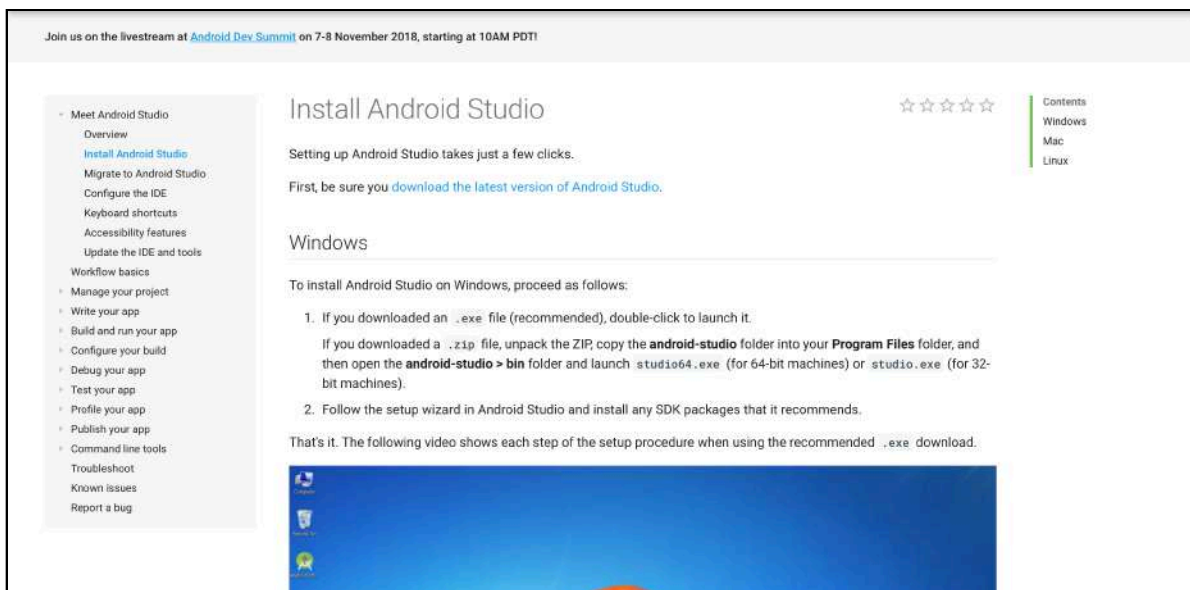
Follow the instructions and/or video relevant to your operating system until the welcome to Android Studio window is open.

> **Note**: Unless you have an extremely fast internet connection, it will take a while for all of the components you need to download. Depending on how your system is set up, you may need to enter your password or an administrator's password to allow installation to complete.
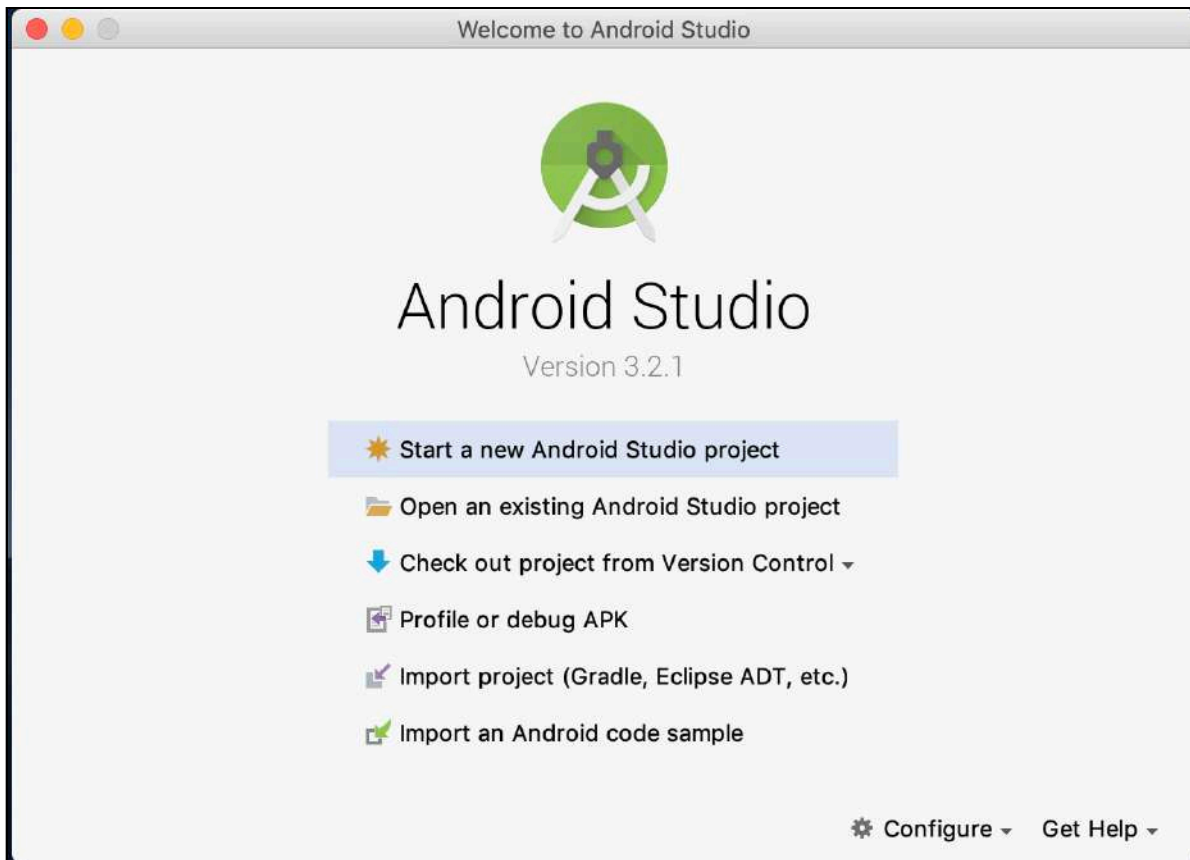
Your first Android project

Now that you have Android Studio installed, it's time to create your first project. This chapter focuses on getting your app running as quickly as possible on a device. Along the way, you'll encounter a few screens that you won't quite understand at first, but don't worry: you'll get a chance to experience these screens in detail later on in "Chapter 6: Creating a New Project." For now, just enjoy the ride!

On the welcome screen, click **Start a new Android Studio project**:

The welcome window will disappear and a new window will take its place. This is where you can set up a few key elements of your app.
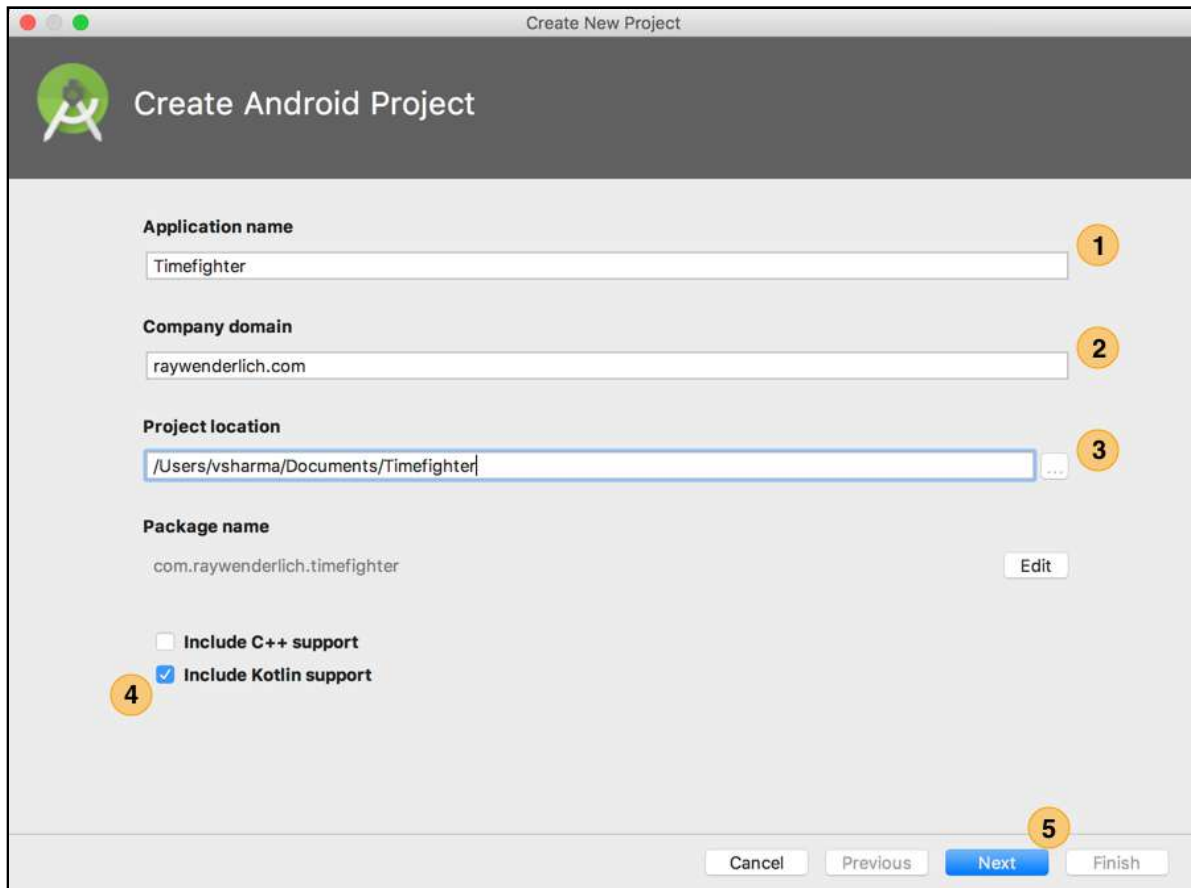
Here's what each field means:

1. The first and most important thing in any app is its name. In the **Application name** textfield, enter **Timefighter**.

2. The **Company domain** textfield provides your app with a package name, a concept you should be familiar with from Java or Kotlin. Here, simply enter **raywenderlich.com**.

3. The **Project location** textfield tells Android Studio where to create the directory containing your project.

Note: Feel free to create your project anywhere you want. The ellipsis button to the right provides you with a system navigator to easily find the place to create your project.

4. The **Include Kotlin Support** checkbox informs Android Studio that your project requires the Kotlin libraries to be added so you can write your app in Kotlin. It also ensures the starter project code is all Kotlin. Check the box if it isn't already checked.

5. The **Next** button in the bottom right of the window moves you to the target device section.

The next window provides a variety of checkboxes and dropdowns for you to configure which versions of Android you want to support:
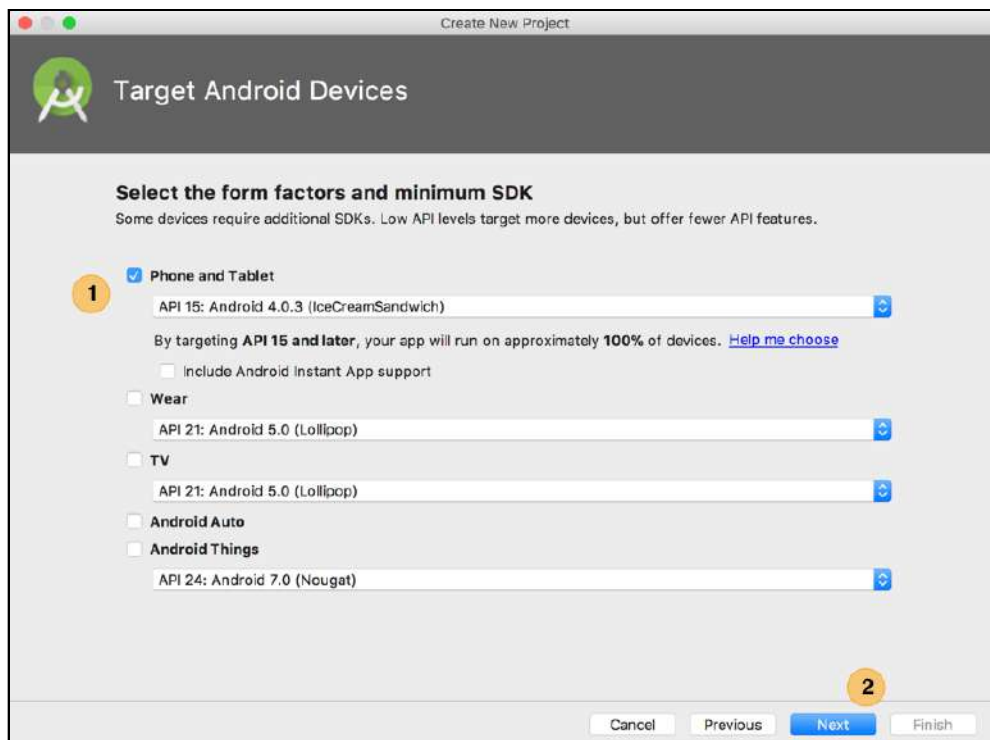
On this screen:

1. Android supports a variety of devices: phones, watches, cars and even refrigerators! If you want to configure a new app to run on these devices, this is the screen to do it.

   Timefighter will only run on a phone, and Android Studio has already checked that option for you. As well, it has set a minimum Android version for the app in the dropdown box.

   This book requires your apps to run on API 19, or Android KitKit in English. So click the dropdown and click **API 19: Android 4.4 (KitKat)**.
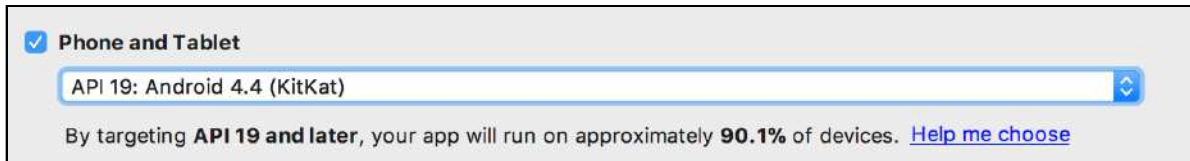
2. The **Next** button in the bottom-right of the window moves you to the project template section.

The next window provides a variety of preset projects you can choose from to set up the foundation of your app. Each choice provides your starter project with different code and resources generated by Android Studio.
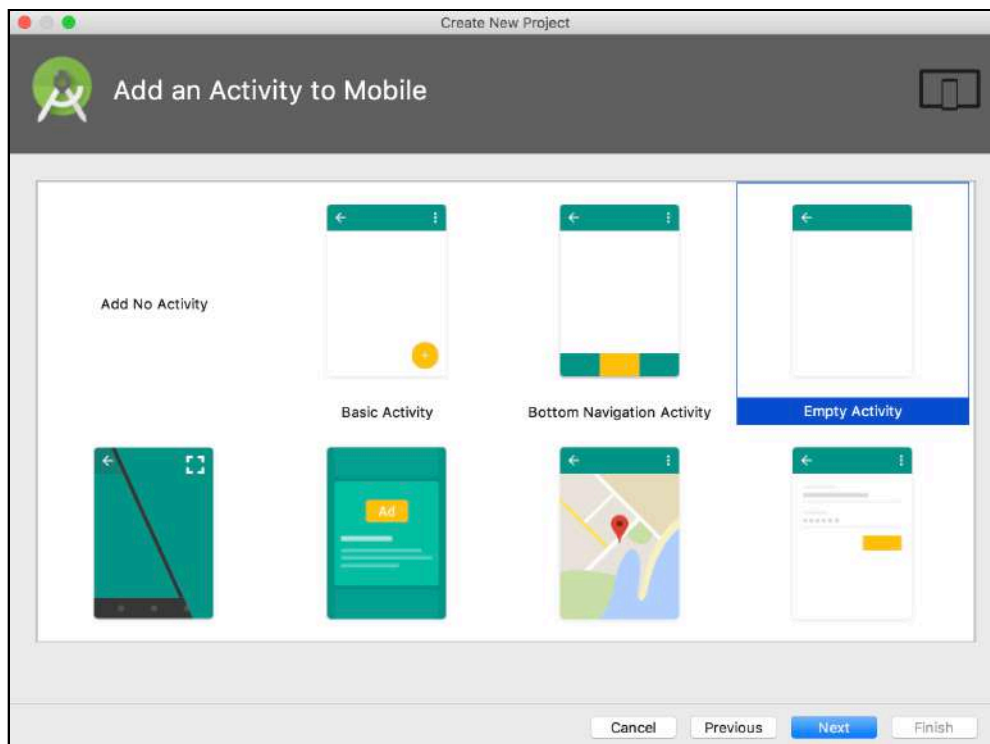
The **Empty Activity** is already selected for you by Android Studio, which is exactly what you want. Don't worry about what an Activity is right now; all will be revealed in the chapters to come.

Go to the bottom of the window and press the **Next** button.

The final window is dedicated to setting up your empty Activity — in particular, giving it a name.
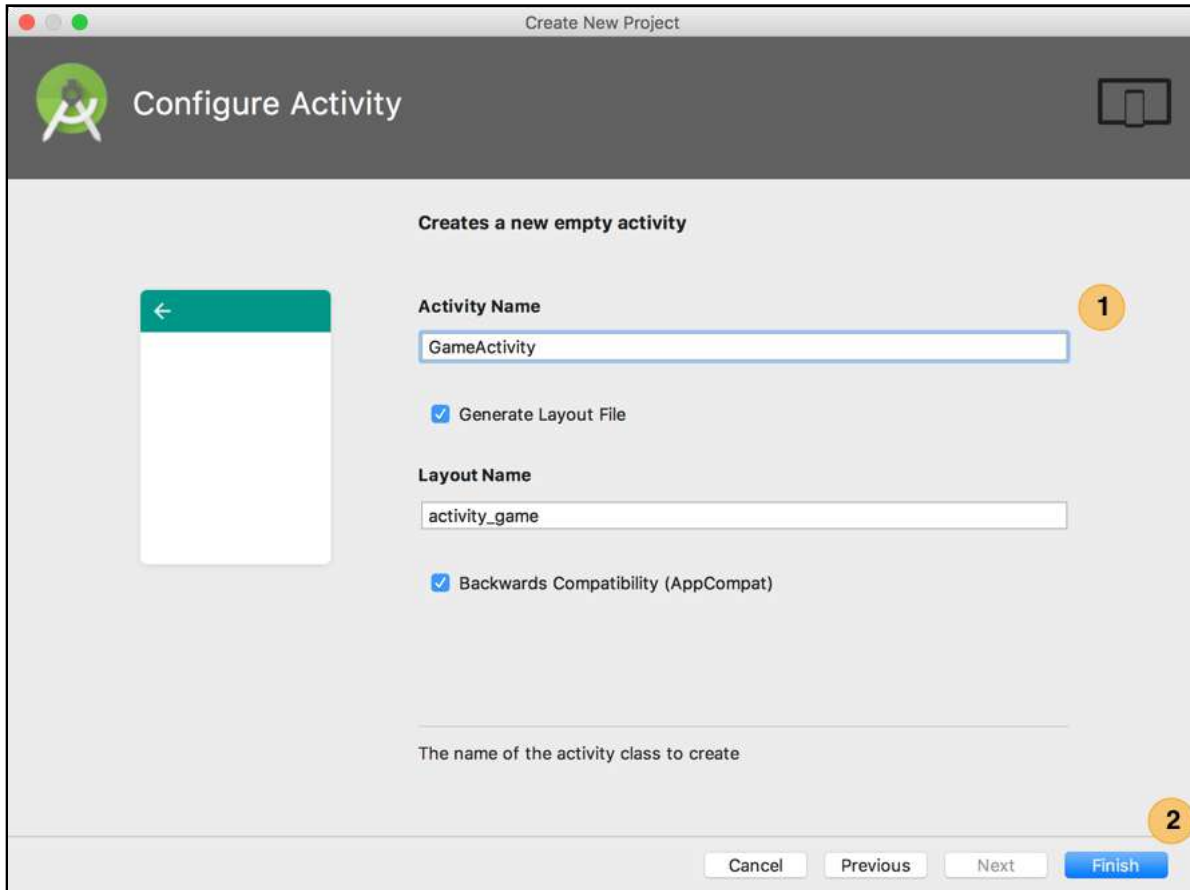
1. In the **Activity Name** textfield, enter the name **GameActivity**. You'll notice that the name of the layout automatically updates itself to `activity_game`.

2. Skip past the rest of the options and click **Finish**.

With the project creation taken care of, Android Studio will take all the information you provided, and gather the required libraries and resources to generate a fresh project for you.

> **Note**: The "gathering" phase may take a while, depending on the speed of your internet connection.

When that process finishes, Android Studio will show you your created project, with the `GameActivity.kt` and `activity_game.xml` files already open for you:
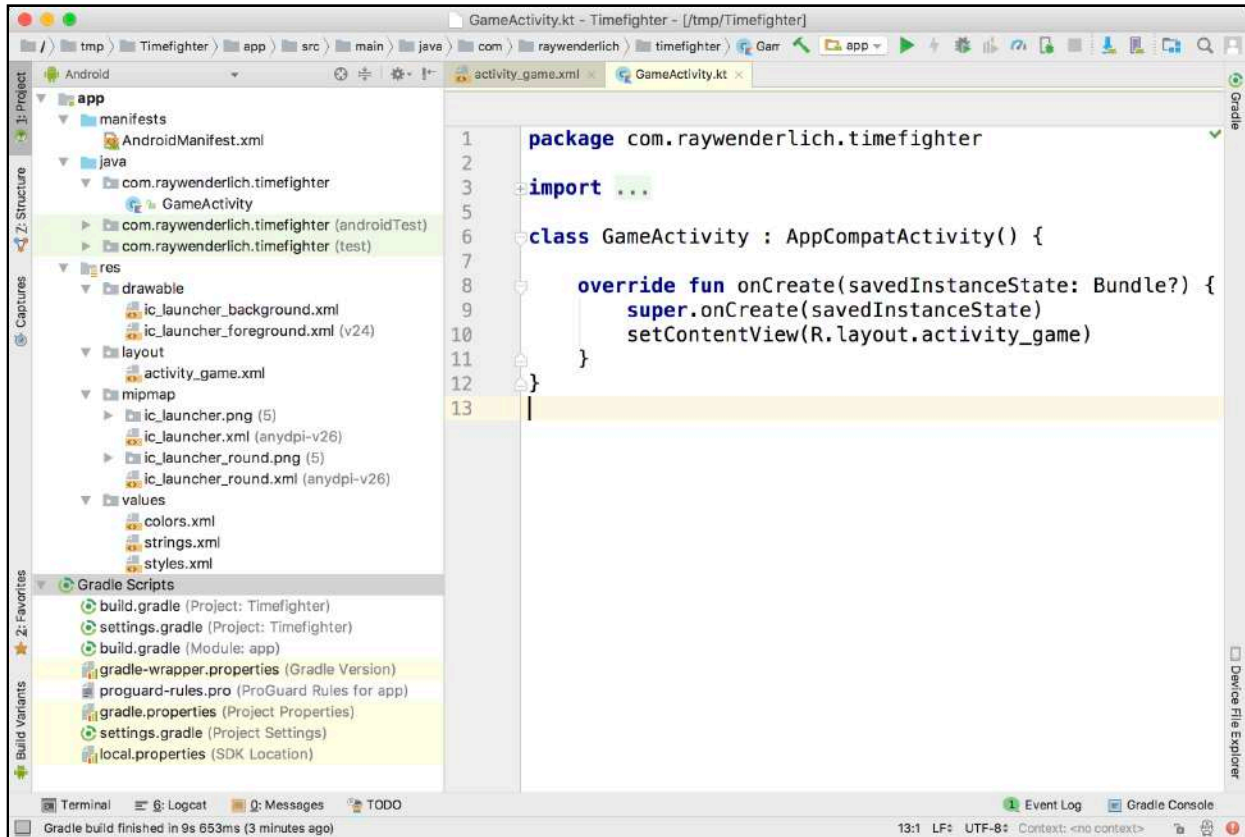
Android Studio

With your project created, you are now free to work on your project as you please.

Android Studio is a large and complex piece of software and, if you dive in without a good map, you may find yourself lost!

Before you start to build you app, take a look at what Android Studio has to offer as part of your app development experience.
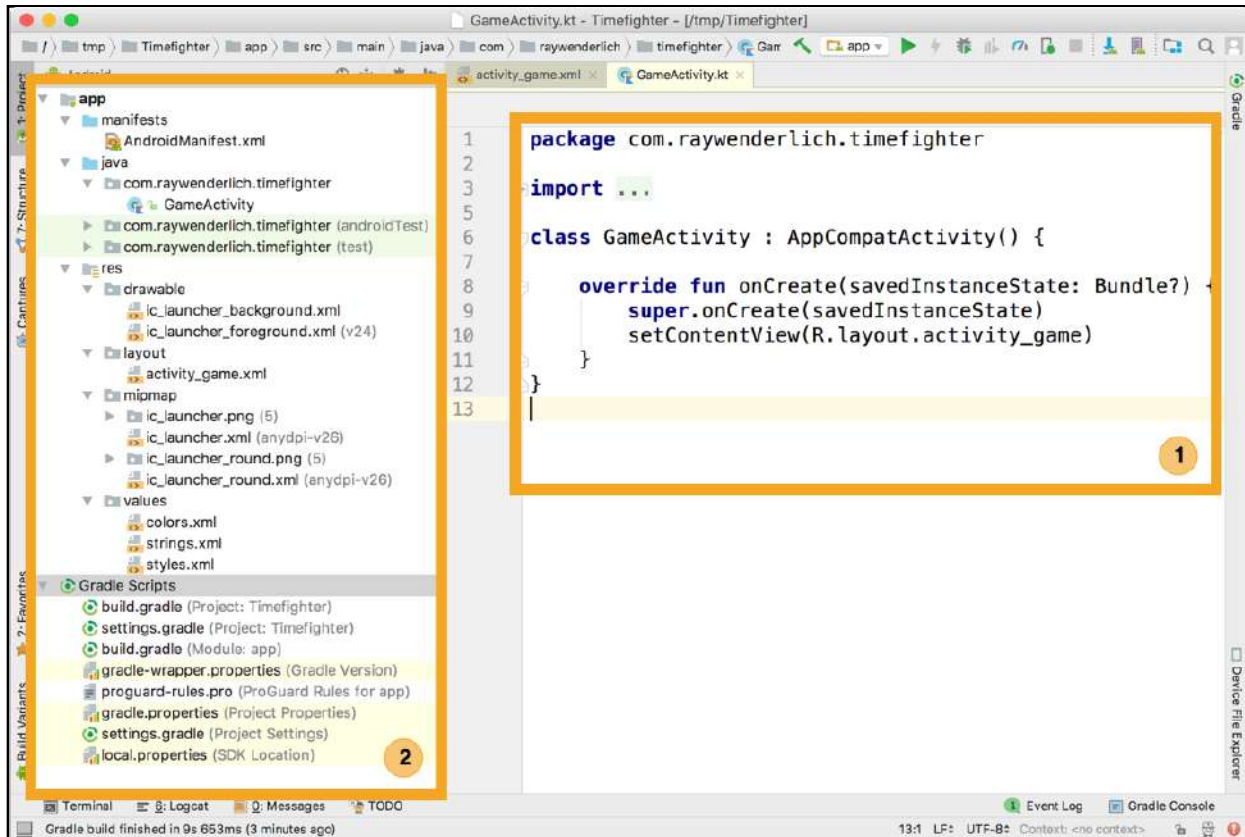
1. The most obvious window that first appears to you is the **Editor**.

   This window provides you with space to edit your app's source code. It provides syntax highlighting, auto completion for methods and objects as well as the ability to drop breakpoints into your code while debugging. You'll learn more about breakpoints and debugging in "Chapter 4: Debugging." You'll spend most of your development time using the Editor to code your app to work exactly the way you intend.

2. The other window that you'll spend most of your time with is the **Project navigator**, to the left of the Editor.

   This window shows you everything your project contains; from code to image assets, you can find it here. Android Studio already provides you with a lot to begin with. You can see this by left-clicking on the arrow to the left of the items in the Project navigator.

   Don't worry about these files for now. You'll become well-acquainted with them in the chapters to come.

Creating an Android virtual device

> **Note**: If you have a physical Android device you want to use for development, feel free to skip to the next section.

Looking at editors and files is great, but after you're done writing code, you'll likely want to run your app. But before you can run your app, you need a device — real or virtual — on which to run it! Take a look at the button highlighted in the following image.
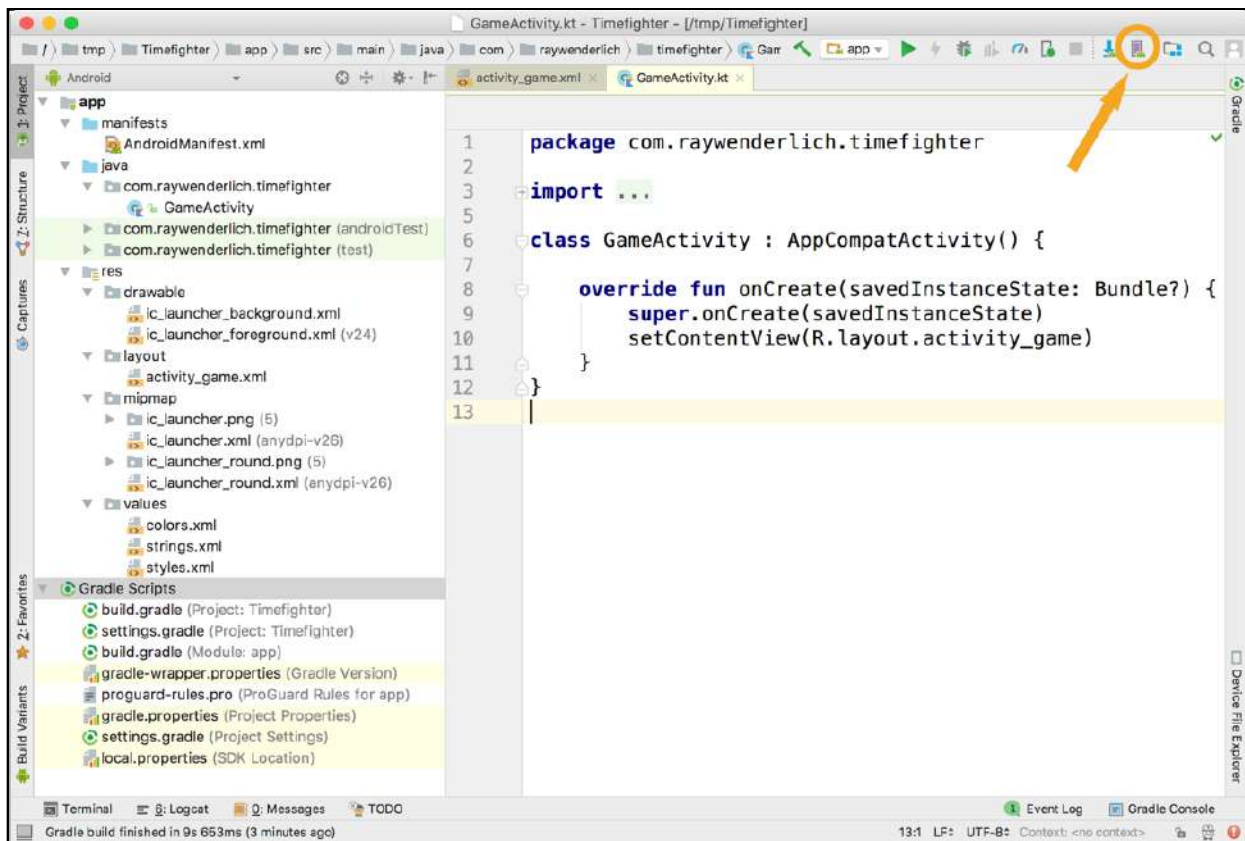
This button allows you to create an **Android Virtual Device**, or AVD for short. This is an emulator that pretends to be a device on your computer, which lets you test your app without requiring a physical device. If you don't have a physical device to test your app with, you'll need to create a virtual device before you can run your app.

Click the Android Virtual Device button, and a new window will appear.

This window shows all the available AVDs that exist on your machine. Since you've just installed Android Studio, no AVDs will be available yet.
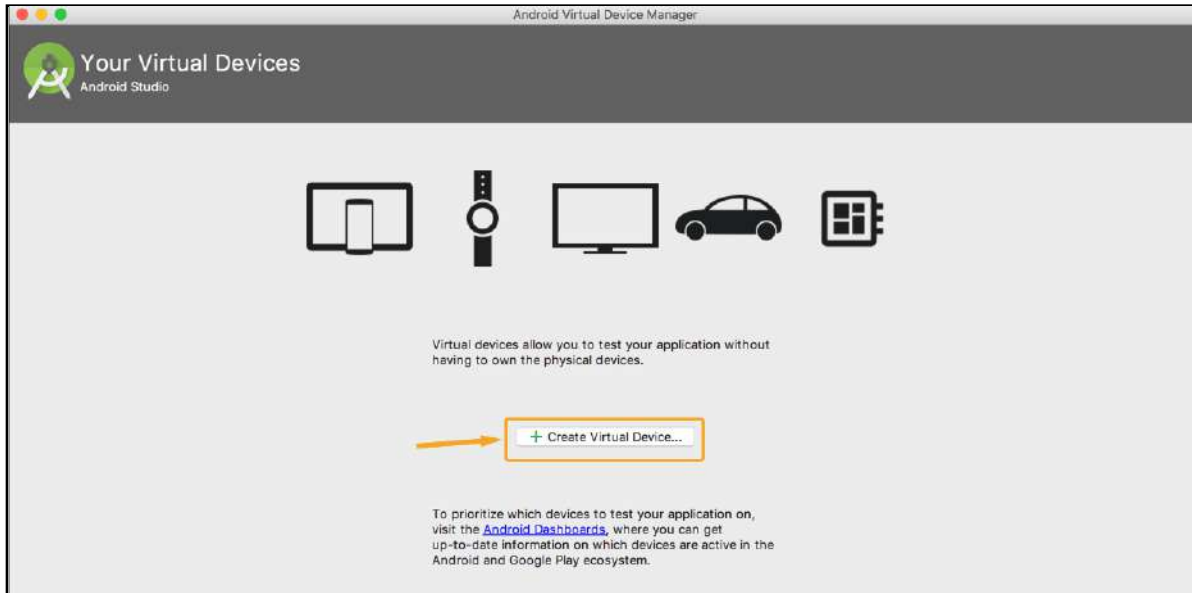
Click the **Create Virtual Device** button in the middle of the screen and the **Select Hardware** window. This window allows you to select what kind of device you want your AVD to emulate.
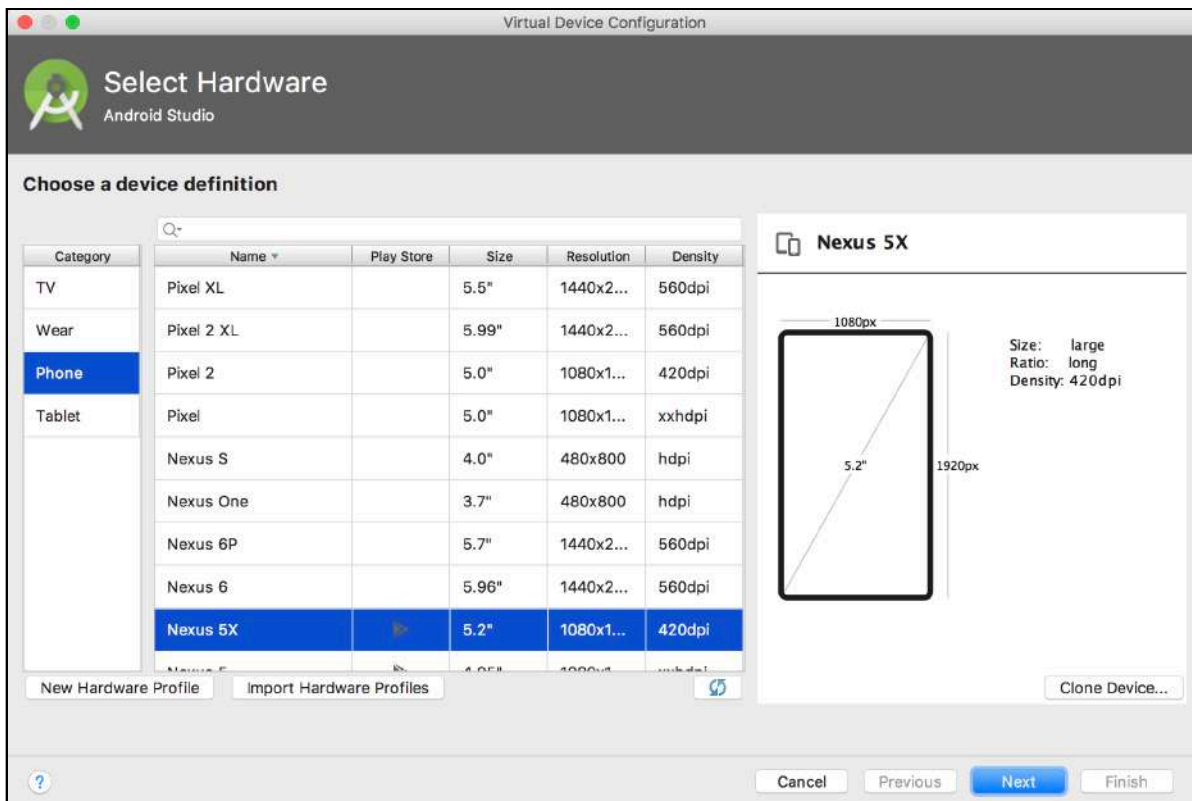
You will already have a device selected for you: a Nexus 5X. You'll use this device since it closely emulates a real device used by many people.

In the bottom-right of the window, click **Next**. This will display the **System Image** window, where you can choose the version of Android that runs on your emulator.
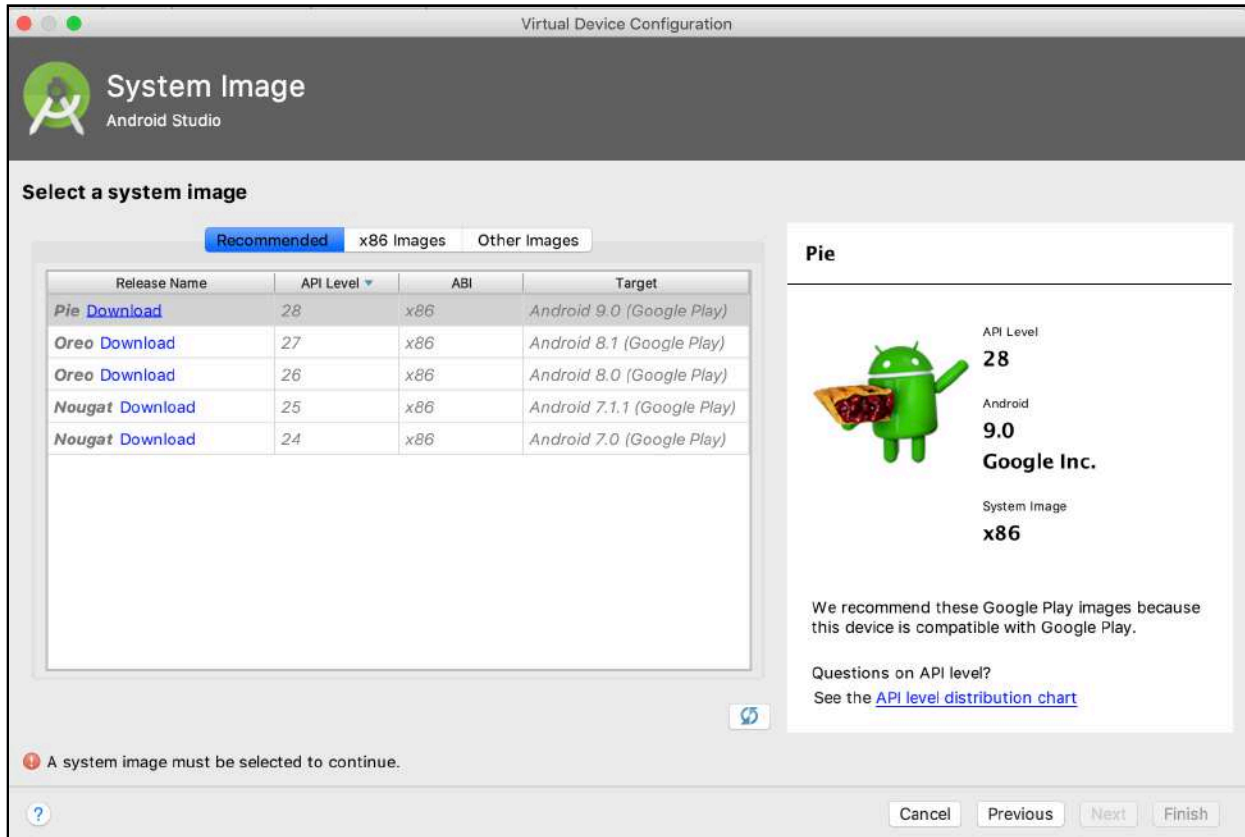
A number of tabs run along the top of the list within the window. The **Recommended** tab is a list of Android versions that Google recommend you use to test your apps.

At the moment, those versions are grayed out. That's because you haven't installed any of them onto your machine.

You'll download the latest and greatest recommended by Android Studio. Select the top item in the table and click the Download button in the **Release Name** column.

You'll see another legal agreement you need to agree to:
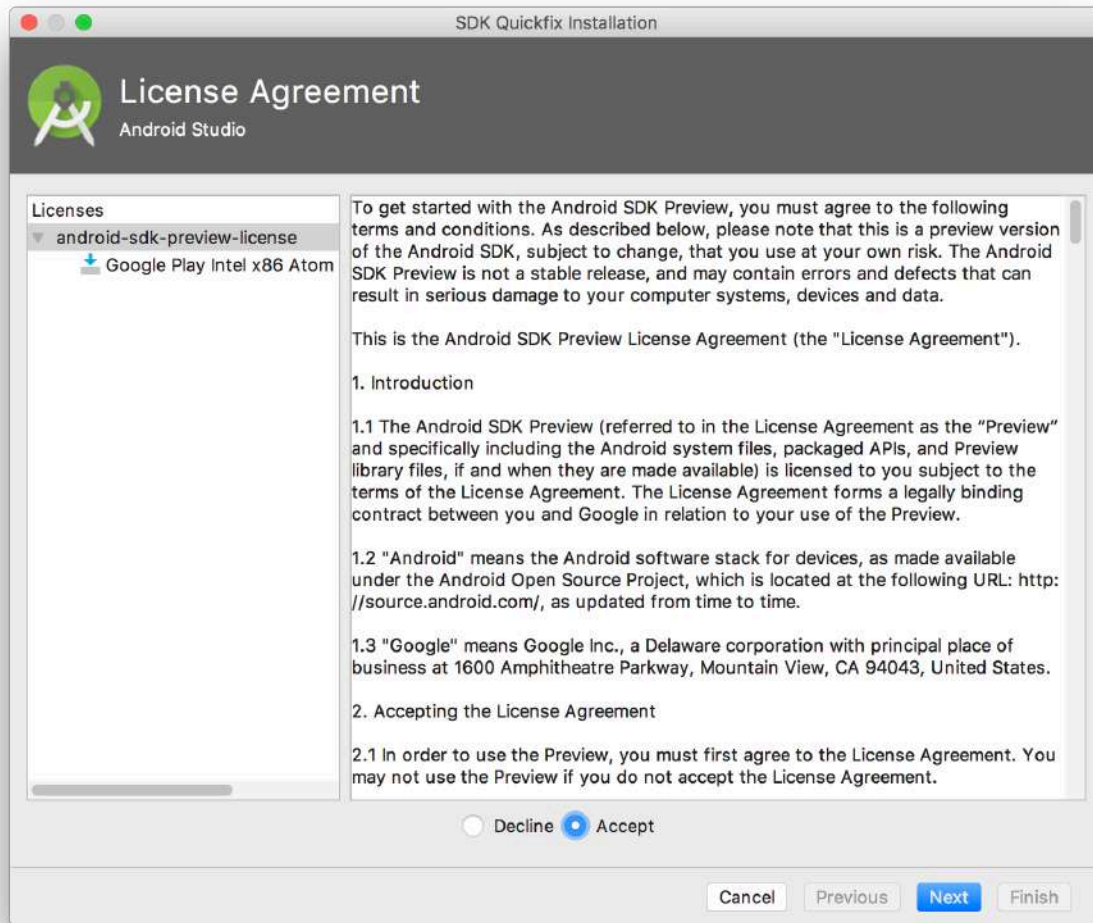
Select **Accept**, then click **Next**.

The **Component Installer** window will appear and automatically download the version of Android you selected.
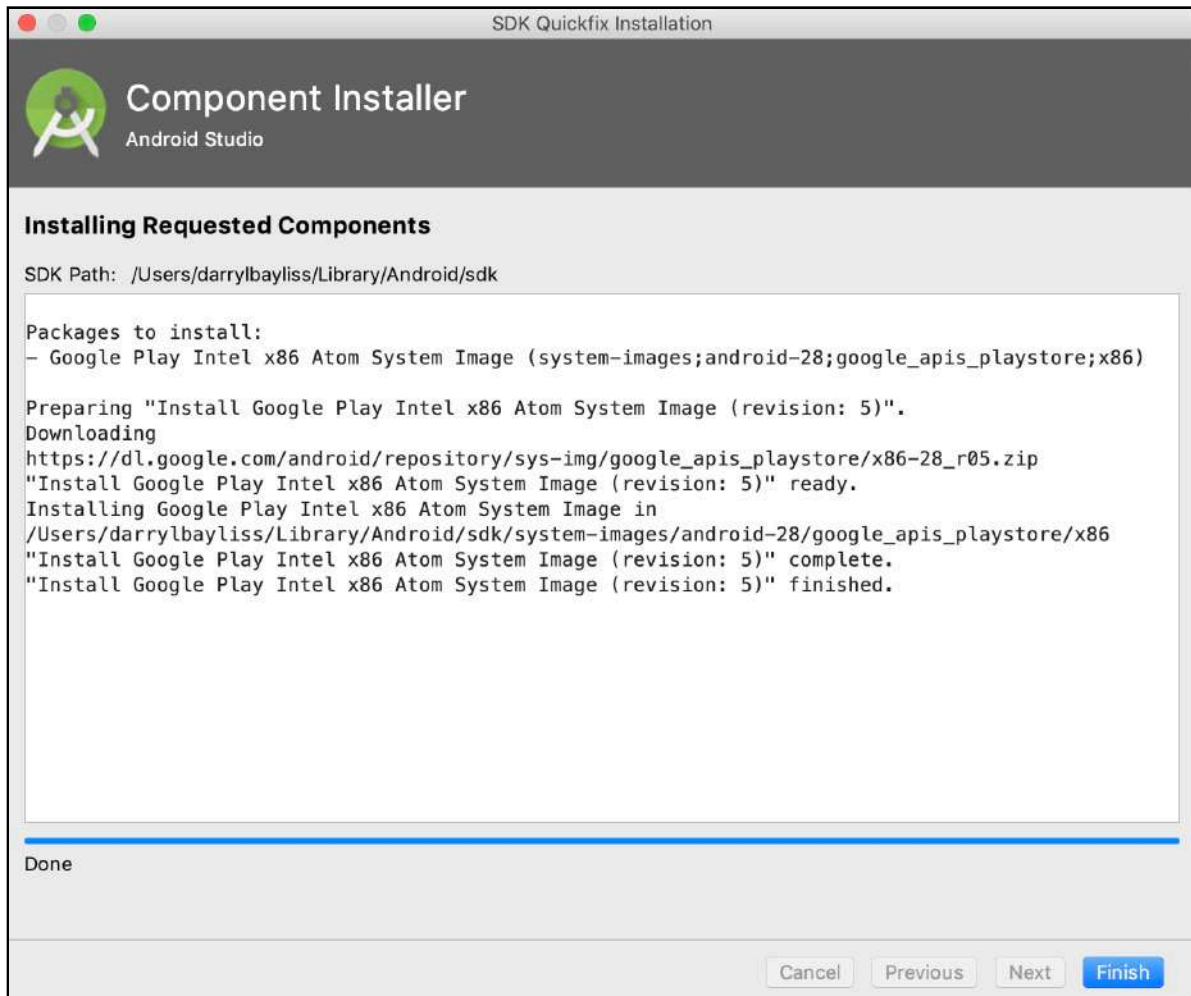
Once the download has finished, **Click** the Finish button in the bottom-right. The component installer window will disappear and the System Image window will appear again. At this point, your Android version is ready to use. To move on, **Click** the Next button in the bottom-right of the window.

The next and final window for creating your emulated device is a summary of the characteristics your device will have.
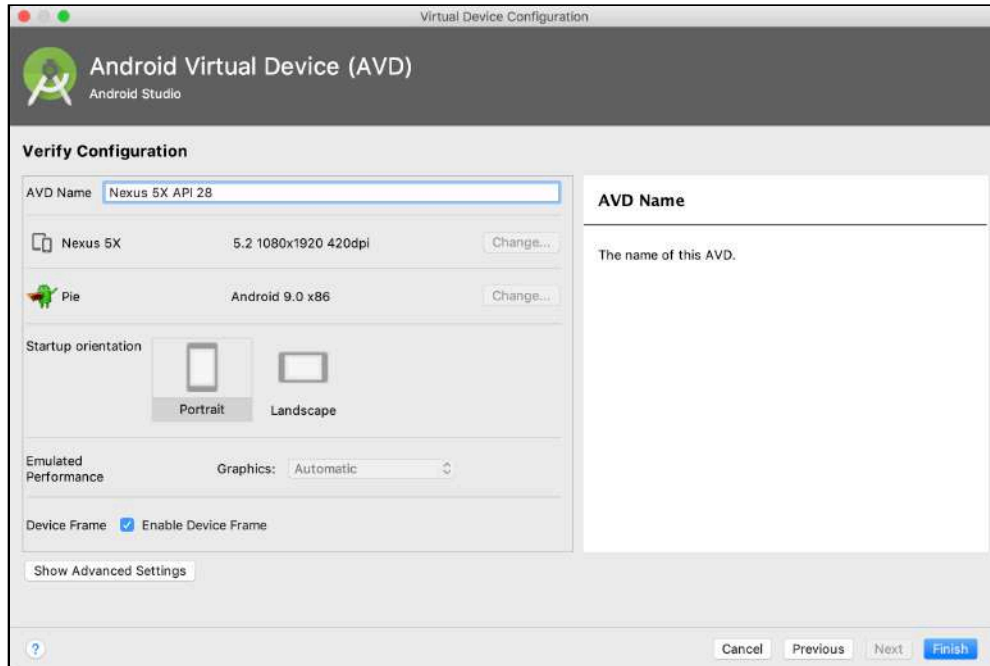
This window gives you the opportunity to give your AVD a name and to confirm other aspects of the device such as the Android version. You don't need to do anything here, so **Click** Finish at the bottom-right of the screen to create your AVD.

The current window will disappear. In the original AVD window that listed all available AVDs, you'll see your freshly created AVD ready for use:
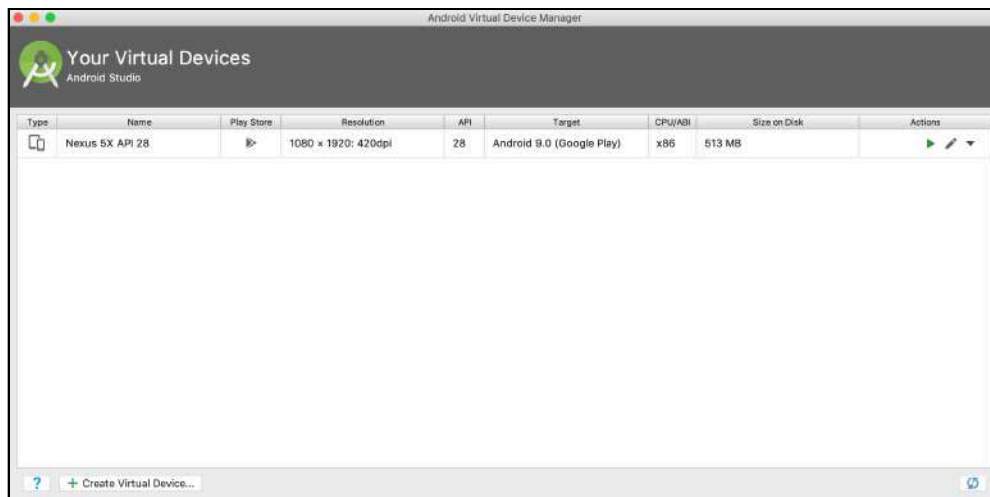
Setting up an Android device

Note: If you don't have an Android device to use for development, read the previous section on how to set up an Android Virtual Device.

One of the joys of Android development is having your app working on your own device to show to your friends. But before you can install Timefighter onto your device, you need to get your device up for use with Android Studio. The first thing to do is to connect your Android device to your machine via a USB cable.

Note: If you're using a Windows machine, you'll need to download a USB driver for your device first. You can download the driver and find instructions on installing it at https://developer.android.com/studio/run/oem-usb.html.

On your device, open the **Settings** app.
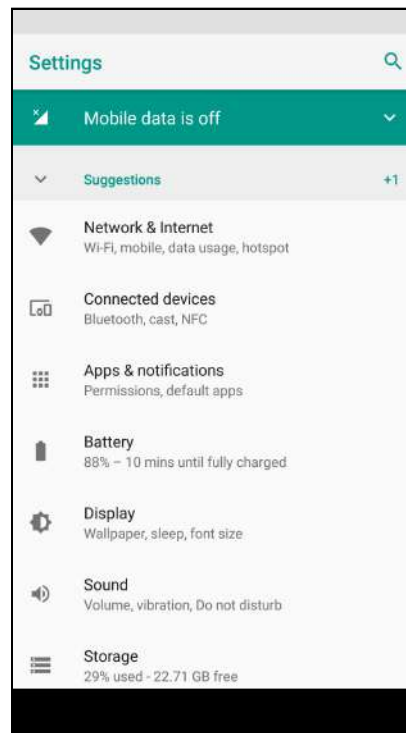
Note: If your device is running Android 8.0 (Oreo) or above, you need to tap **System** first to find the **About Phone** section.

Scroll through the settings until you find the **About Phone** button and tap it.
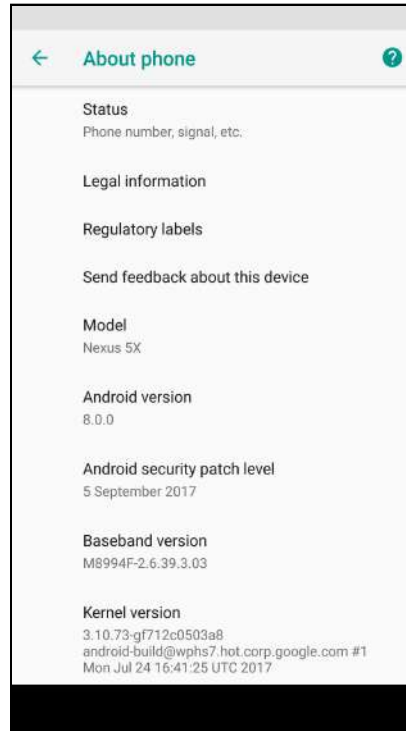
Now for the magical part! Scroll to the bottom of the **About Phone** screen, until the build number item appears:
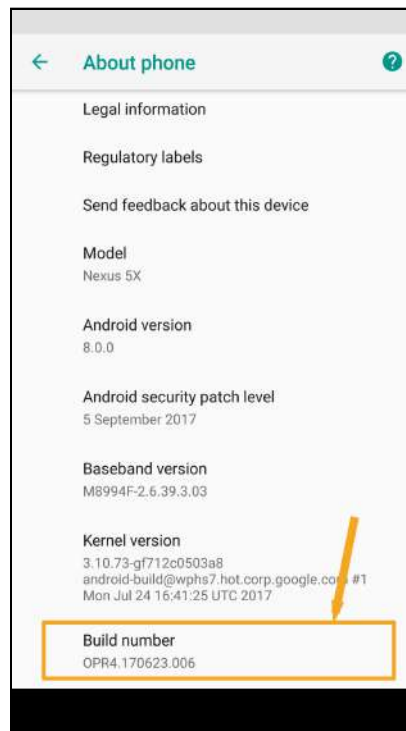
When you see the build number item, tap it several times until you see a toast message appear, informing you are a few steps away from being a developer. Keep tapping away until you get another toast message telling you that you've become a developer.

> **Note**: If your device is locked with a PIN, you will need to enter it first before you can reach this stage.

So what did this magical button do? Tap the Back button to go to the previous Settings page. Notice anything different?
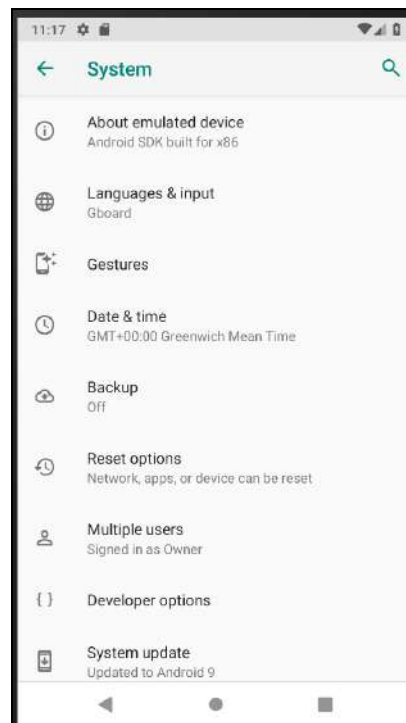
A new item has appeared called **Developer Options**! Tap the option to check out all the developer features available to you.
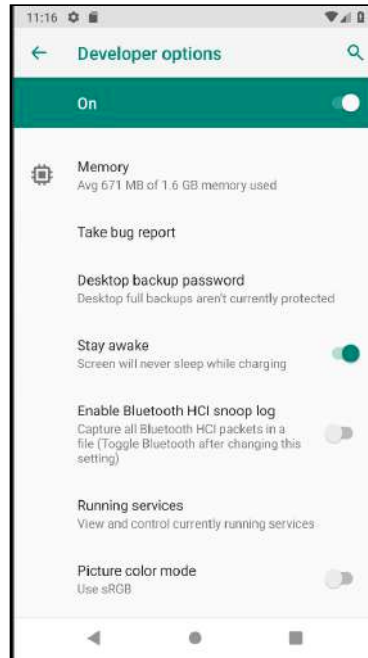
There's a lot, here, but there is only one option you need right now: **USB Debugging**. Scroll down to the option and toggle it on. A dialog will appear informing you of the intended usage of USB debugging.
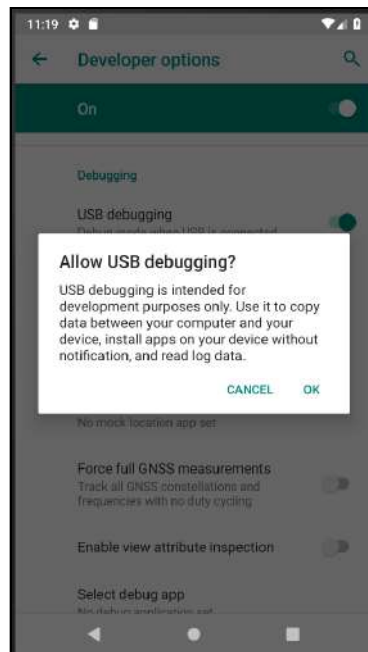
Granting **USB debugging** privileges is a potential security hole, so most devices have this turned off by default. Since you will need to install apps over USB as a developer, you need to turn this on.

When you are ready, tap **OK** and the USB Debugging toggle will enable.
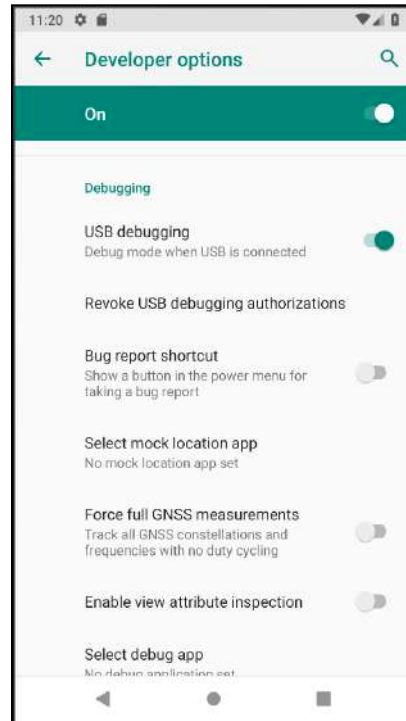
Congratulations: your device is now set up for development!

Running the app

It's time to run Timefighter! Along the top of Android studio, there is a button that looks like a green Play button:

Click the button, and a new window will appear over Android Studio.
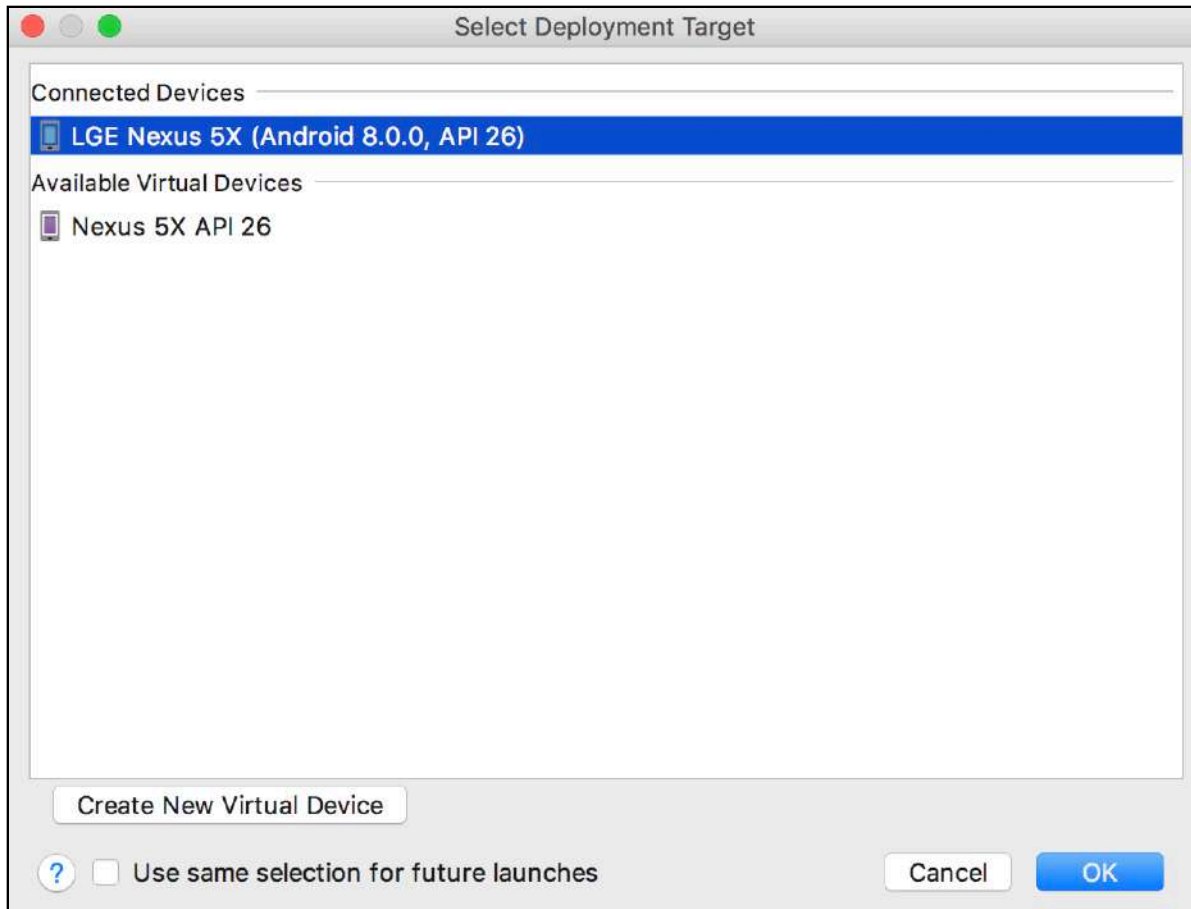
This is the **Select Deployment Target** window, showing you all available devices and emulators you can use to run your app.

If you have a physical device with developer mode enabled, this will appear in the **Connected Devices** section. If not, then the emulator you set up will be available in the **Available Virtual Devices** section. If you have both, then well done for tackling both sections!
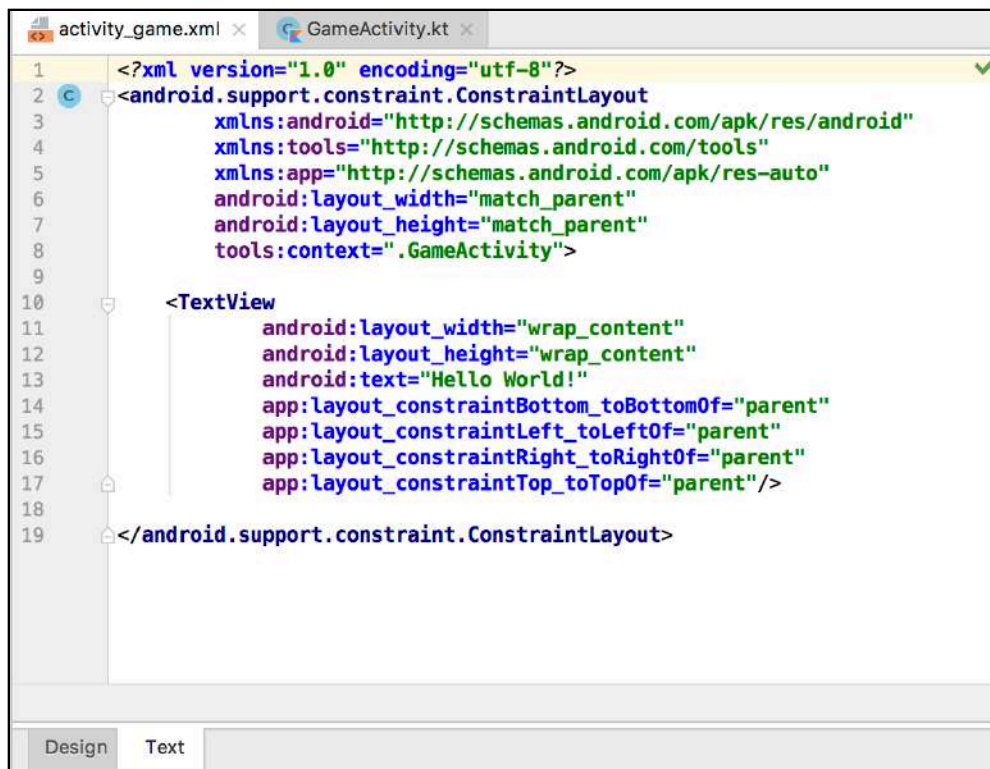
Select either of the devices available and click OK on the bottom-right of the window. Android Studio will begin building Timefighter and installing the built app on your device. You can see this happening at the bottom of Android Studio.
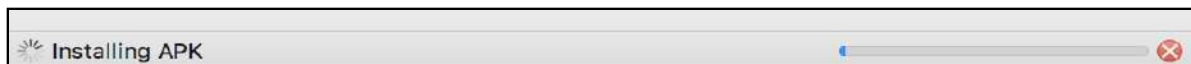
When Android Studio finishes, Timefighter will appear on your device:
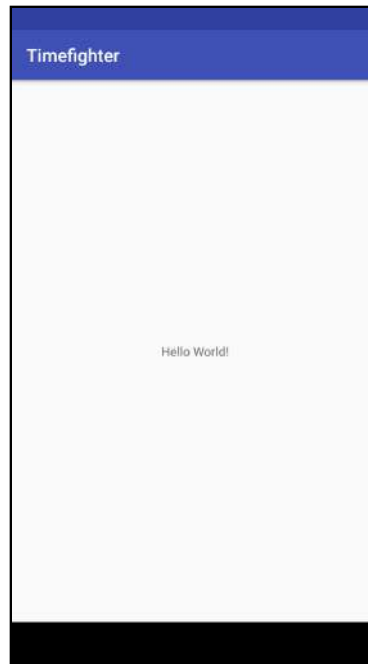
You've just built your very first Android App! To celebrate, let's make it a little more personal. Head back to Android Studio, and in the project navigator open **app ▸ res ▸ layout ▸ activity_game.xml**, then switch to the `Text` tab on the bottom.

Don't worry too much now about everything you see in this file. All you need to know is that it represents the app screen that appeared on your device earlier. You'll learn more about this in the next chapter.

For now, inside the **TextView** tag, update `android:text` property to greet you with your own name:

Click the green Play button again to run your app:
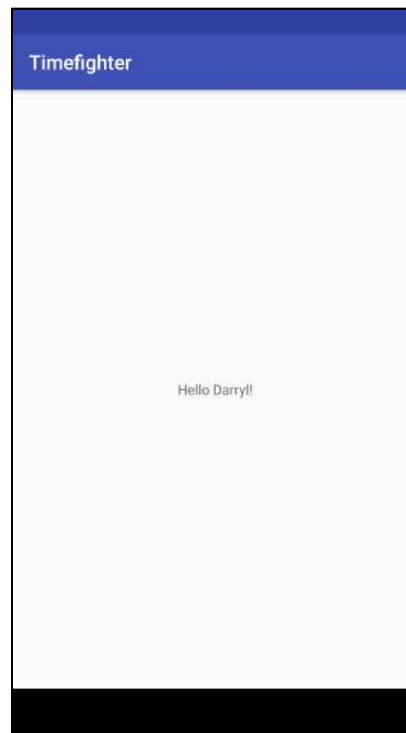
Installing new versions of Android studio

This book was written assuming a specific version of the Android SDK; as of this writing, we used Android Pie API version 28 as a baseline. However, you might be reading this book long after that version has been superseded. In that case, you may need to install the latest versions of Android Studio and the Android SDK.

> **Note**: Google engineers have decoupled Android Studio from versions of Android. This means you can build apps in Android Studio with any version of the Android operating system you want, including any future versions of the Android SDK.

Android Studio will do its best to prompt you when new versions of either Android Studio or Android SDK are available; however, you don't have to wait on Android Studio to do that for you.

In the **Android Studio** menu, selecting **Check for Updates** will give you a dialog with all things that can be updated on your machine, or which lets you know you're up to date already.

If you'd like to download a newer (or older) version of the Android SDK, in the same menu, select the **Preferences...** menu item.
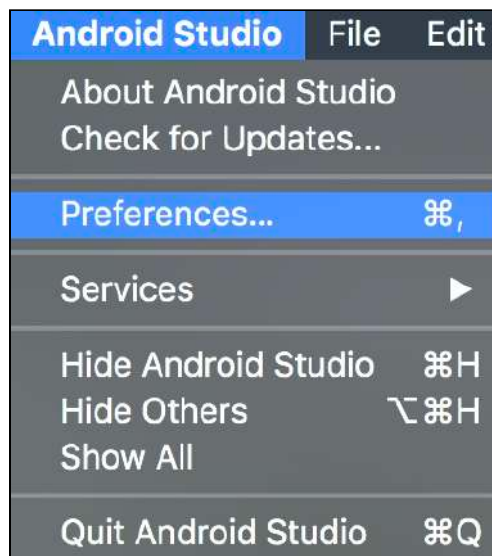
In the **Preferences** dialog, drill down through the menu items in the tree to **Appearance & Behavior ▸ System Settings ▸ Android SDK**.

In this window, there are two tabs to note: **SDK Platforms** and **SDK Tools**. In **SDK Platforms**, you should see a list of all the available Android SDK.
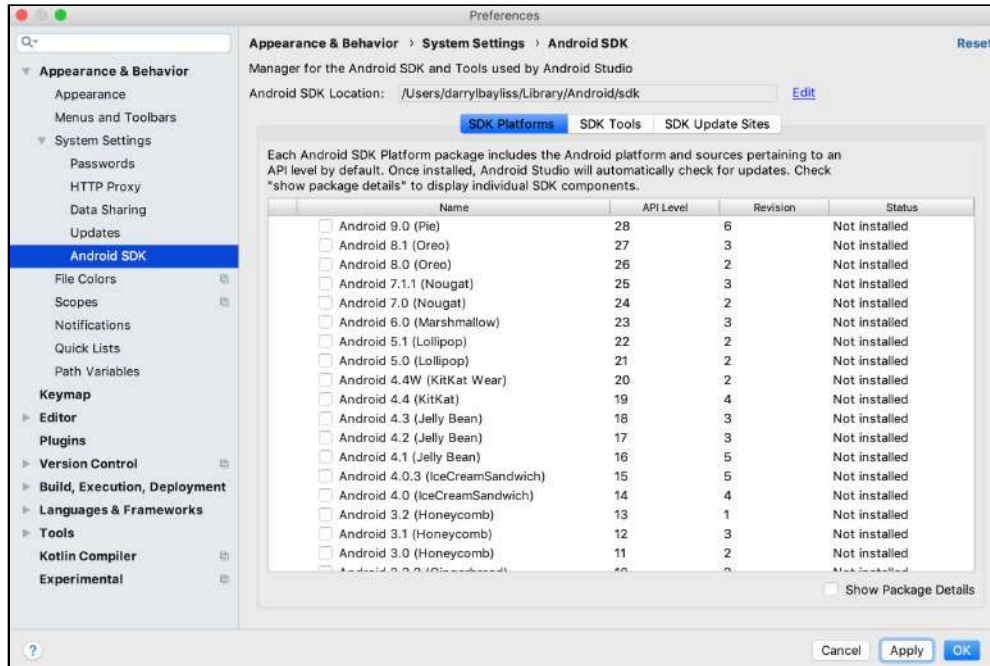
Clicking on any one of the SDK in the list, and then clicking **OK**, will install that SDK.

In **SDK Tools**, you will see a list of all the available build tools that Android Studio and your app have access to.
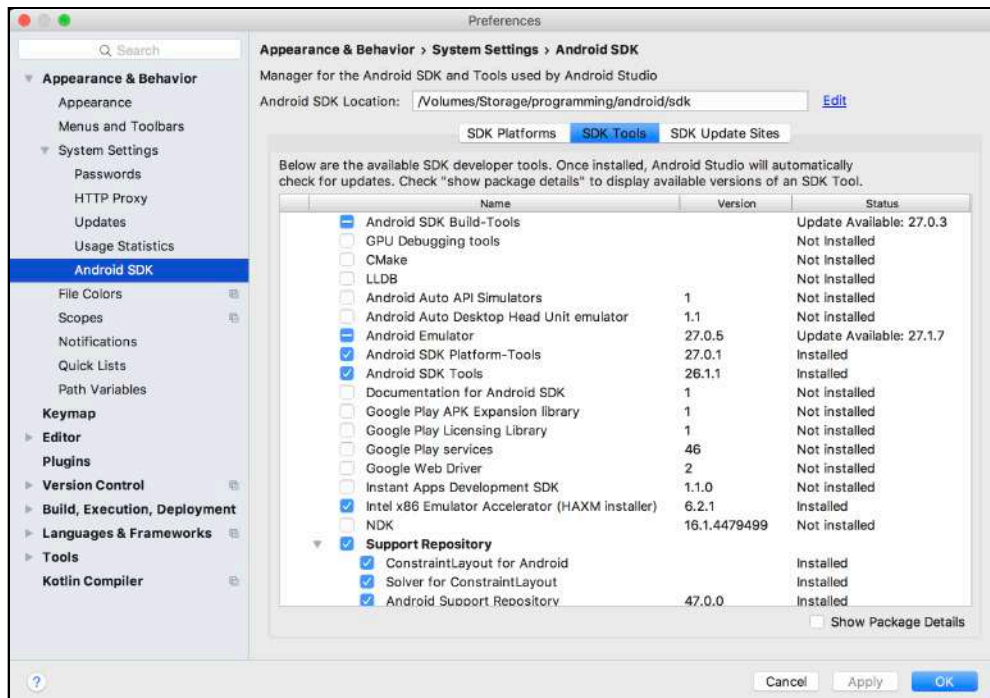
Clicking on any one of them, and then clicking **OK** will install that SDK.

At this point in the book, don't worry too much about why you would need to download any of these items. You'll learn more about each of these topics throughout the book. However, it's worth seeing these things now so that you'll recognize them as you encounter them in later chapters.

Where to go from here?

Well done getting your first app up and running! This is just the beginning; the next few chapters in this section will teach you even more about the basics of Android development. As you work through the chapters in this book, you'll end up with a fully featured app! Head on into the next chapter to start building out your app!

Chapter 2: Layouts

By Darryl Bayliss

If bricks and mortar are the foundation of a sturdy building, then **Layouts** are the Android equivalent of a sturdy app. Layouts are incredibly flexible and let you define how your user interface is presented on the device to the user.

You can create layouts in one of two ways:

1. Using an XML file that lets you declare your user interface ahead of time.

2. Writing Java code to programmatically create your layout when your app runs.

In this book, you'll define your layouts in XML. This is because Android Studio has a powerful Layout editor that covers 90% of the cases you'll ever need when creating a user interface.

Getting started

Before diving into the wonderful world of layouts, take a moment to think about what makes an app. Most often, your app will be a self-contained program that lets you perform one or more tasks.

You also want your user to accomplish those tasks quickly and intuitively, which is where a well-thought-out user interface comes into play.

The app you'll build in this section, TimeFighter, is no different. The app is quite minimal in its design, so usability won't be an issue.

Your first task is to set up your user interface, which has two **TextViews** and a **Button**.

Locate the **projects** folder for this chapter and open the TimeFighter app under **starter**. The first time you open the project, Android Studio takes a few minutes to set up your environment and update dependencies.

These are not the SDKs you're looking for

When you open the project you may get the following error in the **Messages** tab:
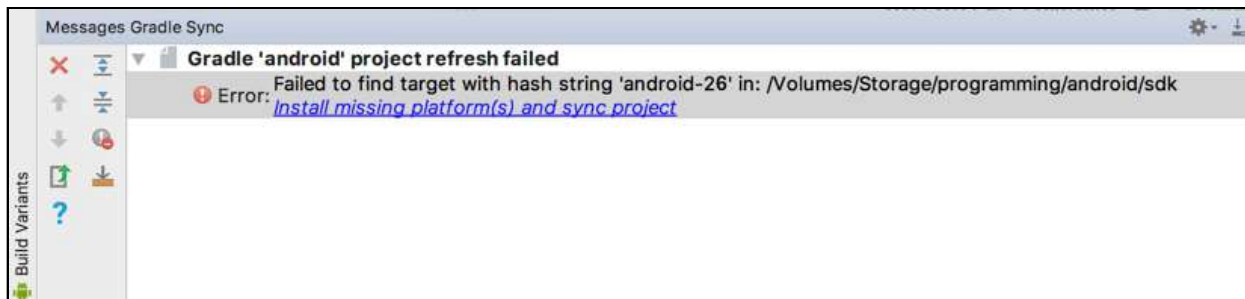
If you followed along with Chapter 1, and installed a fresh version of Android Studio, you may not see this error, however, if you're already running Android Studio, it could be that you don't have the version of the Android SDK that was used to create this book available on your machine. Do not fret young padawan learner, as Android Studio will always do its best to help resolve these sorts of issues for you. As you can see, Android Studio provided you with a convenient link, with a single click will install the required version of the Android SDK, and rebuild your project.

Once this error goes away, you should see the following in the **Messages** tab:
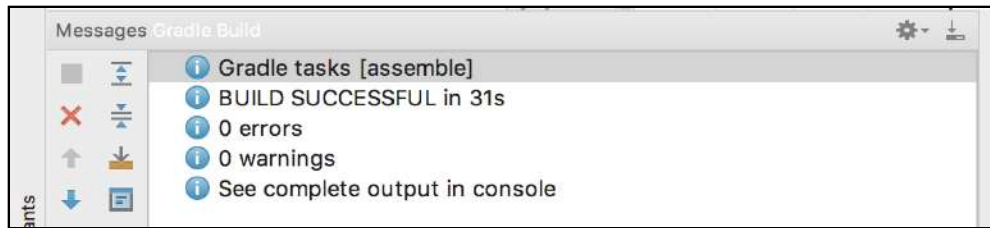
Everything is right with the world again! Continue on to the next section and get comfy with everything Android Studio has to offer.

The Visual Editor

In the project structure sidebar on the left of Android Studio, expand the **app**, **res**, **layout** folder and double click on **activity_game.xml**. You'll be presented with a screen that looks like this:
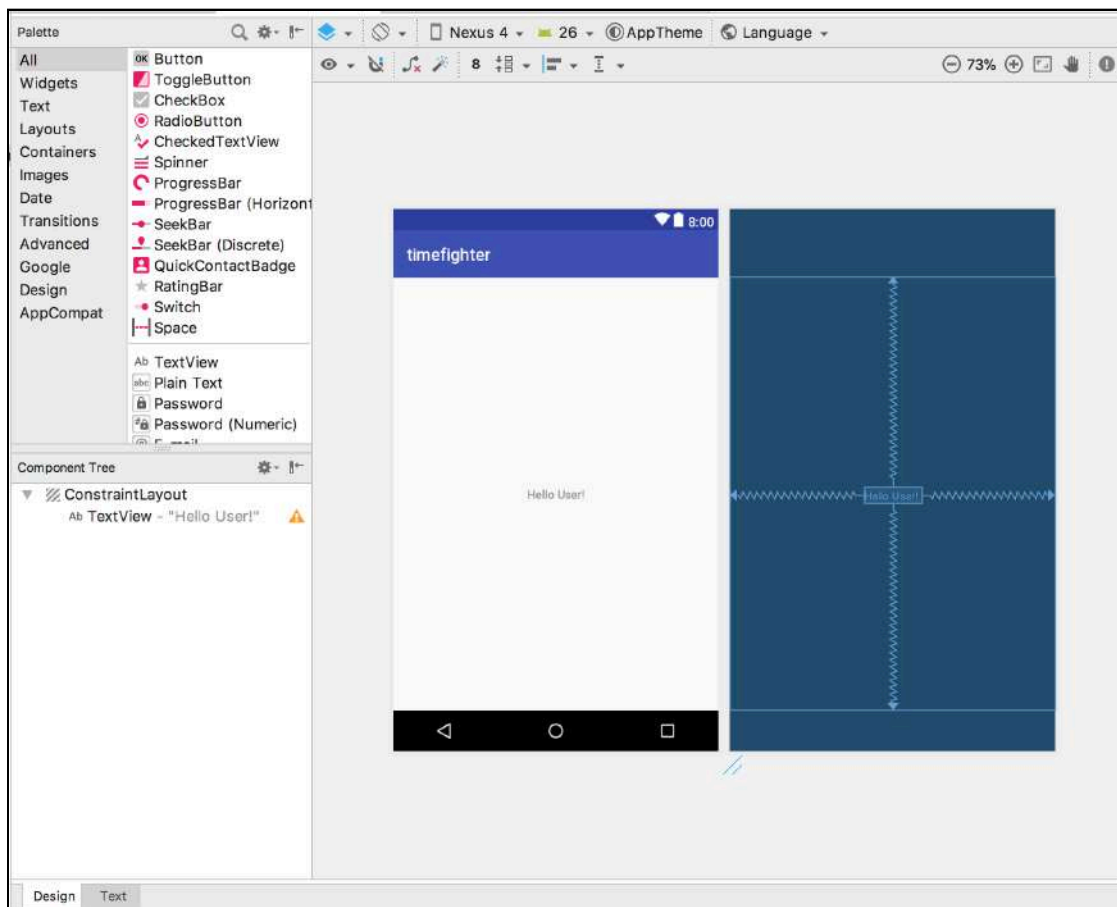
Editing activity_game.xml in Visual Editor

Behold the **Visual Editor**! In **design mode**, the middle of Android Studio will show a few different screens.

The first screen of interest is the preview area in the middle, next to what looks like a blueprint. This is where you'll begin to build your user interface.

At the bottom of the visual editor, you'll find two tabs labelled **Design** and **Text**. Click **Text**, and you'll be presented with a screen that looks like this:
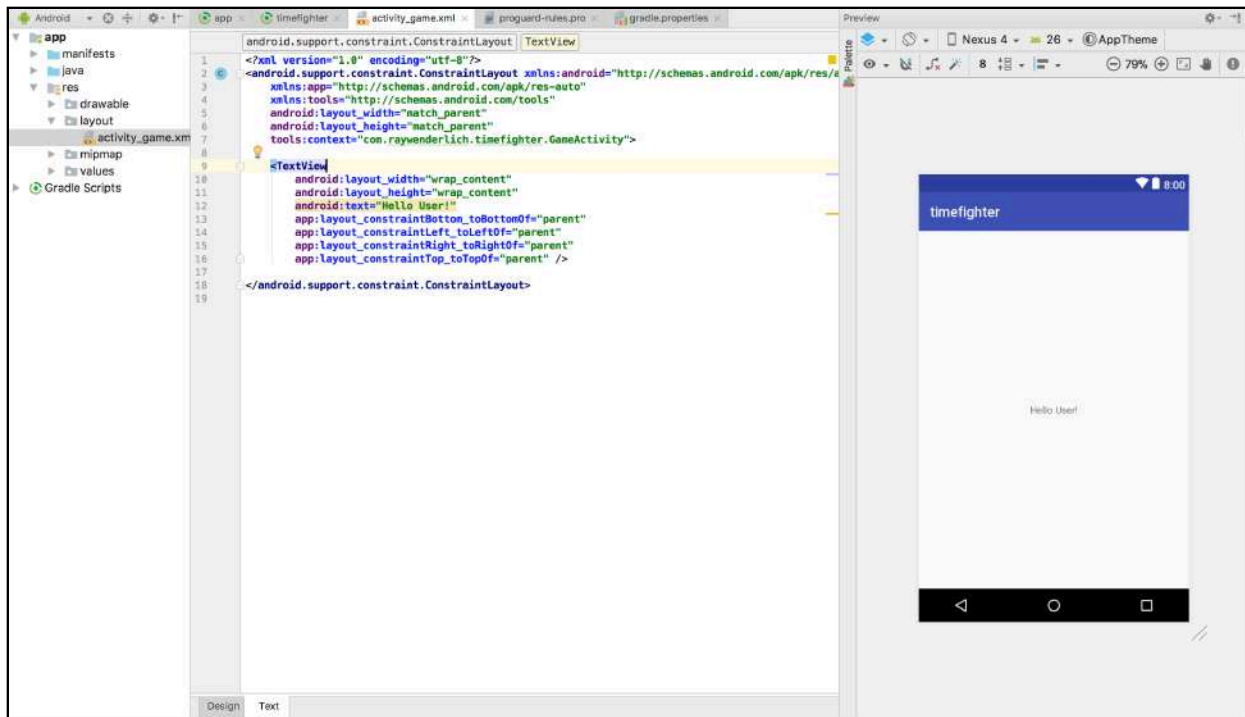
Editing activity_game.xml in XML Editor

In the middle section of Android Studio, you'll find the **Text Editor**. This shows the XML representation of the first screen of your app. You can create your interface here if you like — but using the Design tab is much more visual to begin with.

Click the **Design** tab to switch back to design mode. You'll start by adding a **TextView** to the user interface.

In the top left of the middle section of Android Studio, you'll see the **Palette**:
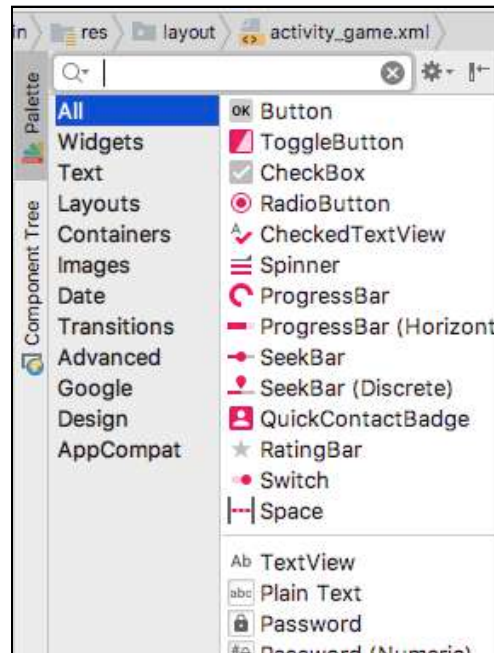
Palette of interface components

This contains all the built-in user interface components you can use to build the screens of your Android app.

What's even more useful is that you can drag and drop from this palette straight into the preview screen to add a component.

Head over to the Palette and select **Text**. The palette will change its offerings to everything text-related for you to peruse.

Next, drag a **TextView** from the palette for your score label and drop it in the top left of the preview screen. You should end up with something like this:
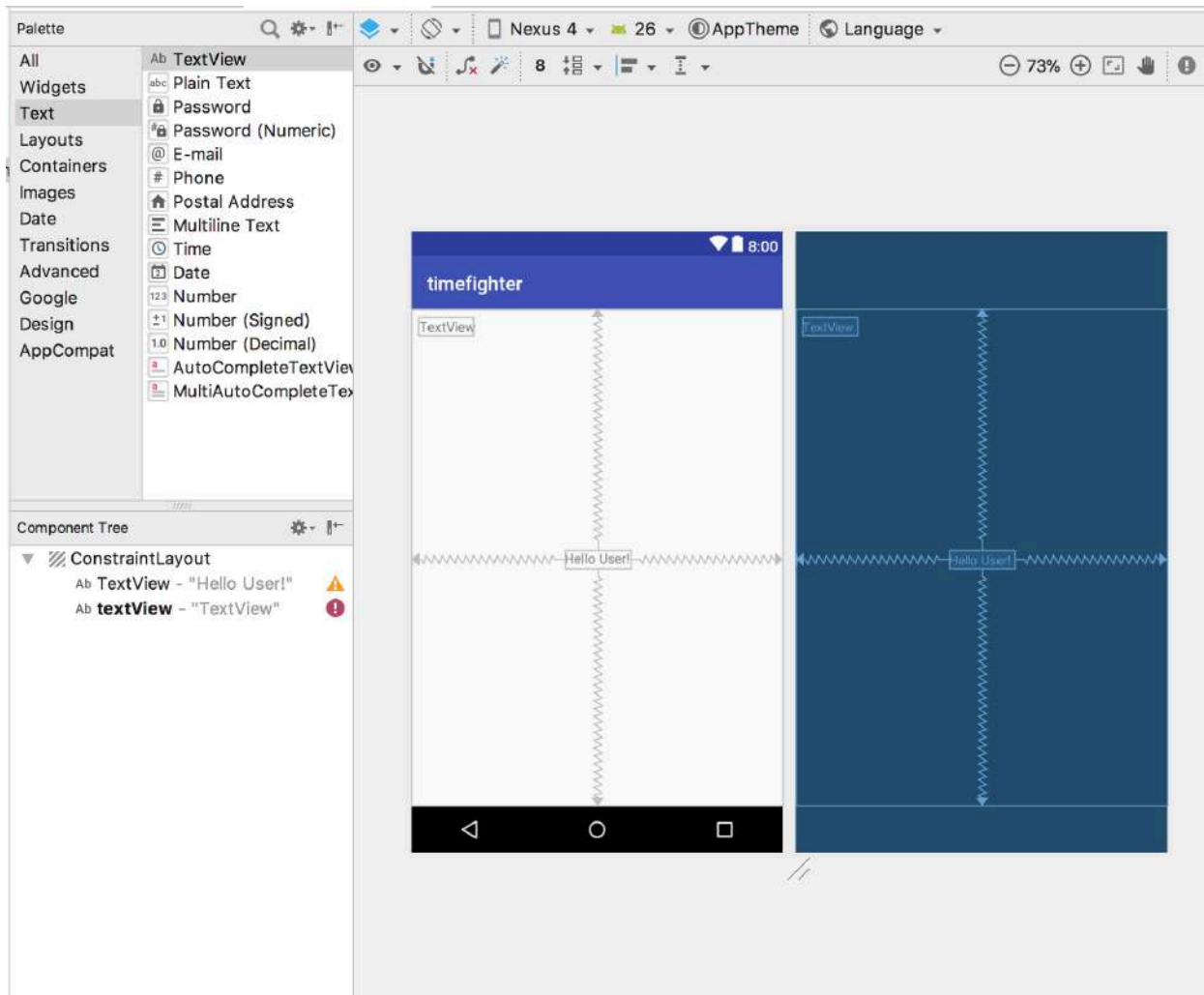
That was easy enough!

Component Tree View

Just before we move on, it's worth noting that although dragging and dropping components into the Preview area can be extremely refreshing and easy to do, when you're dealing with project's that have **lots** of views, it can be tricky to get it in the exactly the right spot inside the right parent view. As an alternative, you can drag components from the Palette directly into a **Component Tree** area, and drop it underneath the desired parent component.
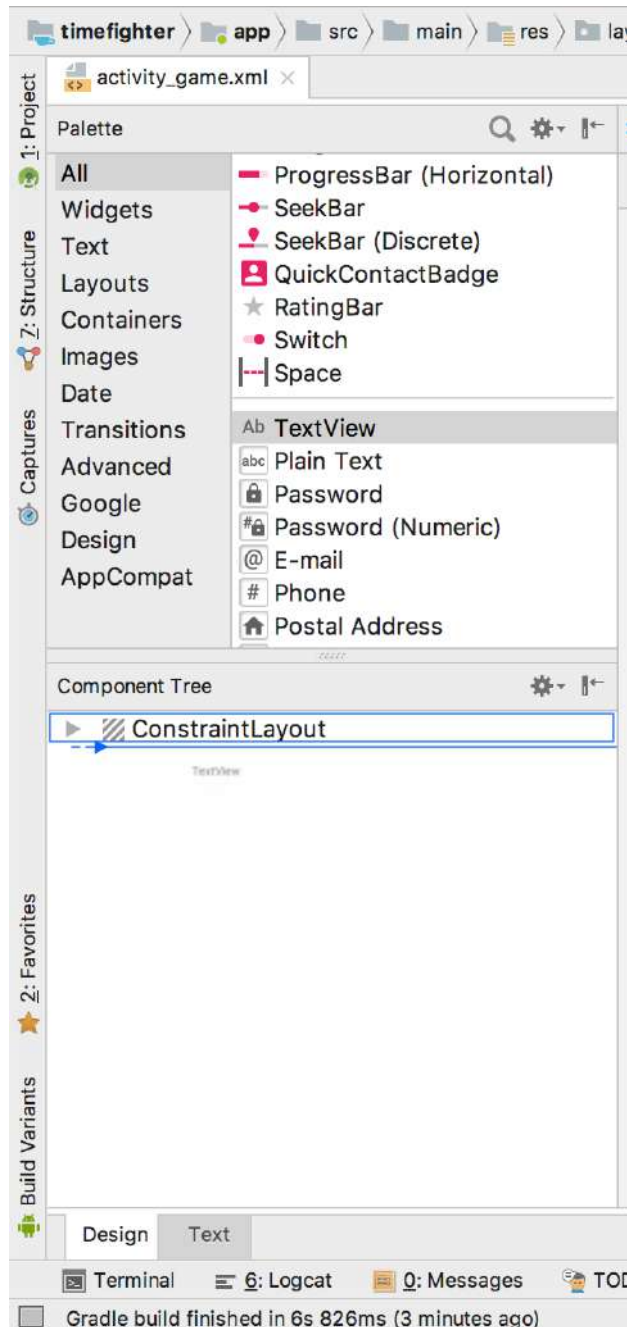
Keep that little feather in your hat as you progress through this book. You may find it easier to drop components this way, and then deal with positioning them later. Ok, with that little bit out of the way, we rejoin our originally scheduled programming already in progress!

Positioning your views

At this point, you have the start of your app, with your TextView sitting in the top left-hand corner.

Or is it?

How does the device know where to position that lonely TextView? What would happen if someone rotated the device into landscape mode?

In fact, the app doesn't know where to place your TextView — and you can prove it.

In the Visual Editor, drag that newly placed TextView somewhere in the middle of the screen like so:
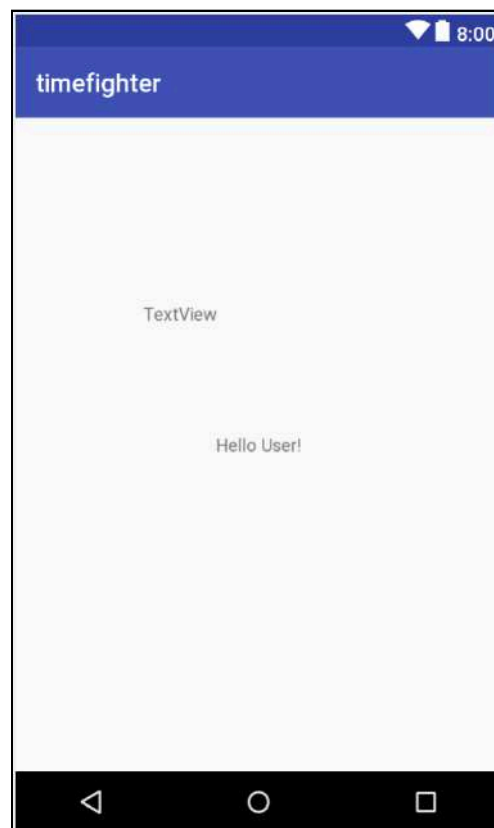
Layout as seen in the Visual Editor

Click **Run 'app'** in the top right of Android Studio and launch your Emulator. Once it loads you'll see this:
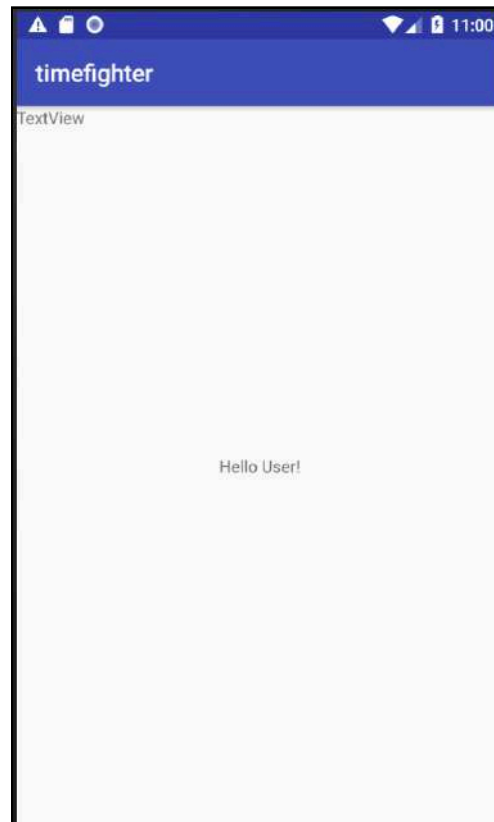
Layout as seen on device

That's not where you placed your TextView! Perhaps it really likes to hide in that corner?

Don't fret — in the next section, you'll make sure that TextView stays put.

Adding rules to your position

So why does the TextView not stay where you put it? The answer is you need to give the TextView some *rules* on where it should be on the screen. There are millions of Android devices out there that come in all shapes and sizes.

To make your app look great on all those different screens, you need to do a little layout work.

The blueprint screen to the right of the preview gives you a visual representation of all the rules that exist within your layout. You'll use this tool to create new rules for your TextView.

In the preview screen, click and drag your TextView to the top-left corner of the screen. Then hover your mouse over the left side of your newly placed TextView in the blueprint screen. A circle with a white outline will appear and a **Create Connection** bubble will pop up:
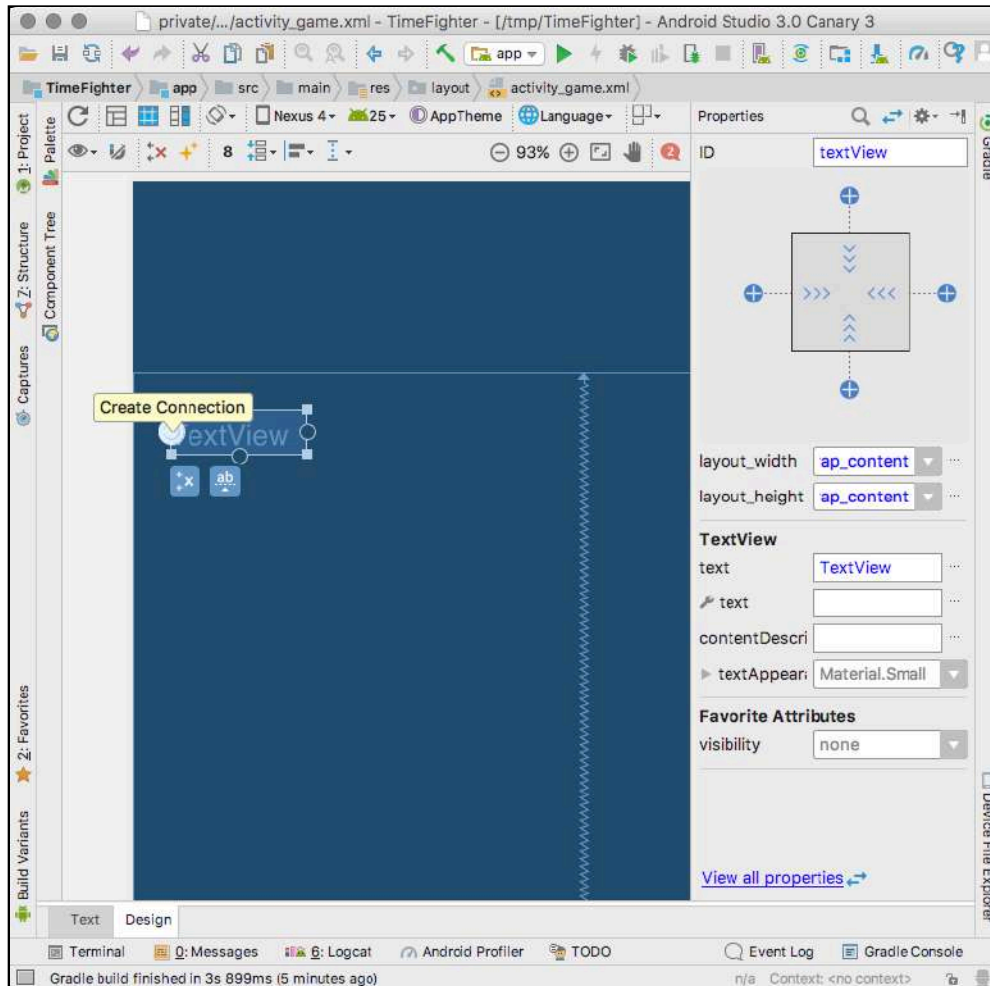
Click and drag towards the left edge of the blueprint screen, and you should see your TextView move slightly to the right. At this point, release your mouse button.

Congratulations — you just created your first layout rule!

Next, you'll need to create the top layout rule. Move your mouse to the top of the TextView until the outlined circle appears, and drag to the top edge of the screen until the TextView moves down slightly. Release the mouse button again to create your second layout rule.
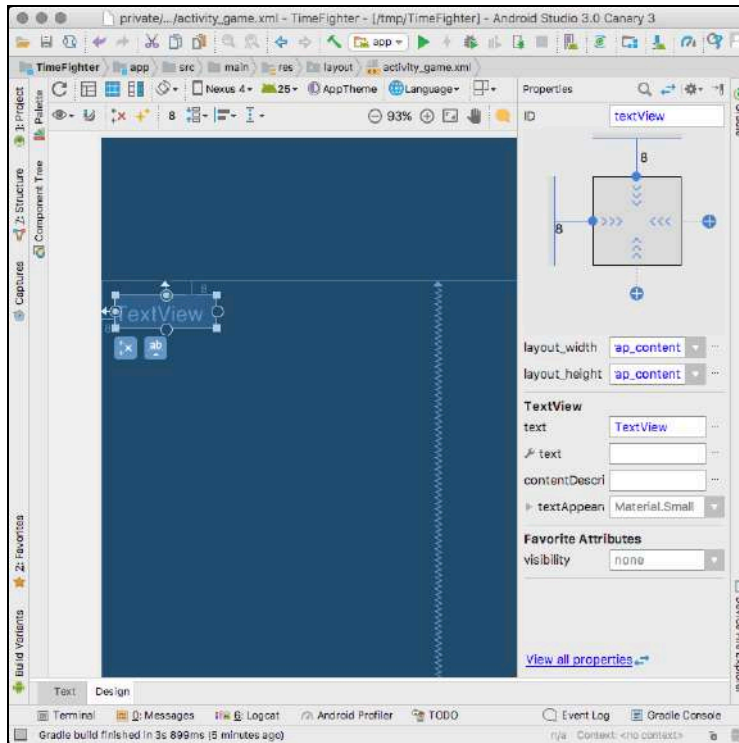
To see what's happened, make sure your TextView is selected, and look for a panel on the right side of Android Studio, in the **Properties** window. Look at the top of the properties window and you'll see a square with some chevrons inside:
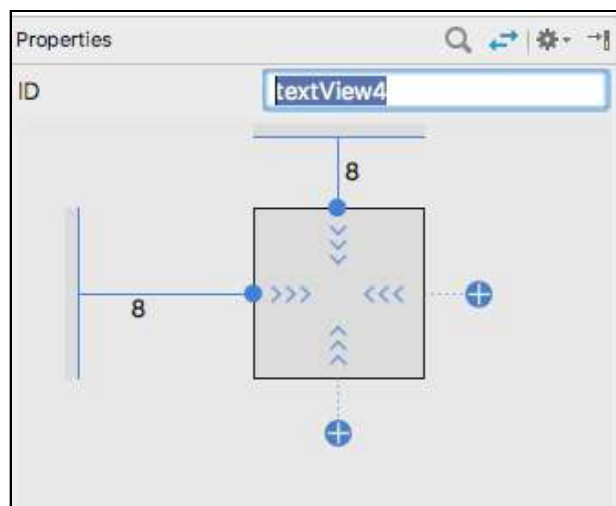
Layout rule for your TextView

If you look closely, you'll see two solid lines running from the left and the top of the rectangle, pushing against two grey rectangles with a number **8** floating beside them. These are the rules, or **constraints**, you just created that hold your TextView against the edges of your screen. They instruct your TextView how to position itself relative to the screen edge.

How Android does this isn't really important at this stage. But if you did want to more finely position this TextView, you can adjust the margins of your constraint by clicking the number beside the constraint line and selecting one of the preset numbers in the dropdown or entering your own. Useful to know!

Finishing the screen

Now that you're armed with the basic knowledge of how constraints work, you can finish off your screen.

Head over to the Palette window and drag another TextView into the preview window, this time putting it in the top right corner of the screen to serve as your time remaining label.

In the blueprint window, select your new TextView and hover over the right edge of it until the **Create Connection** bubble appears. Create a constraint against the right side of the screen. Then do the same for the top of the TextView against the top of the screen.

You should end up with something like this:
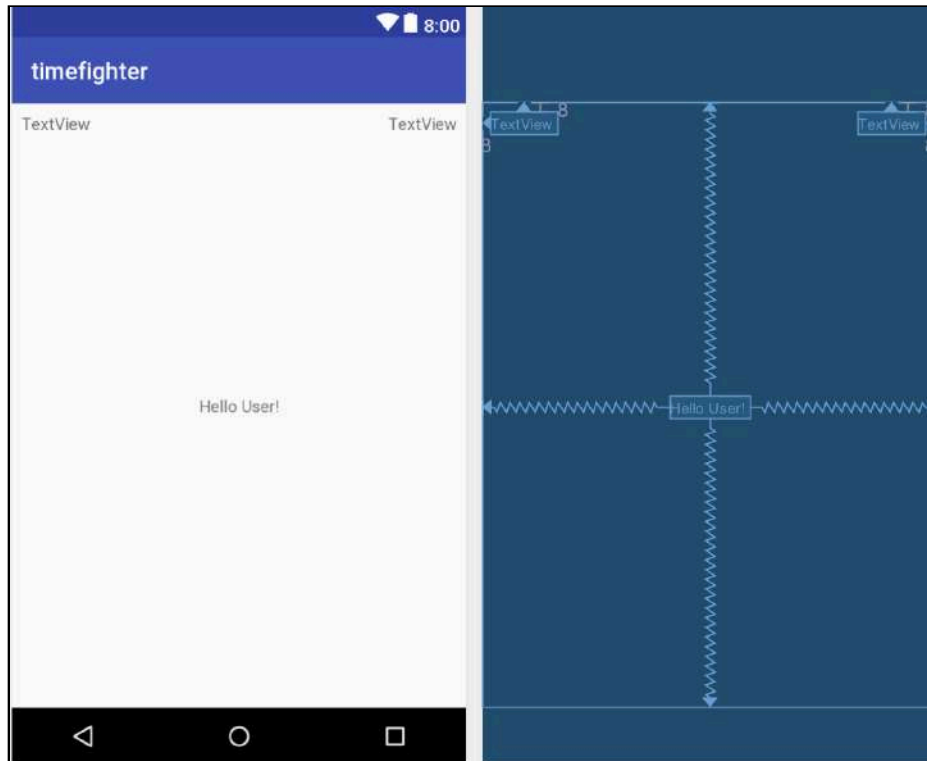
That takes care of your two TextViews. Now you need to add the "Tap Me!" button.

First, remove that TextView floating in the middle of the screen. Click on the "Hello User!" TextView, press the **delete** key and the TextView will disappear.

With the TextView removed, you can add the button. Head over to the Palette window and click the **All** tab. Once you see the **Button** in the Palette, click and drag it to the center of the screen. You may even see some helpful dotted guidelines to help you position your Button right in the center of the screen.

Now you need to create some constraints for the Button, just like you did for your TextViews. This button needs to stay in the center of the screen, so you're going to need four constraints, one for each side of the button.

In the blueprint screen, hover over each side of the Button and pull it towards its respective edge of the screen. The button will move around quite a bit as you do this, but don't panic! That's just the Button trying to respect the constraints as you add them.

Keep dragging each constraint to the edge of the screen and you'll end up with this:
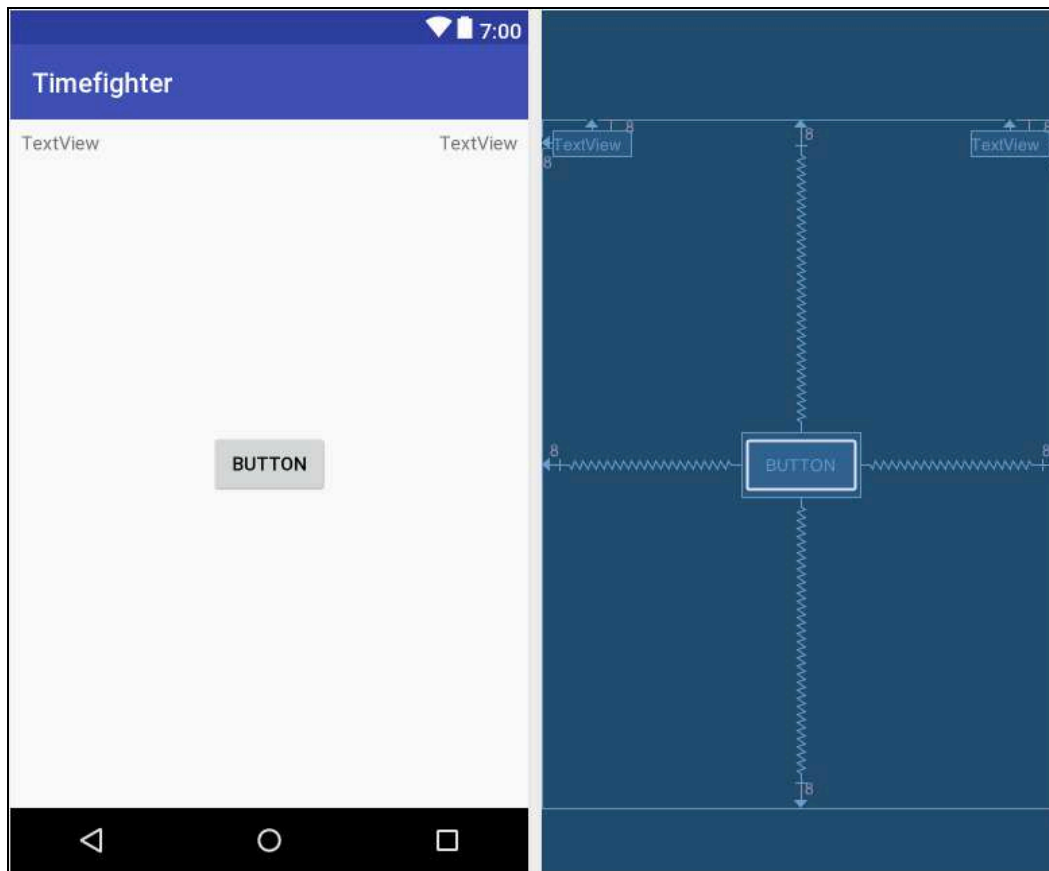
Finally click **Run 'app'** from the top menu in Android Studio, and your Emulator or device will load the latest changes to your app. Your hard work should be rewarded with an app that contains two correctly placed **TextView**s and a **Button**.

Where to go from here?

Nice job! Although you've learned a lot, you've only used a fraction of the power that Constraints offer. There is a dedicated component for Constraints — **ConstraintLayout** — that provides all this functionality.

There are other Layouts that provide other structures your Views can leverage, such as **LinearLayout** and **FrameLayout** among others. It's recommended to use a **ConstraintLayout** where possible however, there are times where it might be awkward or not practical.

This book uses ConstraintLayout as its go-to Layout choice. If you want to learn more, check out the documentation on ConstraintLayout on the Android Developer website at https://developer.android.com/training/constraint-layout/index.html.

Pat yourself on the back for making it this far! You've taken your first step into the world of Android development.

If you had any problems following along with the starter app, have a look at the completed solution in the **final** folder for this chapter's materials.

In the next chapter, you'll attach some logic to your button and make those TextViews display something more interesting than the words "TextView". You'll also get your first taste of writing code. See you there!

Chapter 3: Activities

By Darryl Bayliss

One of the things keeping us healthy and happy in life is a lifestyle comprised of varied activity. These activities could be a mixture of anything, but they all have a specific purpose or goal in mind.

Android apps are exactly the same — they're built around a set of screens that have a specific purpose. Each screen you see in an app is known as an **Activity**. An Activity is built around a single task that you want your user to perform.

For instance, in most apps you have a settings screen to adjust the settings of the app. There is a sign-in screen where you login using your username and password, and many other screens that have specific purposes.

In this chapter, you'll begin putting together an Activity focused around the gameplay for **TimeFighter** — and you'll finally get to lay down some code!

Getting started

Before you jump head first into code, you need to understand how IDs work. In Android, IDs play a fundamental role in connecting things such as Views back to the code.

In the previous chapter, you positioned views on screen and established that the top left TextView will show the score, the top right TextView will show the time and the Button, when pressed, increments your score. If your code wants to connect with these views, you'll need to give them IDs.

If you were following along with an app, open it and keep going with it for this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **TimeFighter** app under the **starter** folder.

The first time you open the project, Android Studio takes a few minutes to set up your environment and update dependencies.

Open **activity_game.xml** where you built your layout, and make sure you've selected **Visual Editor**. Next to the Palette tab, you should see a window called the **Component Tree**:
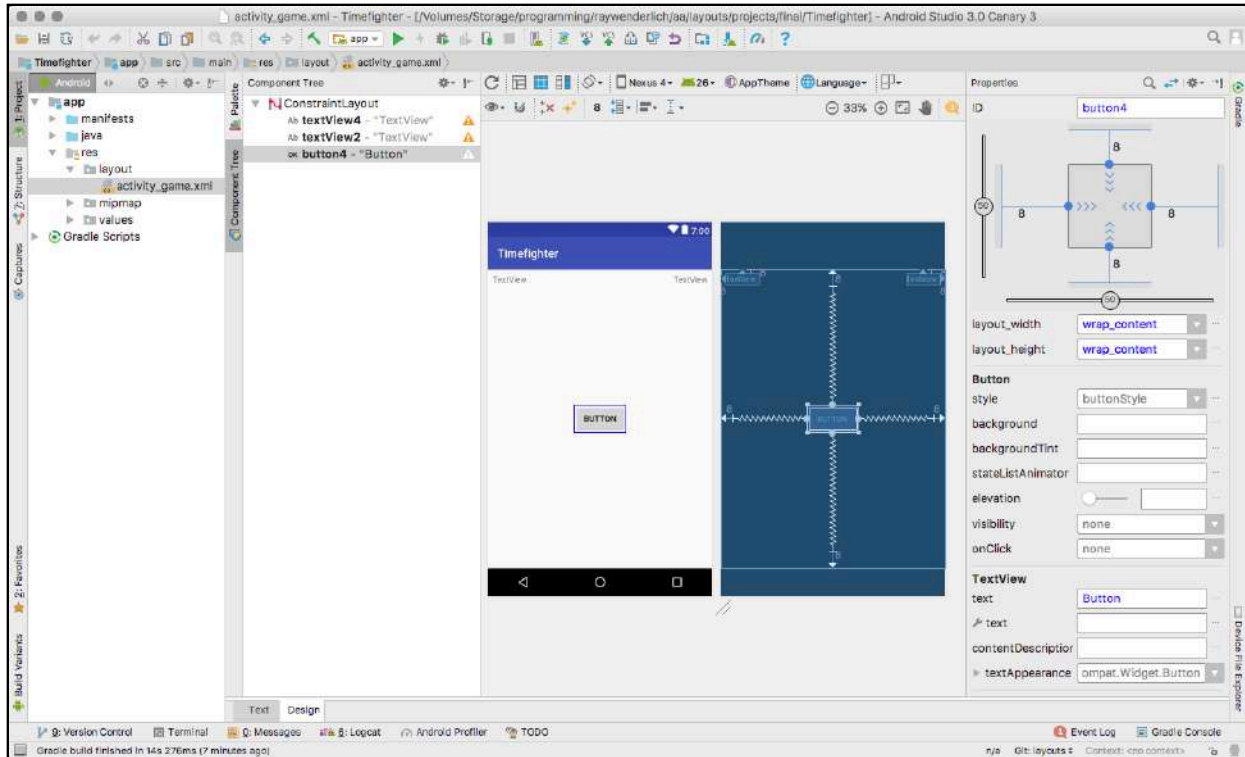
This window provides you with an overview of all the Views available in your layout and their relationship relative to one another.

Click on the row labeled `button` or `buttonX` where `X` is a number; this will highlight the **Button** in the middle of the screen, and update the Properties window on the right with details about the Button.
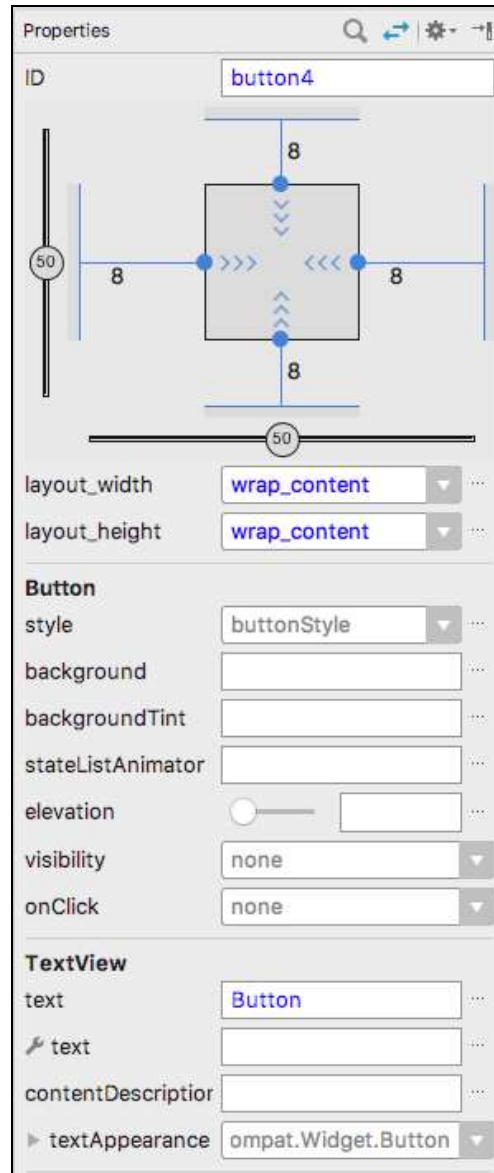
The button in the screen above already has an ID value of `button4` (and in your project it might have a different number or no number at all), but this isn't very descriptive. Theoretically, you could leave the ID as `button4`, but would that mean anything to you in a year from now? Probably not. Using descriptive IDs makes it easier to know which IDs refers to which Views.

Change the value of the **ID** field from `button` (or whatever it is in your project) to `tap_me_button`.

It would also be nice if the Button could have something more interesting than the word "Button". Change the value of "text" in the `TextView` section of the Properties window to "Tap me!".
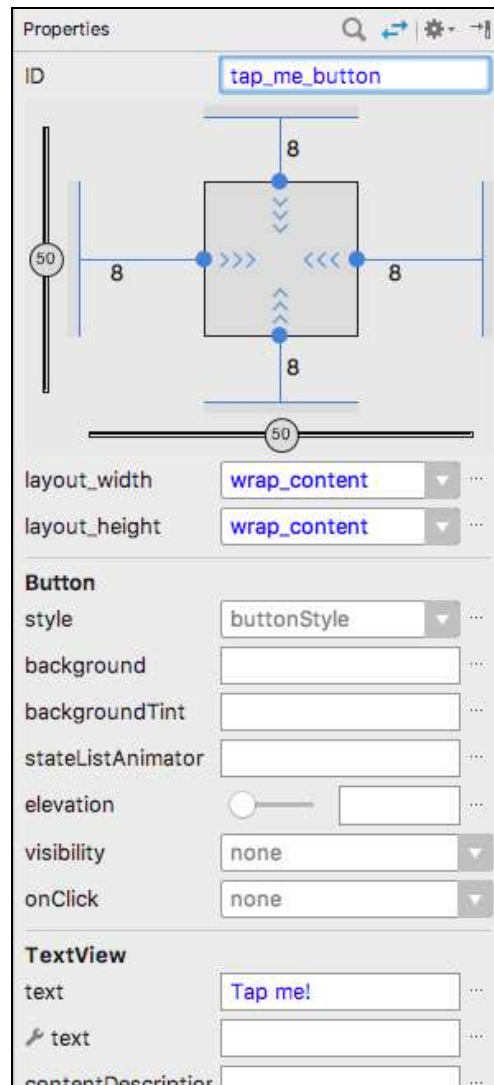
Select the text view on the top left from the **Component Tree**, set its ID to
`game_score_text_view` and change the text to "Your Score: %s". Finally, select the text
view you added to the top right, and change its ID to `time_left_text_view` and its text
to "Time left: %s".

What's the deal with the `%s` in the text you added? That's a placeholder for any string
you want to inject into your text values. You'll fill in those placeholders later.

At build time, Android Studio will take these IDs and turn them into constants your
code can access through what's known as the `R` file. You will see more about `R` files in
the upcoming sections, but simply know that Android will take an ID such as
`game_score_text_view` that you assigned to your view in your layout and create a
constant named `R.id.game_score_text_view`, which can then be accessed by the code.

Run your app now in your Emulator or on a device, and you'll see these text changes reflected on the screen:

Now that all the views in the project have IDs, you can finally start exploring and understanding your first Activity.

Exploring Activities

In the project navigator on the left, ensure the app folder is expanded. Navigate to the **GameActivity.kt** file, which is found in the **src/main/java/ com.raywenderlich.timefighter** folder. Open the file and you'll see the following contents:

```kotlin
package com.raywenderlich.timefighter

import android.support.v7.app.AppCompatActivity
import android.os.Bundle

// 1
class GameActivity : AppCompatActivity() {
  // 2
  override fun onCreate(savedInstanceState: Bundle?) {
    // 3
    super.onCreate(savedInstanceState)
    // 4
    setContentView(R.layout.activity_game)
  }
}
```

This is where the logic for your game screen will live. It doesn't do much at the moment, but take a moment to explore what it does:

1. `GameActivity` is declared as extending `AppCompatActivity` and is your first and only Activity in this app. What `AppCompatActivity` *does* is isn't important right now; all you need to know is that subclassing it is required to deal with content onscreen.

2. `onCreate` is the entry point to this Activity. You can see that it starts with the keyword `override` meaning you'll have to provide a custom implementation from the base `AppCompatActivity` class.

3. Calling the base's implementation of `onCreate` is not only important — it's required. You do this by calling `super.onCreate`. Android needs to set up a few things itself before your own implementation executes, so you notify the base class that it can do so at this point.

4. This line takes the layout you've created and puts it on your device screen by passing in the identifier for the layout. Android Studio generates the identifier in the `R` file at build time using the layout file name created in the previous chapter. So if you had a layout called `really_good_looking_screen` then the identifier generated would be `R.layout.really_good_looking_screen`.

These four lines are core to creating Activities in Android. You will see them in every activity you create. In the most general sense, any logic you add must come after you've called `setContentView`.

> **Note**: `onCreate` isn't the only entry point available for Activities, but is the one you should be most familiar with. `onCreate` also works in concert with other methods you can override that make up an Activity's *lifecycle*. We'll cover a number of those lifecycle methods in this book, but if you're curious, you can find out more over at https://developer.android.com/guide/components/activities/activity-lifecycle.html.

Replace the entire contents of **GameActivity.kt** with the following skeleton:

```kotlin
package com.raywenderlich.timefighter

import android.os.Bundle
import android.os.CountDownTimer
import android.support.v7.app.AppCompatActivity
import android.widget.Button
import android.widget.TextView
import android.widget.Toast

class GameActivity : AppCompatActivity() {
  internal lateinit var gameScoreTextView: TextView
  internal lateinit var timeLeftTextView: TextView
  internal lateinit var tapMeButton: Button

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_game)

    // connect views to variables
  }

  private fun incrementScore() {
    // increment score logic
  }

  private fun resetGame() {
    // reset game logic
  }

  private fun startGame() {
    // start game logic
  }

  private fun endGame() {
    // end game logic
  }
}
```

This contains a number of placeholder functions. You'll explore the purpose of each one in this chapter; however, this skeleton gives you an overview of the things you'll need to complete this time fighting app!

> **Note**: Sometimes when using new objects in your Classes, Android Studio will not recognize them until you import the class definition. This is shown by Android Studio highlighting the object in red.
>
> To import the class definition on a Mac, click on the object and press Alt + Return. For Windows or Linux, press Alt + Enter.

You can also choose to let Android Studio handle imports automatically for you when pasting code. On a Mac, select **Android Studio ▸ Preferences ▸ Editor ▸ General ▸ Auto Import** from the top menu. Set the **Insert imports on paste** dropdown to **All**. Finally, tick the **Add unambiguous imports on the fly** checkbox.

To do this on Windows or Linux, select **File ▸ Settings ▸ Editor ▸ Auto Import** from the top menu. Set the **Insert imports on paste** dropdown to **All**. Finally, tick the **Add unambiguous imports on the fly** checkbox.

Hooking up the views

As an Android developer, one of the most common things your app will do is react to a button click and convert that click into a change reflected in the app.

In **GameActivity**, you added three variables named `gameScoreTextView`, `timeLeftTextView` and `tapMeButton`. The first thing you need to do is attach these variables to the views you added to the layout.

In `onCreate(savedInstanceState: Bundle?)`, add the following code right after `setContentView`:

```
// 1
gameScoreTextView = findViewById<TextView>(R.id.game_score_text_view)

timeLeftTextView = findViewById<TextView>(R.id.time_left_text_view)

tapMeButton = findViewById<Button>(R.id.tap_me_button)

// 2
tapMeButton.setOnClickListener { v -> incrementScore() }
```

Taking each commented line in turn:

1. First, `findViewById` searches through your `activity_game` layout file to find the View that has the corresponding ID and provides a reference to it that you can store as a variable. Note you're using the `R` file constants in this case. It's crucial that you cast it to the View type you're expecting because `findViewById` only returns a View, not the subclass you will need.

2. `setOnClickListener` attaches a click, or *tap*, listener to the Button which calls `incrementScore`. You're instructing the button to listen for a click, then whenever it is clicked, increment your score.

You're nearly there now. Add a new variable to `GameActivity`, and initialize it to `0` as follows:

```
internal var score = 0
```

Next, replace the contents of `incrementScore` with the following:

```
private fun incrementScore() {
  score++

  val newScore = "Your Score: " + Integer.toString(score)
  gameScoreTextView.text = newScore
}
```

Here you increment your new score variable to the next number and then convert that number into a string to use with your score text view.

Finally, you use your `newScore` variable to set the text of `gameScoreTextView`.

Time to realize the fruits of your labor. Run your app and tap the button a few times. The score in the top left corner of the screen will increment with each tap.

You've just hit quite a milestone in your Android app development! You've created a view, given it an ID, accessed it in your code, and reacted to some user input. These are the fundamental tasks of app development, and you will repeat this cycle many thousands of times in your career. Take a moment to appreciate this major accomplishment. Well done!

Managing strings in your app

You've just gotten your first taste of writing code, you've got something up and running resembling a game, and you undoubtedly want to take your app further.

One of the most important elements of any app is the text, or strings, displayed onscreen. As you move ahead in your Android development career, you'll do well to master the ins and outs of using strings.

For instance, you're using English labels in your app, but that doesn't mean it's the only language your app can support. Supporting multiple languages in your app can often lead to broader markets for your app, and it's a feature you should seriously consider when putting your app in the app store.

In the previous section, you set the `gameScoreTextView` to use the string `"Your Score: " + Integer.toString(score)`. This works well if you're only targeting English-speaking users. But how would you support one, two, or a dozen other languages in the future, without writing spaghetti code?

The answer to this is **String resources**.

In your project navigator, expand the res/values folder and open **strings.xml**. You'll see a file with the following content:

```
<resources>
    <string name="app_name">Timefighter</string>
</resources>
```

strings.xml gives you a place to store all the strings in your app, in order to keep strings from being sprinkled throughout your code. This makes it really easy to add support to your app for another language. Rather than hunting through your entire project to change all the strings, you simply copy the file and change it to hold the language translations of your choice. Easy!

For now you'll use this file to keep your English text in a separate location. Add the following lines underneath the `app_name` string:

```
<resources>
  <string name="app_name">Timefighter</string>
  <string name="tap_me">Tap me!</string>
  <string name="your_score">Your Score: %s</string>
  <string name="time_left">Time left: %s</string>
  <string name="game_over_message">Times up! Your score was: %s</string>
</resources>
```

Now go back to `incrementScore` in **GameActivity.kt** and replace the contents of that method with the following:

```
private fun incrementScore() {
  score++

  val newScore = getString(R.string.your_score, Integer.toString(score))
  gameScoreTextView.text = newScore
}
```

`getString` is an Activity-provided method that lets you grab strings from the R file name or ID. In this case, you're retrieving the strings you added earlier to **strings.xml**.

You're also passing in a string for the placeholder %s you added way back at the beginning of this chapter.

> **Note**: To learn more about String Resources in Android, checkout the Android developer documentation at https://developer.android.com/guide/topics/resources/string-resource.html where you can also learn about string arrays and plurals.

Besides following the best practices for strings, your app is also ready for porting to another language, should you ever want to do that. Sprinkling strings throughout your app is one of the worst types of technical debt to incur.

With that out of the way, you can get back to developing your game.

Progressing the game

Currently, the game lets you increment the score infinitely. However, for a game named **TimeFighter** there isn't much fighting of time going on. In this section, you'll add a countdown timer that will limit the amount of time you have to increase your score.

At the top of the **GameActivity** class add the following new properties underneath your View properties:

```
internal var gameStarted = false

internal lateinit var countDownTimer: CountDownTimer
internal var initialCountDown: Long = 60000
internal var countDownInterval: Long = 1000
internal var timeLeft = 60
```

Here you declare a few new properties to give your game some life: a Boolean property to indicate when the game has started, a countdown object named countDownTimer for you to race against, a count down interval variable to indicate the rate at which the countdown will decrement and finally a variable to hold an integer representation of the countdown.

Finally, replace resetGame with the following method:

```
private fun resetGame() {
  // 1
  score = 0

  val initialScore = getString(R.string.your_score,
Integer.toString(score))
```

```
    gameScoreTextView.text = initialScore

    val initialTimeLeft = getString(R.string.time_left,
Integer.toString(60))
    timeLeftTextView.text = initialTimeLeft

    // 2
    countDownTimer = object : CountDownTimer(initialCountDown,
countDownInterval) {
      // 3
      override fun onTick(millisUntilFinished: Long) {
        timeLeft = millisUntilFinished.toInt() / 1000

        val timeLeftString = getString(R.string.time_left,
Integer.toString(timeLeft))
        timeLeftTextView.text = timeLeftString
      }

      override fun onFinish() {
        // To Be Implemented Later
      }
    }

    // 4
    gameStarted = false
}
```

Here, you initialize your game with a default state. You may have noticed when you first ran your game before there were some oddities like symbols appearing next to the time left TextView or the score TextView before you started the game. This method will ensure your game always has a default state to begin. Let's look at it closely.

1. You first set the score to 0, convert it into a string and use `getString` to insert the score value into your string stored in **strings.xml**. You then initialize the TextView with this value. You then repeat the process for the time left TextView.

2. Here you create a new **CountDownTimer** object and pass it into the `initialCountDown` and `countDownInterval` properties, which are set to 60000 and 1000 respectively. The CountDownTimer object will count from 60000 milliseconds, or 60 seconds, in 1000 milliseconds, or 1 second, increments, until it hits zero.

3. Inside the **CountDownTimer** you have two overridden methods: `onTick` and `onFinish`. `onTick` is called at every interval you passed into the timer; in this case, once a second. Each interval, the `timeLeft` property is updated with the time remaining by converting the millisecond representation into seconds. You then update `timeLeftTextView` with this new time. You'll call `onFinish` when `CountDownTimer` has finished counting down. You'll add some code to this later.

4. Finally, you inform your `gameStarted` property that the game has not started by setting it to `false`.

The next step is to hook up `resetGame` to run when you first create the Activity. How do you do that? That's right, back to your friend `onCreate`.

Add the following line to the bottom of the `onCreate` method:

```
resetGame()
```

Starting the game

Run your app, and things should look a little less jarring. The score TextView and time left TextView now show numbers instead of placeholders. Nice!

Click the button, and — no countdown! What is this madness?

Ah — you haven't told your countdown timer to begin counting down once the button has been clicked. Let's do that now. Replace `startGame` with the following:

```
private fun startGame() {
   countDownTimer.start()
   gameStarted = true
}
```

Here you inform the countdown timer to start. You also set `gameStarted` to `true` to inform any interested parties that the game has indeed started.

Finally, add the following lines to the top of `incrementScore`:

```
if (!gameStarted) {
   startGame()
}
```

This snippet of code checks to make the game has started when you tap the button. If not, then it starts the game for you.

Run the app to see what has changed.

Nice! Your countdown timer is now ticking merrily away.

Ending the game

Huzzah. T-Minus 60 seconds and counting to do — what exactly? The answer is "nothing", because the game doesn't know what to do after 60 seconds.

In the `GameActivity` class, replace replace the `endGame` method with the following code:

```
private fun endGame() {
  Toast.makeText(this, getString(R.string.game_over_message,
Integer.toString(score)), Toast.LENGTH_LONG).show()
  resetGame()
}
```

Here, you make use of a **Toast** to notify something to the user. A Toast is a small alert that pops up briefly to inform you of some event that's occurred — in this case, the end of the game.

Here, you pass into the Toast the Activity you want the Toast to appear on and the message to display. The end game state is a good time to display the score along with the game over message you put into **strings.xml**. You then inform the Toast to display for a long time with `Toast.LENGTH_LONG`, which in reality is a few seconds, and then show the Toast. Once that's done, simply reset the game.

You need to call `endGame` from somewhere. The best time to call this is when `countDownTimer` has finished counting.

Head over to `resetGame` and add the following line to `onFinish`:

```
endGame()
```

Run your app one more time, and keep clicking the button. The countdown will continue to decrement until it hits 0. Once it does, you'll see your Toast with your score and game over message, at which point the game will reset.
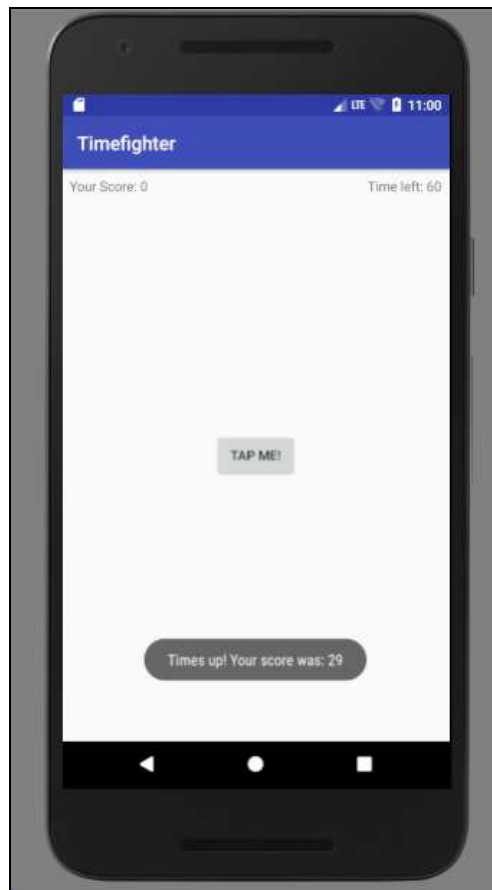
Where to go from here?

With a relatively small amount of code, you've created a functional game while learning some of the foundational elements of building an Android app. Although this Activity is rather small, Activities can quickly get very complex as you add more Views.

But no matter how large or small, creating any Activity has the same flow:

1. Create a Layout for the Activity.

2. Give your Views in your Layout some IDs.

3. Create properties in your Activity code and reference the IDs.

4. Manipulate your views as you see fit.

In the next chapter, you'll look at some potential problems in your app and learn how to fix them with some debugging techniques for Android.

Chapter 4: Debugging

By Darryl Bayliss

In the previous two chapters, you focused on developing **TimeFighter** into a fully-fledged app. In this chapter, you'll focus on how to debug your app when it begins to exhibit bugs.

All apps have bugs at some point in their lifetime. Some will be subtle, such as oddities within the UI, while others will be obvious, such as outright crashes. As a developer, you need to avoid bugs in your app at all costs to make sure your users are happy as can be.

Android and Android Studio provide developers with a number of tools to help you track down and fix bugs with relative ease. In this chapter, you will learn:

1. How to debug your app using Android Studio's debug tools.

2. How to add landscape support to **TimeFighter**.

Getting started

If you were following along with an app of your own, open the project in Android Studio and keep going with it for this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **TimeFighter** app under the **starter** folder. The first time you open the project, Android Studio takes a few minutes to set up your environment and update dependencies.

TimeFighter suffers from a problem you might not be aware of. Start the app in the emulator or on your device. Push the **TAP ME** button a few times, then change the orientation of your device to landscape.
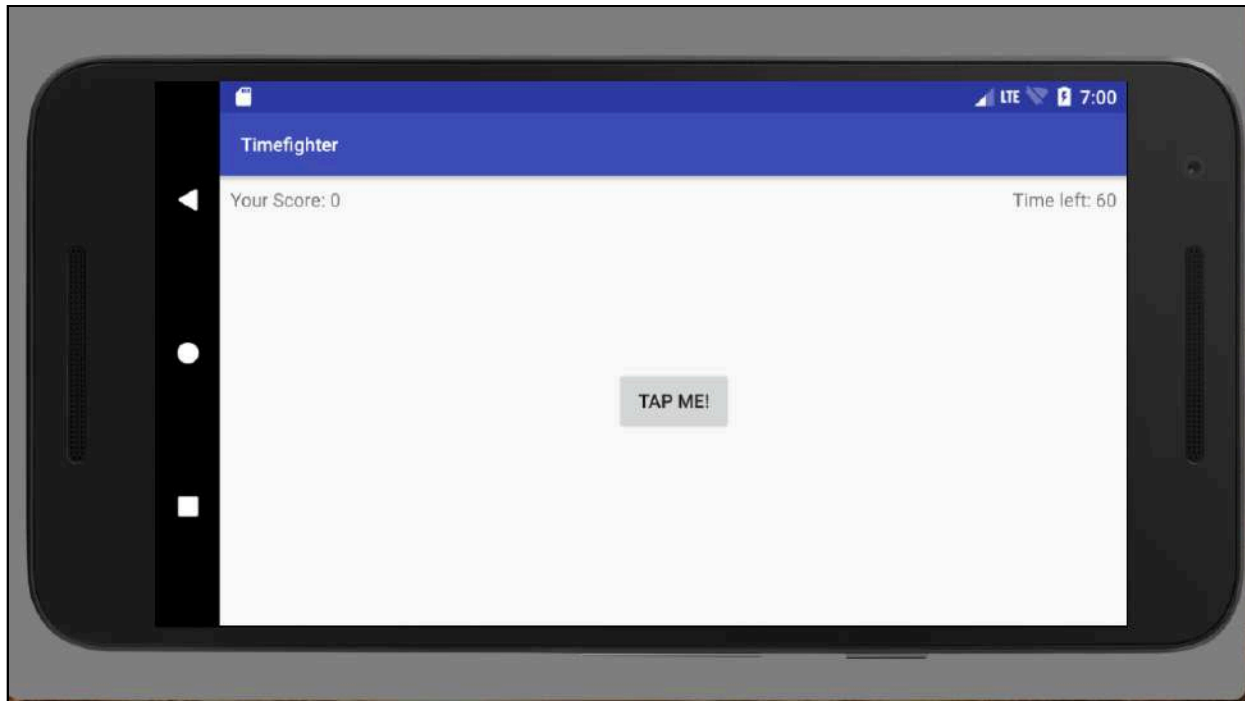
Notice the issue? **TimeFighter** resets mid-game and leaves you to start off again. That's way beyond annoying. To understand why this happens, you'll have to put on your debugging hat and analyze the code.

Add some logging

The first basic debugging approach that's available in most modern languages is to add logging to your app. Logging tells you what's what's happening at certain points in your code. You can even log the values of variables at runtime to check that they contain what you *think* they should contain.

In the **GameActivity.kt** class, add the following property to the top of the existing properties:

```
// 1
internal val TAG = GameActivity::class.java.simpleName
```

Then add the following line underneath the call to `setContentView` in `onCreate`:

```
// 2
Log.d(TAG, "onCreate called. Score is: $score")
```

What's going on here?

1. You assign the name of your class to the **TAG** variable. The convention is to use the class name in your log messages, which makes it easier to see what class the message is coming from.

2. You `Log` a message when the Activity is created. You app will inform you when `onCreate` is called and will also tell you what the current score is. Injecting `$score` into the message is an example of **string interpolation** in Kotlin. At runtime, Kotlin will look for the `score` variable and replace it in the log message.

Run your app again; once it's loaded, go to Android Studio and along the bottom of the window you will see a button labeled **Logcat**. Click it and the bottom of Android Studio will display a console-like window:
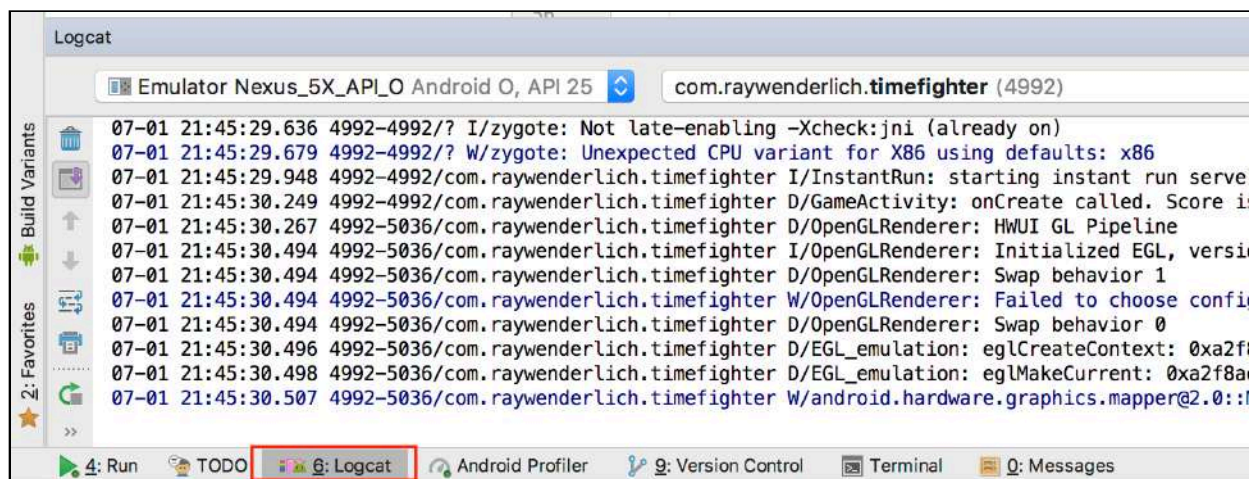
Logcat lets you see everything your emulator or device is doing through log messages, which includes messages coming from outside your app. For now, you can simply ignore most messages and filter down to just the ones you've added yourself.

In the Logcat window there is a search bar with a magnifying glass in it. Any text you enter here will filter the logs messages that match the text you are looking for. In the Logcat search bar (the one with a magnifying glass) type the name of your activity — **GameActivity** — and watch as the logs begin to clear up.

Huzzah! You can see the log messages you added earlier! The score is currently 0 because you haven't yet started the game. Try to reproduce the bug by rotating the screen as you play the game:
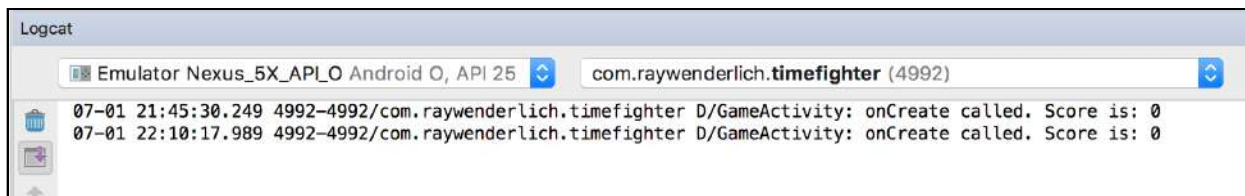
Huh? Why is the score reset to 0? This is due to the way Activities react to changes in orientation, which you will learn about in the next section.

> **Note**: You'll only scratch the surface of Logcat in this chapter. For a more complete explanation of what Logcat is capable of, head over to: https://developer.android.com/studio/command-line/logcat.html

Orientation changes

From your log messages, you've established that the `score` property is reset to 0 whenever you rotate the device. The reason for this relates to how Android handles device orientation changes. Android does three things whenever it detects a change in orientation:

1. It attempts to save any properties for the activity specified by the developer.

2. It destroys the activity.

3. It recreates the activity for the new orientation by calling `onCreate` and resets any properties specified by the developer.

Android performs these steps any time there's a change to the **configuration** of a device. A configuration change can be a result of many things, including changes to the orientation or the selected language. Your activity may be destroyed or recreated several times while your user is using the app.

This is why it's incredibly important that you develop your app so that it can recover cleanly from these changes.

In **GameActivity.kt**, add the following companion object at the bottom of properties you've declared:

```
// 1
companion object {

  private val SCORE_KEY = "SCORE_KEY"

  private val TIME_LEFT_KEY = "TIME_LEFT_KEY"
}
```

Next, add the following method underneath `onCreate`:

```
// 2
override fun onSaveInstanceState(outState: Bundle) {

  super.onSaveInstanceState(outState)

  outState.putInt(SCORE_KEY, score)
  outState.putInt(TIME_LEFT_KEY, timeLeft)
  countDownTimer.cancel()

  Log.d(TAG, "onSaveInstanceState: Saving Score: $score & Time Left:
$timeLeft")
}

// 3
override fun onDestroy() {
  super.onDestroy()

  Log.d(TAG, "onDestroy called.")
}
```

To recap the code above:

1. You create a companion object that contains two String constants, `SCORE_KEY` and `TIME_LEFT_KEY`, to track the variables you want to save when the orientation changes. You'll use these constants as keys into a dictionary of saved properties.

2. You override `onSaveInstanceState` and insert the values of `score` and `timeLeft` into the passed-in `Bundle` object. A Bundle is effectively a dictionary which Android uses to pass values across different screens. You also cancel the game timer and add a log to track when the method is called.

3. You override `onDestroy`, call the `super` implementation so your Activity can perform any essential cleanup, and added a final log to track when `onDestroy` is called.

Run your app again, play the game for a few seconds and change the orientation. Then look at your Logcat output:
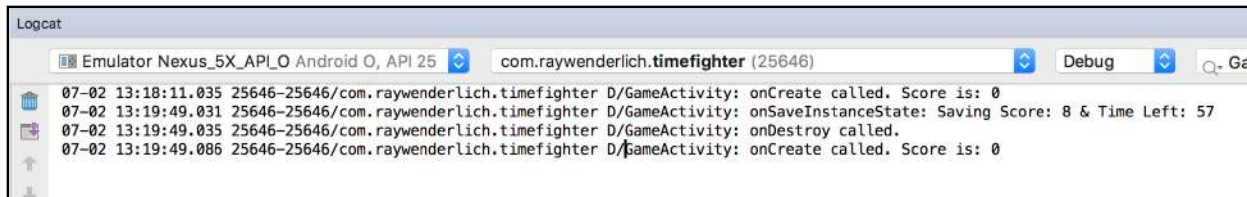
That's looking promising!

Breakpoints

Logging can be an effective way of understanding what your app is doing. However, it can be tedious to have to write a log message, recompile, rerun your app, and attempt to reproduce the bug. Android Studio provides **breakpoints**, which let you pause the execution of your app so you can inspect its current state.

In **GameActivity.kt**, scroll to `onSaveInstanceState` and find the log line at the bottom of the function. Then click on the grey border (also known as the *gutter*) to the left of the line:
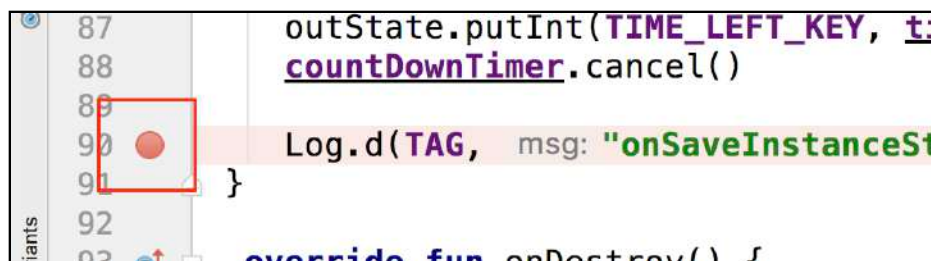
This add a red dot to the gutter to indicate where your breakpoint sits.

Next, click the **Debug** button at the top of the window. This looks like a bug with a small green play button in the bottom right:
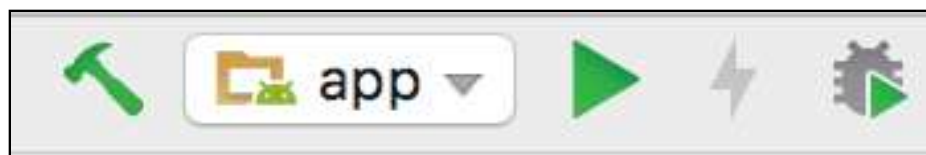
Once the app has reloaded, rotate the screen. Android Studio will shift windows and highlight the breakpoint:
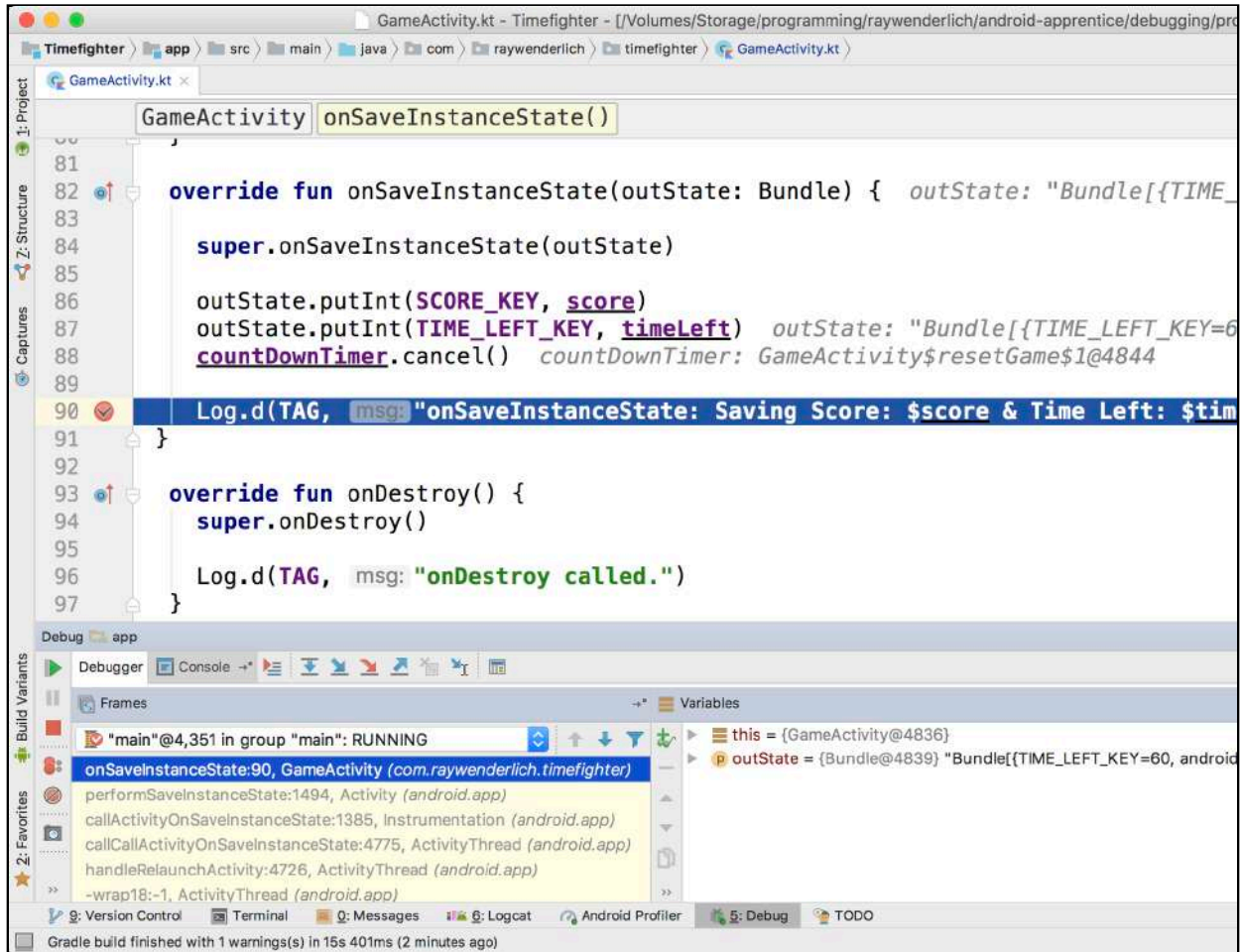
Your app is paused at the line that will be executed next. In this case, it's the log message you added earlier where you save your game variables to a bundle.

When Android Studio hits a breakpoint, it gives you the opportunity to inspect your app's state at that exact moment in time. You can see much of this information in the **Debug** window below your code. Move to the debugger view and click the arrow next to `this = {GameActivity}`.

The number postfixing your GameActivity will likely be different, since this number indicates where your activity is allocated in memory.

You'll recognize some of the values as your own, as well as some other values you'll be unfamiliar with. These are values specific to an Activity and give you an appreciation of how much work the Activity Class does behind the scenes.

Android Studio also inlines a lot of debug information within your code when it hits a breakpoint, making it even easier to inspect the state of you code.

Time to put this knowledge to use. Expand the `outState` Bundle object in the debugger and expand `mMap`. You should see some familiar values:

Compare those numbers with the values of your `score` and `timeLeft` variables, and they should match. This tells you those values are now safely stored in the `Bundle`. In the next section, you'll see how to get those numbers back when the device orientation has changed.

Restarting the game

Up to now you've only used `onCreate` to set up your Activity. You've never used the `savedInstanceState` object passed in as a parameter… until now!

Inside `onCreate`, replace the call to `resetGame` with the following:

```
if (savedInstanceState != null) {
   score = savedInstanceState.getInt(SCORE_KEY)
   timeLeft = savedInstanceState.getInt(TIME_LEFT_KEY)
   restoreGame()
} else {
   resetGame()
}
```

Here, you're checking to see if the `savedInstanceState` Bundle contains a value. If it does, then attempt to get the values of `score` and `timeLeft` from the Bundle you passed in earlier from `onSaveInstanceState`, assign those values to your properties, and restore the game. If the `savedInstanceState` Bundle doesn't contain a value, then simply reset the game.

Next, implement `restoreGame` underneath your `resetGame` method:

```
   private fun restoreGame() {

     val restoredScore = getString(R.string.your_score,
Integer.toString(score))
     gameScoreTextView.text = restoredScore

     val restoredTime = getString(R.string.time_left,
Integer.toString(timeLeft))
     timeLeftTextView.text = restoredTime

     countDownTimer = object : CountDownTimer((timeLeft * 1000).toLong(),
countDownInterval) {
        override fun onTick(millisUntilFinished: Long) {

          timeLeft = millisUntilFinished.toInt() / 1000

          val timeLeftString = getString(R.string.time_left,
Integer.toString(timeLeft))
          timeLeftTextView.text = timeLeftString
        }

        override fun onFinish() {
```

```
        endGame()
      }
    }

  countDownTimer.start()
  gameStarted = true
}
```

`restoreGame` will set up your `TextView` and `CountDownTimer`'s properties to the exact values inserted into the `Bundle` before the change in orientation.

Now run your app, play the game for a few seconds and rotate the device to see what happens:
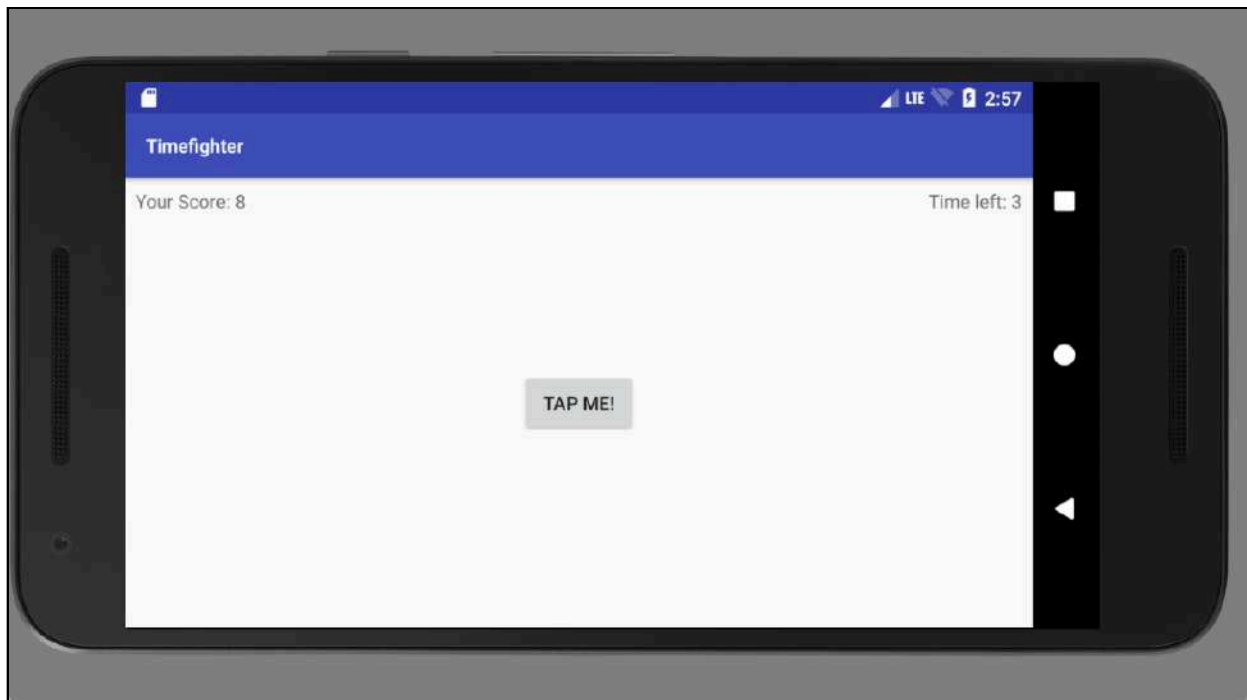
Woohoo! The score and time remaining stayed exactly the same. Bug fixed!

Where to go from here?

You've only scratched the surface of debugging in Android Studio. Finding and fixing bugs is an integral part of software development, so it's important that you become comfortable with the tools.

Android Studio contains a number of debugging tools that are beyond the scope of this chapter. To find out more, head over to https://developer.android.com/studio/debug/index.html.

Unfortunately, sometimes you aren't able to fix bugs due to factors beyond your control. There may be bugs in a third-party library you're using, or maybe even within Android itself! At this point, it helps to be a good developer and inform the developers who maintain that code via their bug reporting channels so they can investigate the bug in their own code.

For now, you're armed with enough tools and techniques to debug problems in your own app. In the next chapter, you'll finish up **TimeFighter** and add some polish so the app looks and feels more in place in the Android ecosystem.

Chapter 5: Prettifying the App

By Darryl Bayliss

Before getting into this chapter, take a moment to congratulate yourself and recognize what you've accomplished to this point. You've got a fully-working Android app, ready to entertain your users by fighting the clock to score as many points as possible.

You've also fixed a few undiscovered bugs and added the ability for the user to play the game in both portrait and landscape mode, regardless of the device they use. By all accounts, you have an app ready to go and entertain people for years to come!

…but it's not very *visually exciting*, is it?

Looking quite bland there!

Having an app that looks visually appealing and can surprise the user in gentle ways is one that will stick out when compared to similar apps. While it's not integral to the functionality of your app, it shows that you care about your app and gives it the "wow!" factor.

In this final chapter, you'll learn how to do the following:

1. How to adjust your app to adhere to the Material Design Guidelines.

2. How to add some small touches to give your app that polished feeling.

3. How to add a simple animation to your app to give it some life.

Getting started

If you were following along with your own app, open it and keep going with it for this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **TimeFighter** app under the **starter** folder. The first time you open the project, Android Studio takes a few minutes to set up your environment and update dependencies.

Open the **TimeFighter** project, run the app and think about what you could do to improve the way it looks and feels. Perhaps the color of the bar could be improved? Maybe that button feels a little lifeless when you tap it? Why is the screen white? It's so silent, could we have sound effects?

It probably hasn't taken you very long to come up with a lot of ideas for improvements. The important thing to remember is you don't need to do *everything*. You only need to make changes that add impact to the important elements of the screen — otherwise, you'll end up cluttering the screen and confusing the user.

Changing the app bar color

The bar at the top of the app looks like it could use a little more color.

In the project navigator, on the left side of Android Studio, open the **colors.xml** file in **app > res > values**. You'll see something like this:

```
<resources>
  <color name="colorPrimary">#3F51B5</color>
  <color name="colorPrimaryDark">#303F9F</color>
  <color name="colorAccent">#FF4081</color>
</resources>
```

This file is dedicated to storing the colors used in your app, it's similar to **strings.xml**. It's a good idea to keep your colors in a single place so you can swap them out quickly and avoid having to go through every layout to change colors.

You define colors with a `<color>` tag along with a `name` attribute that you will use as a reference in your app when it's compiled into **R.java**. The reference will be available for use in your XML layouts as well as during runtime in your code.

Within the `<color>` tag, you assign a hexadecimal representation of the color and close off the tag using `</color>`.

With the theory behind you, you can move to update your file. In **colors.xml**, change the values to the following:

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="colorPrimary">#0C572A</color>
  <color name="colorPrimaryDark">#388E3C</color>
  <color name="colorAccent">#8BC34A</color>
  <color name="colorBackground">#D3D3D3</color>
</resources>
```

You might be wondering how this changes the color of the bar at the top. The answer lies in the **styles.xml** file in **app > res > values**. Open up **styles.xml** and take a look at what you see:

```xml
<resources>
  <!-- Base application theme. -->
  <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
    <!-- Customize your theme here. -->
    <item name="colorPrimary">@color/colorPrimary</item>
    <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
    <item name="colorAccent">@color/colorAccent</item>
  </style>
</resources>
```

The important things to note here are the `<item>` tags. They are defining specific items within your app to adhere to a certain color. In this case, these colors are the colors you updated in **colors.xml**.

> **Note**: Curious about what's going on here? Your app is adhering to a **Style** set within this file. This is used to set the presentation of Views and screens and can be used to override items inherited from other themes provided by Android or other developers. For more information, visit: https://developer.android.com/guide/topics/ui/themes.html

One final tweak. Head over to the **activity_game.xml** file in **app > res > layout**, switch from **Design** to **Text** and update the **ConstraintLayout** tag to change the color of the background:

```
<android.support.constraint.ConstraintLayout
  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:layout_width="match_parent"
  android:layout_height="match_parent"
  android:background="@color/colorBackground"
 tools:context="com.raywenderlich.timefighter.GameActivity">
```

Here you reference the new `colorBackground` color added in **colors.xml**. With that done, give the app a run and see if you can see recognize it:
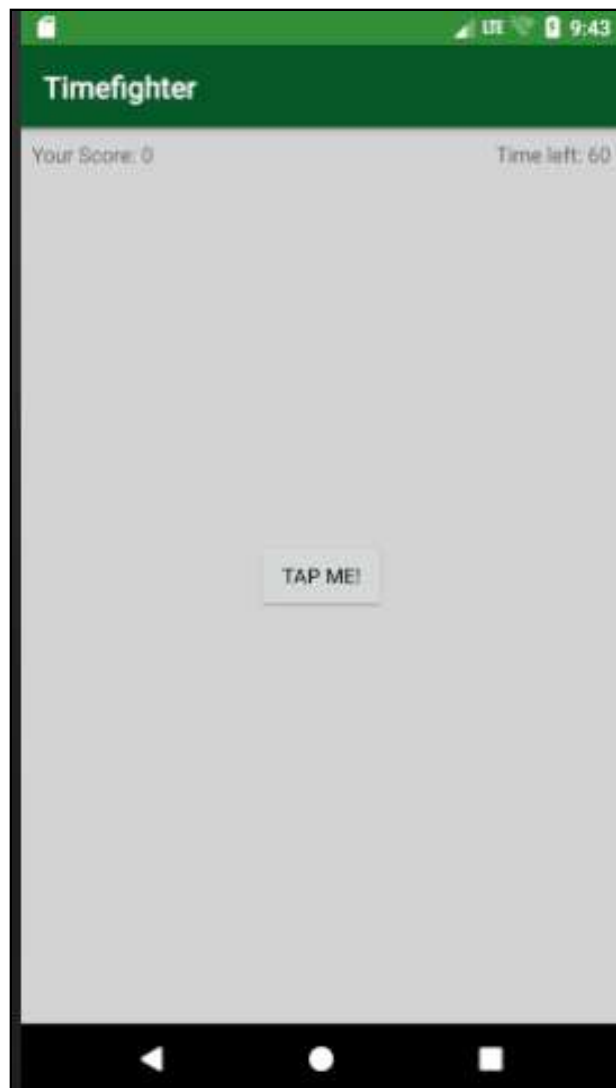
Looking quite interesting!

With a few lines of code, you've managed to transform your app and make it more visually appealing.

Animations

Animations give visual emphasis to elements and help you direct the user's attention. When it comes to animation, the most important rule is to use it where and when it matters — not simply because you *can*.

One of the most heavily-used components in your app is the **Hit Me** button, since that's what earns the user points in **TimeFighter**. Perhaps you could add an animation to it to make pressing it a little more exciting?

Right-click on the **res** folder. In the dropdown window, navigate to **New** and then click on **Android resource directory**.
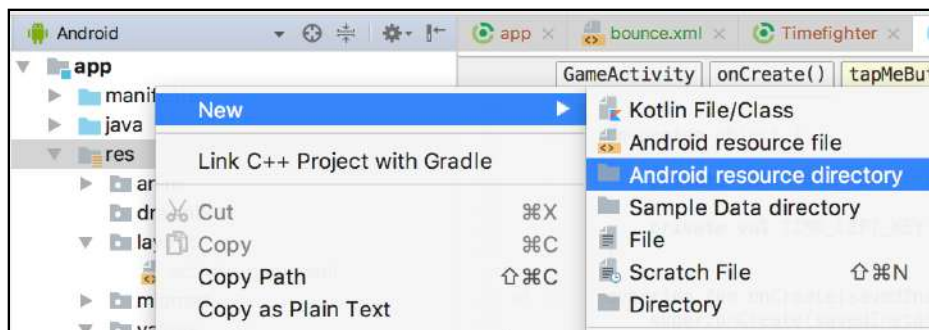
In the **New Resource Directory** window, change the Resource type to **anim** (this will automatically change the name of the directory as well) and click **OK**.
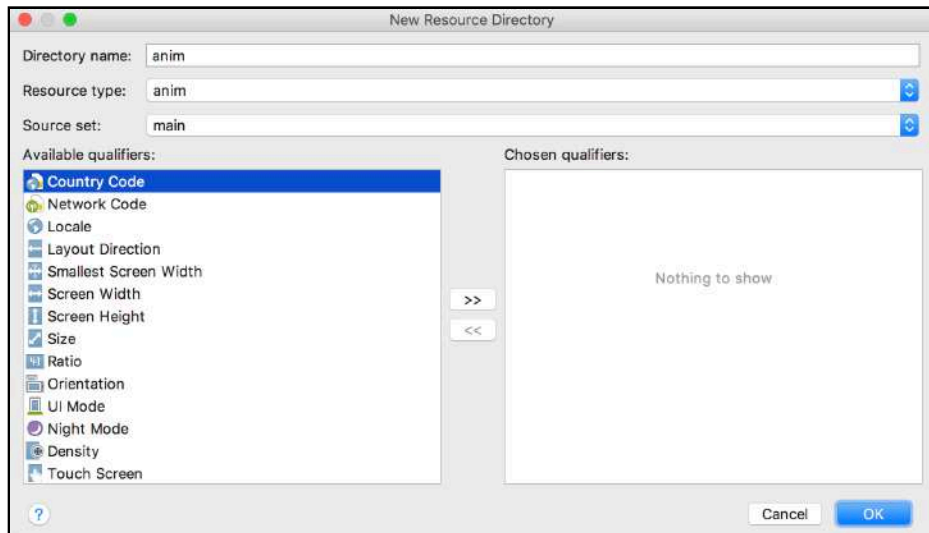
You will now have a new folder under the **res** folder in the project navigator named **anim**.

Next, you need to create the file which defines the animation. Right click on the **anim** folder, navigate to **New** and then click on **Animation resource file** on the rightmost dropdown.

You'll be presented with a dialog similar to the one you saw when you created the **anim** folder. This time though, you need to enter the name of the file. Give it a descriptive name that represents the animation it will contain.

In the File name text field, enter **bounce**, then click **OK**.
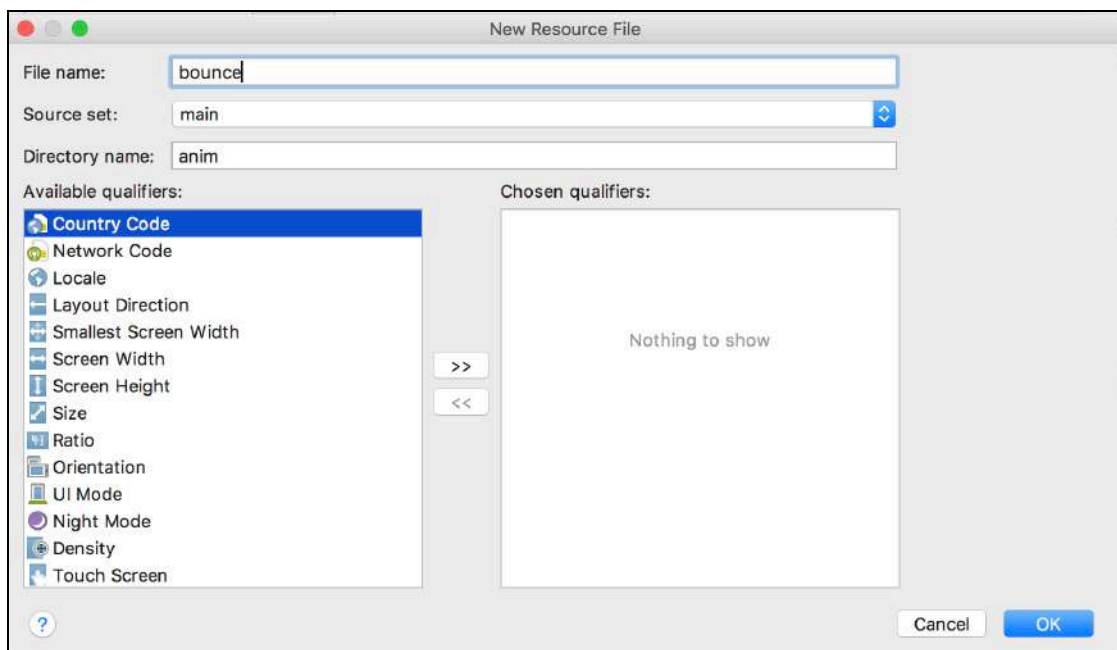
Android Studio will create the file and automatically open it for you. You'll see an XML file that looks like the following:

```xml
<?xml version="1.0" encoding="utf-8"?>
  <set xmlns:android="http://schemas.android.com/apk/res/android">

</set>
```

The most notable attribute here is the **set** attribute. This is a container which holds all the transformations that occur over the course of your animation, since you can bundle more than one transformation in the same animation and have them all run concurrently.

For now though, you'll only need to perform one transformation. Edit **bounce.xml** so it looks like the following:

```xml
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
  android:fillAfter="true"
  android:interpolator="@android:anim/bounce_interpolator">
  <scale
    android:duration="2000"
    android:fromXScale="2.0"
    android:fromYScale="2.0"
    android:pivotX="50%"
    android:pivotY="50%"
    android:toXScale="1.0"
    android:toYScale="1.0" />
</set>
```

Stepping through the XML to see what is going on:

```xml
<set xmlns:android="http://schemas.android.com/apk/res/android"
  android:fillAfter="true"
  android:interpolator="@android:anim/bounce_interpolator">
```

Here, the set is declared and instructed to `fill` after the animation is complete. This means the animation will not reset the View that is animated back to its original position before the animation took place. Instead, it will remain wherever it is when the animation ends.

The set is also being told to use the `bounce_interpolator` from Android. This affects the rate that an animation is performed over time independent of any duration you set within your animation.

Android provides a number of built-in interpolators, and lets you create your own if you don't find one that suits your needs. For now though, the `bounce_interpolator` that comes with Android will work nicely.

Let's look at the next few lines:

```xml
<scale
  android:duration="2000"
  android:fromXScale="2.0"
  android:fromYScale="2.0"
  android:pivotX="50%"
  android:pivotY="50%"
  android:toXScale="1.0"
  android:toYScale="1.0" />
```

Here you've declared a `scale` attribute. This informs the animation that resizing of the View should occur, you've also declared the duration to be 2000 milliseconds, or 2 seconds.

You also specify the scale that the animation should start from in `fromXScale` and `fromYScale`, and that the height and width of the View should be twice the original size at the beginning of the animation.

`pivotX` and `pivotY` specify the center point from which the animation should occur. In this case it occurs from the center of the View. Finally, you specify to which scale the View should shrink via the `toXScale` and `toYScale` attributes.

Here you want the View to shrink back to its original size, so the desired scale is 1.0.

So to review, the animation you've defined will:

- Scale the animated View to twice its size.

- Shrink it back to its original size.

- Do this over the space of two seconds.

- Using a bouncing interpolator.

> **Note**: If you want to know more about animation resources and interpolators on Android, then head over to https://developer.android.com/guide/topics/resources/animation-resource.html for an in-depth review.

That's it for your bounce animation. Open up **GameActivity.kt**, and modify the `tapMeButton.setOnClickListener` callback in `onCreate` to look like the following:

```
tapMeButton.setOnClickListener { v ->
  val bounceAnimation = AnimationUtils.loadAnimation(this,
      R.anim.bounce);
  v.startAnimation(bounceAnimation)
  incrementScore()
}
```

Now, every time you click the `tapMeButton`, Android will load the bounce animation defined within the **anim** folder, then tell your button to begin running that animation. Run the app, and once it has updated, click the button.

That's bound to make clicking a button a lot more interesting.

Adding a Dialog

The final thing you need to do is let everyone know who exactly built the app. Come on, you need to give yourself some credit, don't you? However, you don't want people to get

distracted from playing your game. If only there was a way to do that...

Fortunately there is — in the form of **Dialogs**. Think of a Dialog as a way to provide a snippet of information your users need, without being taken away from the main content on your screen. They come in all shapes and sizes, but in this case you just want to let your users know about the creator of the app and what version of **TimeFighter** they're running.

An easy way to do that is to set up a button in the top bar. To do that, you need to define a menu.

Head over to your **res** folder and right click on it. Just like you did to create the **anim** folder, right-click on the folder and select **Android resource directory**. In the pop-up window, change the resource type to **menu** and click **OK**.
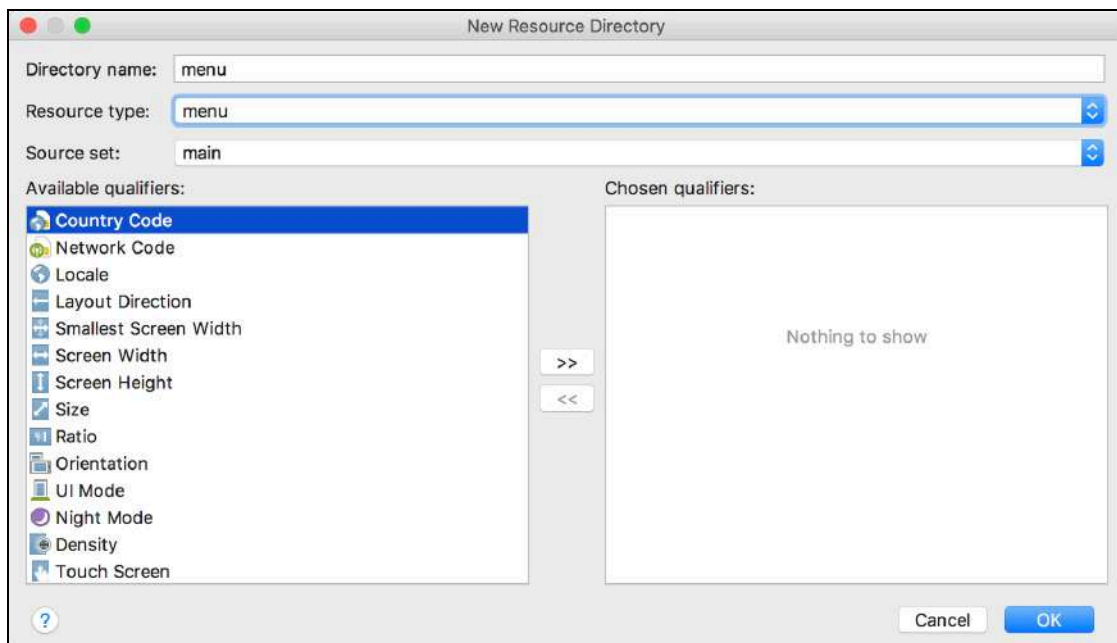
Right click on your newly created **menu** resource folder. In the pop-up menu, hover over **New**, then click on **Menu resource file**.
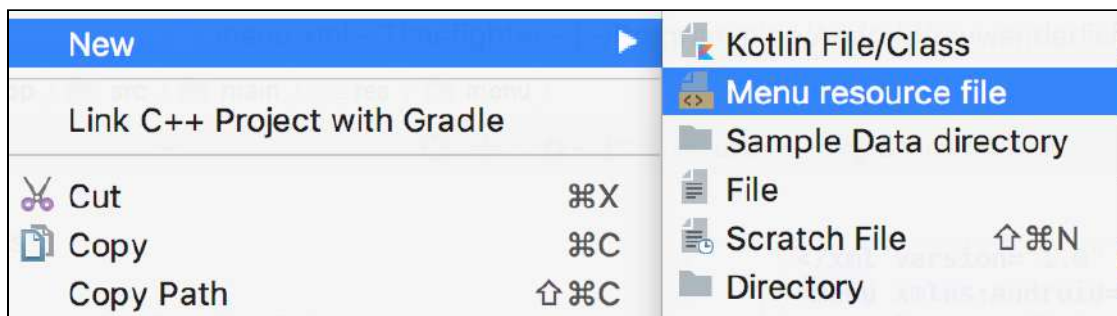
In the **New Resource File** window, enter the file name as **menu** and click **OK**:
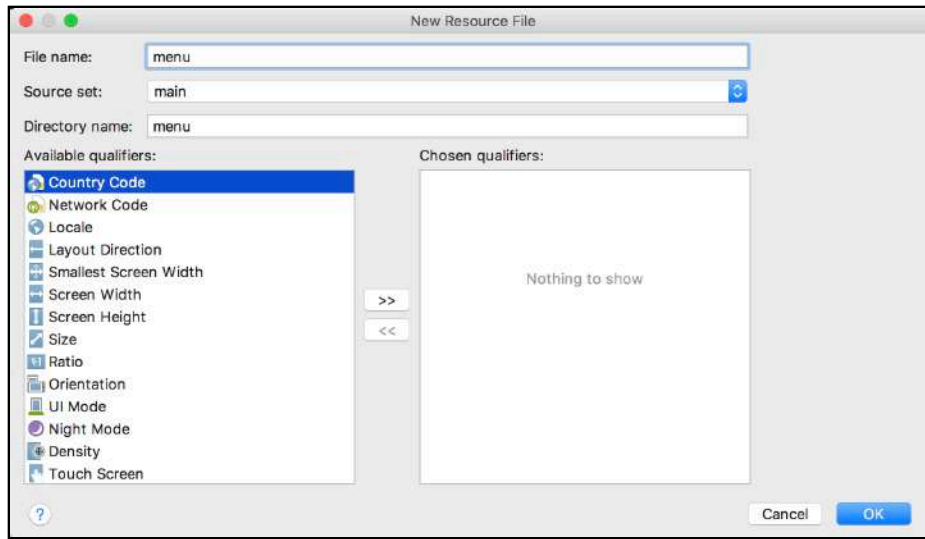
Android Studio will change over to the layout window and show you a similar setup to what you've seen when editing layout files:
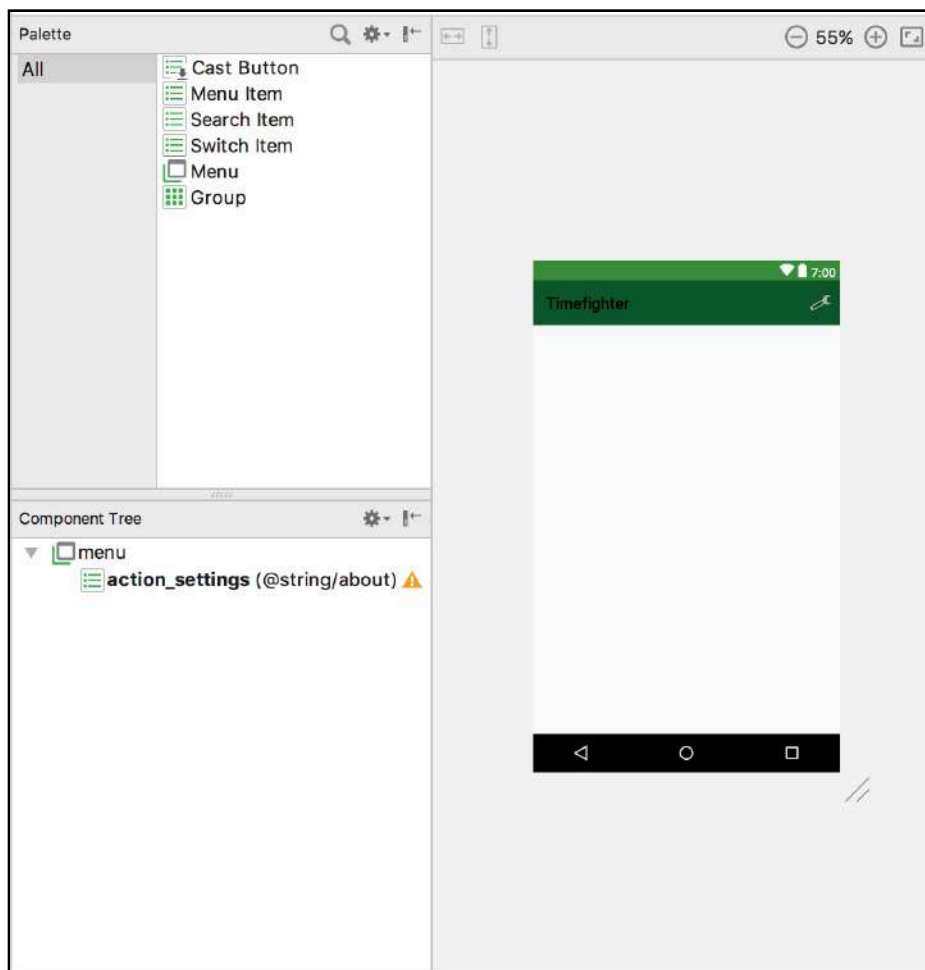
Here you have all the usual windows to help create a menu for your app. The **Palette** in the top left has changed to show only menu-specific items, and the **Component Tree** gives you an overview of the hierarchy for your menu.

You only want one item in your menu. To do that, move your cursor over to the **Menu Item** button in the **Palette** window, and click and drag from the **Menu Item** and onto your layout.

You should end up with something like this:
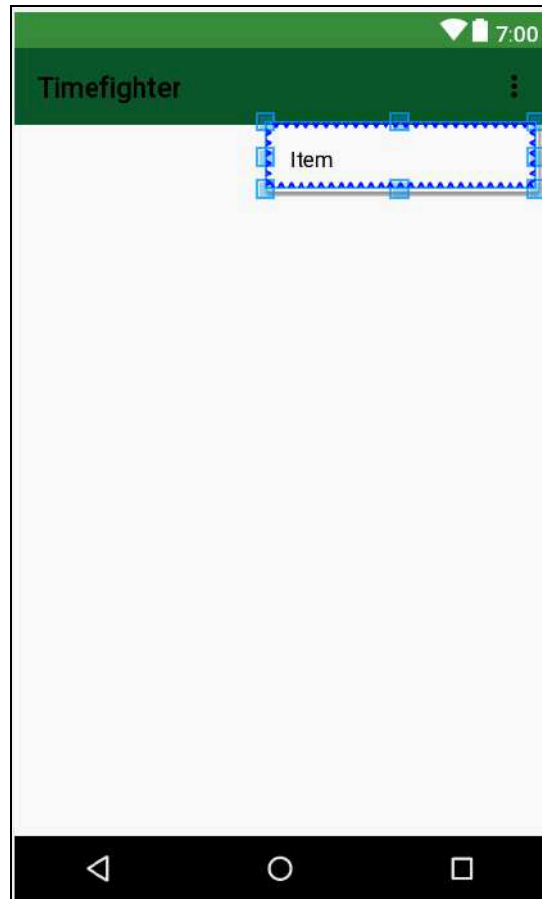
So far, so good. Now you'll tidy it up a little. Your newly-placed menu item should now be highlighted, and the **Attributes** window will be shown on the right. Let's add/edit some of these attributes.

First, set **id** for your menu item and call it **action_settings**. Next, move onto the **title** attribute and name your menu item **About**.

Now you need to decide what icon you want to use for your menu item. Android comes with plenty of embedded images to choose from, so you can use one of those.

Click the small dots next to the **icon** textfield. They appear like so:
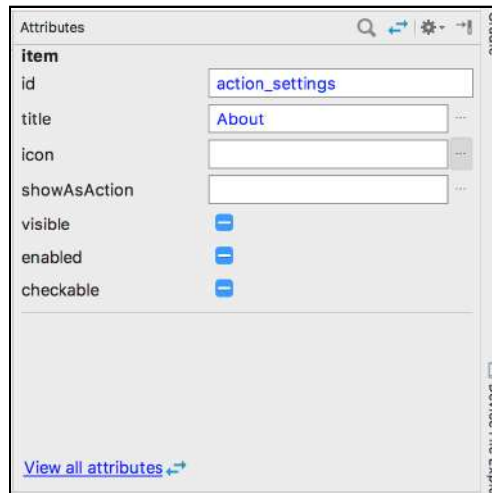
Once you've clicked the small dots, you'll be presented with the resources window.

This window shows you all resources that are available to use within your app, whether they come from Android or your own custom resources. The window shows both images (or **drawables**, as Android refers to them) or colors.
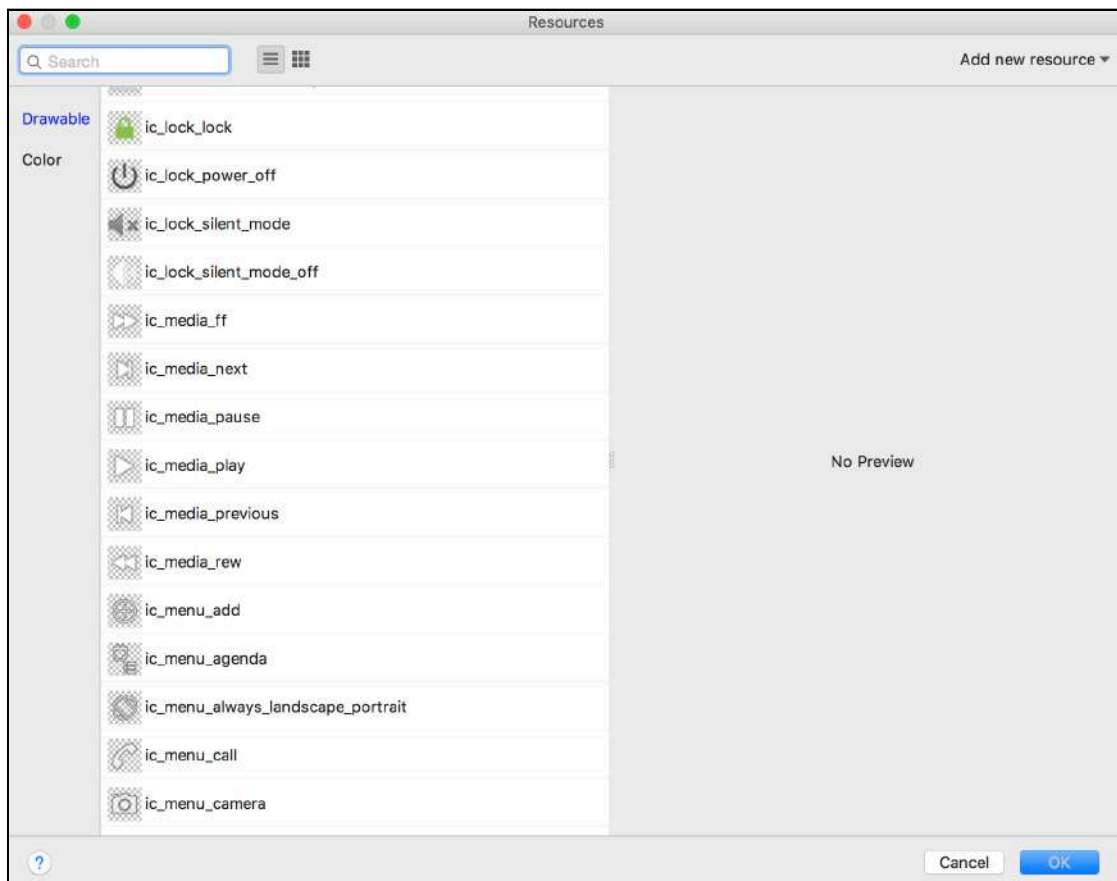
In the top left of the resources window is a search bar. Click in the search bar and type **ic_menu_preferences**. As you type, the list of resources will filter down to match any resources that contain the characters you type. In this case, there is only one.
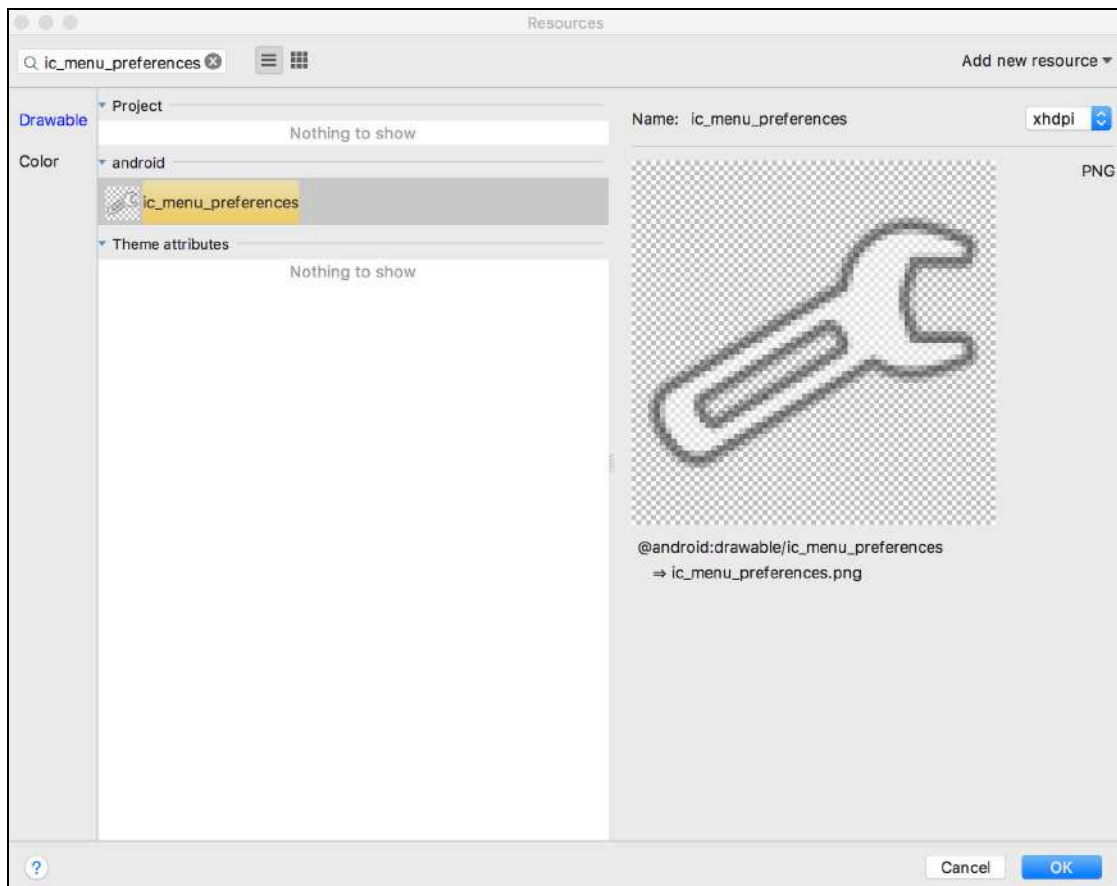
Click the resource under the Android dropdown to select it, then click **OK**. The resource window will close and take you back to the layout window. Now, the icon textfield will be populated with the resource you chose.

Finally, to make sure your button is always shown, you need to set the **showAsAction** attribute. This affects how your menu item is presented, and can range from a number of choices depending on the number of items your menu contains or the screen size of your device.

You want your menu item to always show up regardless of the circumstances. To do that, you need to click the small dots next to the **showAsAction** attribute. In the dialog that appears check **Always**, then click **OK**.
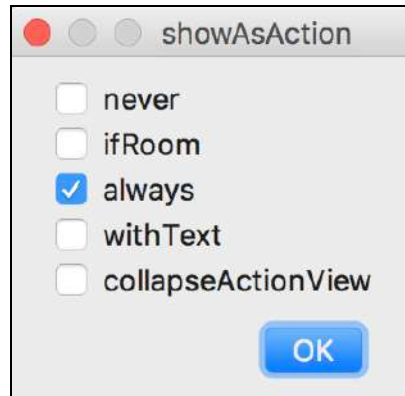
Looking good! Now for some Kotlin code. In **GameActivity.kt**, you need to override a few methods to ensure your menu appears and interacts as expected.

Add the following method underneath `onDestroy()`:

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
  // Inflate the menu; this adds items to the action bar if it is
present.
  super.onCreateOptionsMenu(menu)
  menuInflater.inflate(R.menu.menu, menu)
  return true
}
```

Here you're hooking into the Activity callback for when it attempts to create the menu. You first make a call to the `super` implementation to make sure any superclasses have a chance to set themselves up, and then you use the Activity's `menuInflater` to programmatically setup your menu layout for the Activity. Finally, you return `true` to let the Activity know that the menu is set up.

Next, underneath `onCreateOptionsMenu(menu: Menu)` add this method:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
  if (item.itemId == R.id.action_settings) {
    showInfo()
  }

  return true
}
```

This method is called whenever an item in your menu is selected. If the ID of the menu item is equal to the ID of the item you setup earlier, then run the `showInfo()` method. Once again, you return `true` to let the Activity know the the event has been processed.

Nearly there! Just a few more lines. You need to define the `showInfo()` method you just referenced. To do that add the following method to your **GameActivity.kt** class. It's fine anywhere inside the class, even as the last method in the class. Also, don't worry about editor errors just yet since we'll be adding some more strings next.

```kotlin
private fun showInfo() {
  val dialogTitle = getString(R.string.about_title,
      BuildConfig.VERSION_NAME)
  val dialogMessage = getString(R.string.about_message)

  val builder = AlertDialog.Builder(this)
  builder.setTitle(dialogTitle)
  builder.setMessage(dialogMessage)
  builder.create().show()
}
```

What's going on in this method? `showInfo()` handles the setting up of a dialog View for yourself. It first creates two strings to use in the dialog, one for the title, and one for the message. These strings are created by a mixture of the strings stored in your **strings.xml** file and any strings generated when your app is built. In this case, this is the **VERSION_NAME** of your app. The version name is already available, and you'll set up the other strings you need in **strings.xml** in just a moment.

Next, you create an **AlertDialog.Builder** object and pass in a Context instance to let the Dialog know where it should appear. You then pass in your title and message, create the Dialog, and finally display it.

> Note: When you add the line `val builder = AlertDialog.Builder(this)`, Android Studio will offer to auto-import a library for you and it offers up several options. Be sure to select the Android Support Library `android.support.v7.app`.

It's time to add in those missing strings. Open up your **strings.xml** and add the following strings (substituting your name in the `about_message` string):

```xml
<string name="about_title">Timefighter %s</string>
<string name="about_message">Created by YOUR NAME HERE</string>
```

Finally, run your app and check out the new wrench icon sitting in the top right of the screen:
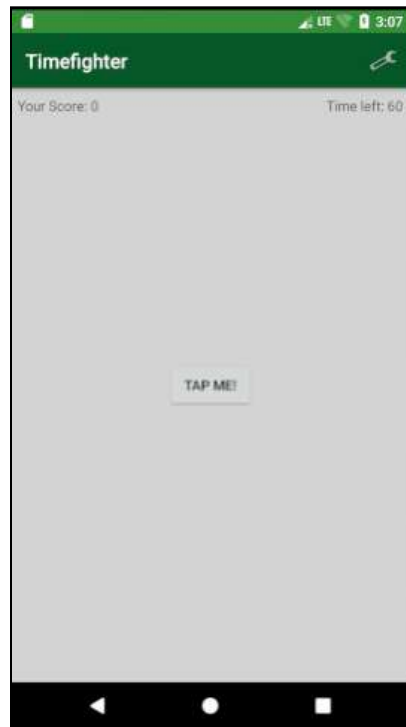
Tap the wrench and you'll be presented with a dialog that doesn't completely obscure the screen:

There you have it. A small place where people know what version of your app they're using, and who created it!

Where to go from here?

Congratulations on completing this chapter and the first section of the book! You've learned a lot over the last few chapters, and now know how to create a simple game app. In the next section, you're going to leave **TimeFighter** and move onto a different app to build upon the skills and concepts you've picked up in this first section.

Take a moment to appreciate what you've achieved, then carry on to the next section!

Section II: Building a Checklist App

Welcome to Section II of the book! You're going to leave behind the last app you made and create a completely new app. This new app is called ListMaker, and will allow you and your users to create handy lists that you can look at later.

In the previous section, you had a starter project to begin building your app. But in this section, you're going to create your own project from scratch! You'll go through the steps and choices given to you to ensure your project is set up right from the very start.

Chapter 6: Creating a New Project

Chapter 7: RecyclerViews

Chapter 8: SharedPreferences

Chapter 9: Communicating Between Activities

Chapter 10: Completing the Detail View

Chapter 11: Using Fragments

Chapter 12: Material Design

Chapter 6: Creating a New Project

By Darryl Bayliss

You're going to leave behind the last app you made and create a completely new app. This new app is called **ListMaker**, and will allow you and your users to create handy lists that you can look at later.

In this chapter, you'll learn to do the following:

1. Give your project an appropriate name and initial package structure.

2. Learn about each step of the project setup process and what each screen is for.

3. Set up your new project so you're ready to begin creating your next app.

Getting started

Open Android Studio and you'll be presented with a screen like this:
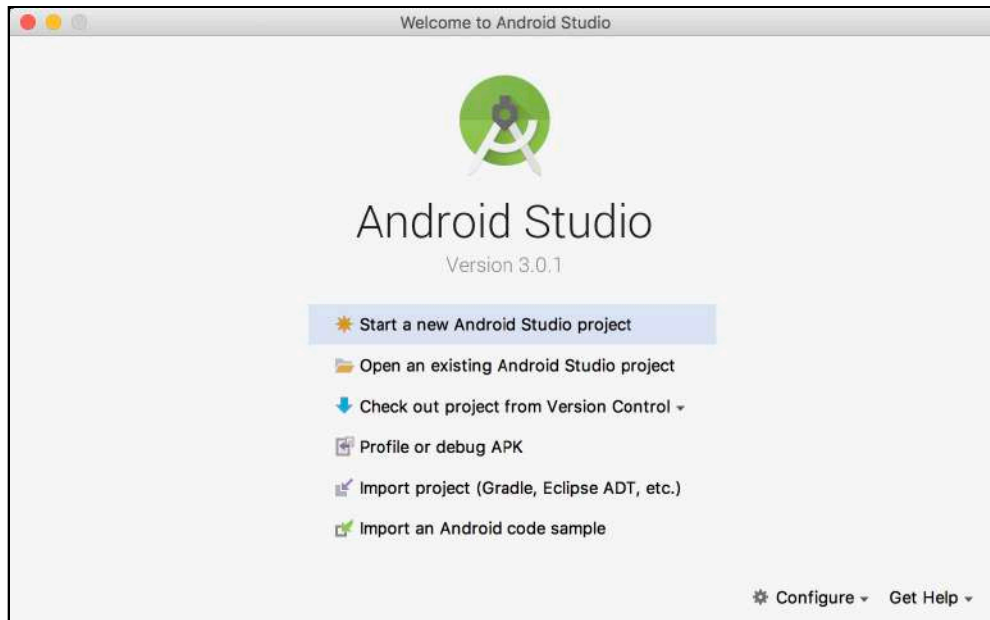
Before you begin to go through the steps of creating a new app, there are some useful features in Android Studio that are worth pointing out for each of the options in this screen.

Start a new Android Studio project: Starts the process of creating a brand new project for you to build your app. You'll come back to this one.

Open an existing Android Studio project: Lets you navigate through your machine's file structure to find and open an Android Studio project you've already gotten from somewhere else.

Check out project from Version Control: Open an Android Studio repository from the Internet on your machine. Because Android Studio is built on IntelliJ, an IDE from the company JetBrains, you get access to powerful version control tools right inside Android Studio.

Clicking on the **Check out project Version Control** button presents you with a number of version control systems Android Studio can work with, such as Git or Mercurial. Android Studio also works with version control storage providers such as GitHub or Google Cloud.
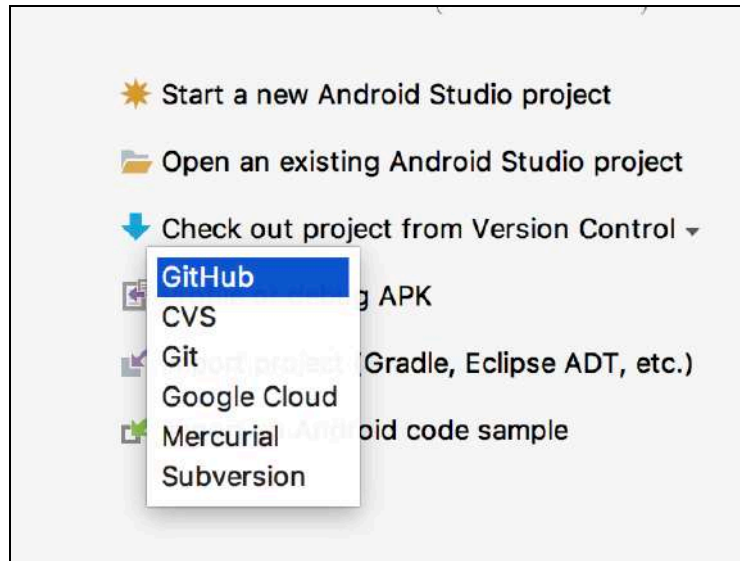
If you don't already use another version control system, we highly recommend checking out the tools within Android Studio for your versioning control needs.

Profile or debug APK: Gives you the option to select an Android app (`.apk`) file from your machine's file system and run it on a device or emulator to gather useful information about the app to learn or make improvements.

The information you can gather with this option ranges from the size of the app and its contents, to more sophisticated information gathered during runtime such as memory usage or network activity.
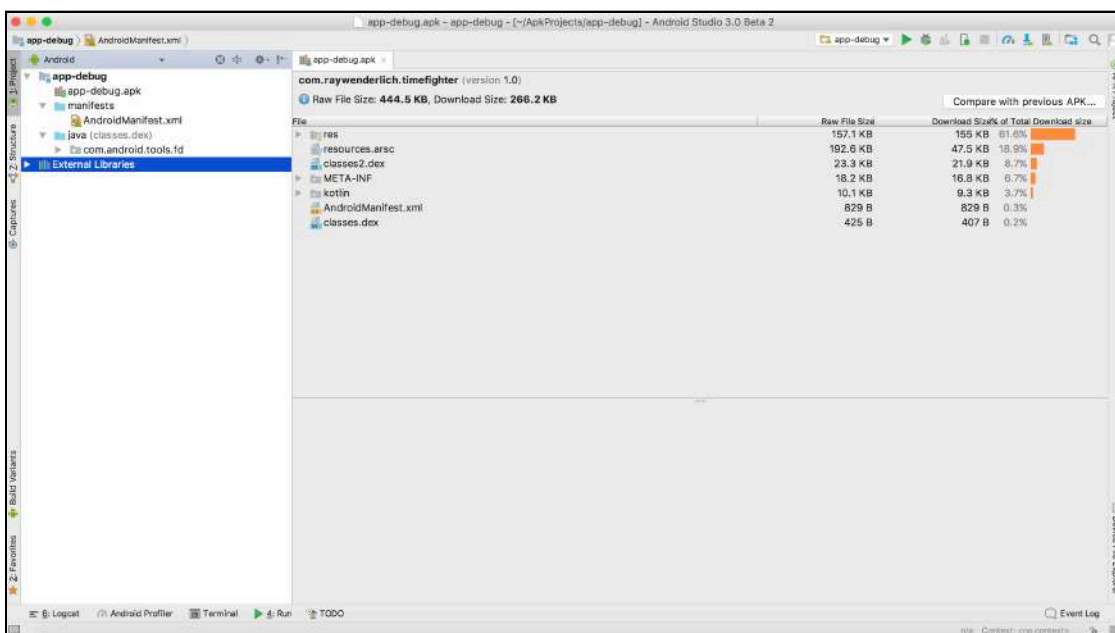
Import project (Gradle, Eclipse ADT, etc): Provides a more sophisticated way to open an Android project. This option allows you to convert older codebases to the newer Gradle build system Android Studio relies on. To keep it simple, if you have a complex Android Studio project, or an ancient project to maintain, this is the place to go.
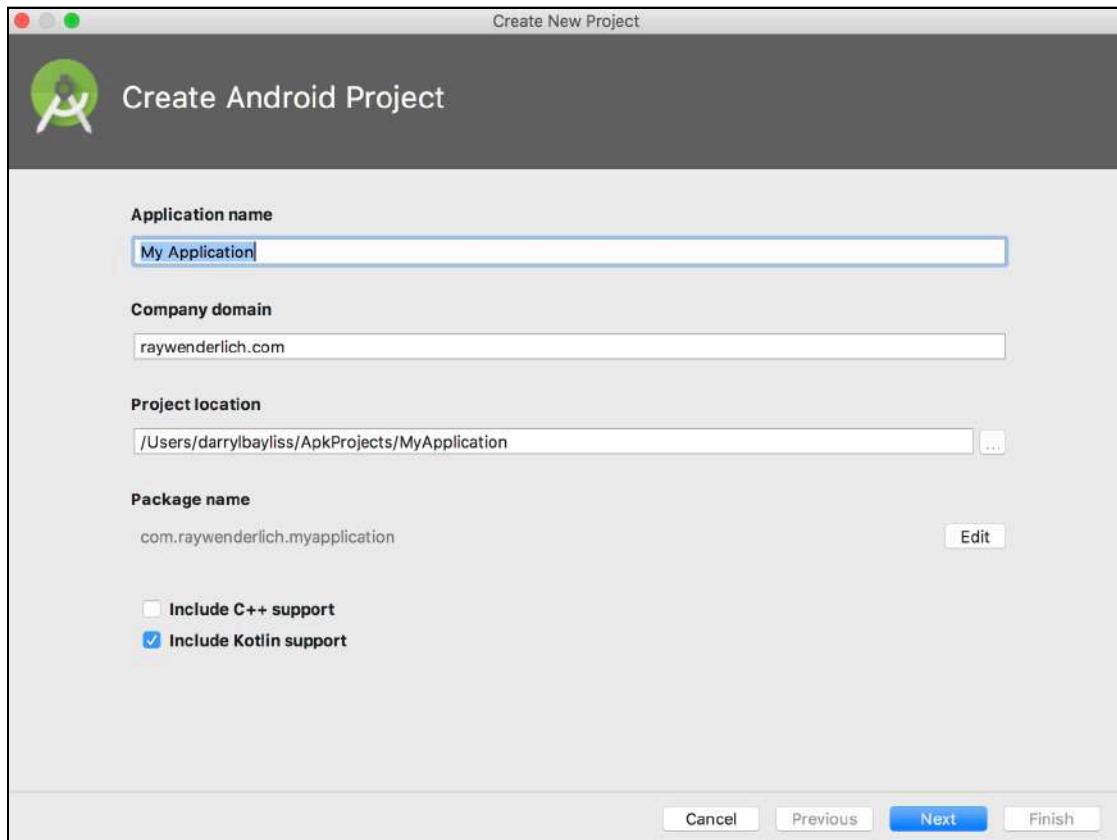
Import an Android code sample: Lets you import a treasure trove of sample projects provided by Google to demonstrate the features of Android. Here you can find Android Studio projects covering all sorts of topics from using emojis in your app, to more technical topics such keeping your users' data secure.
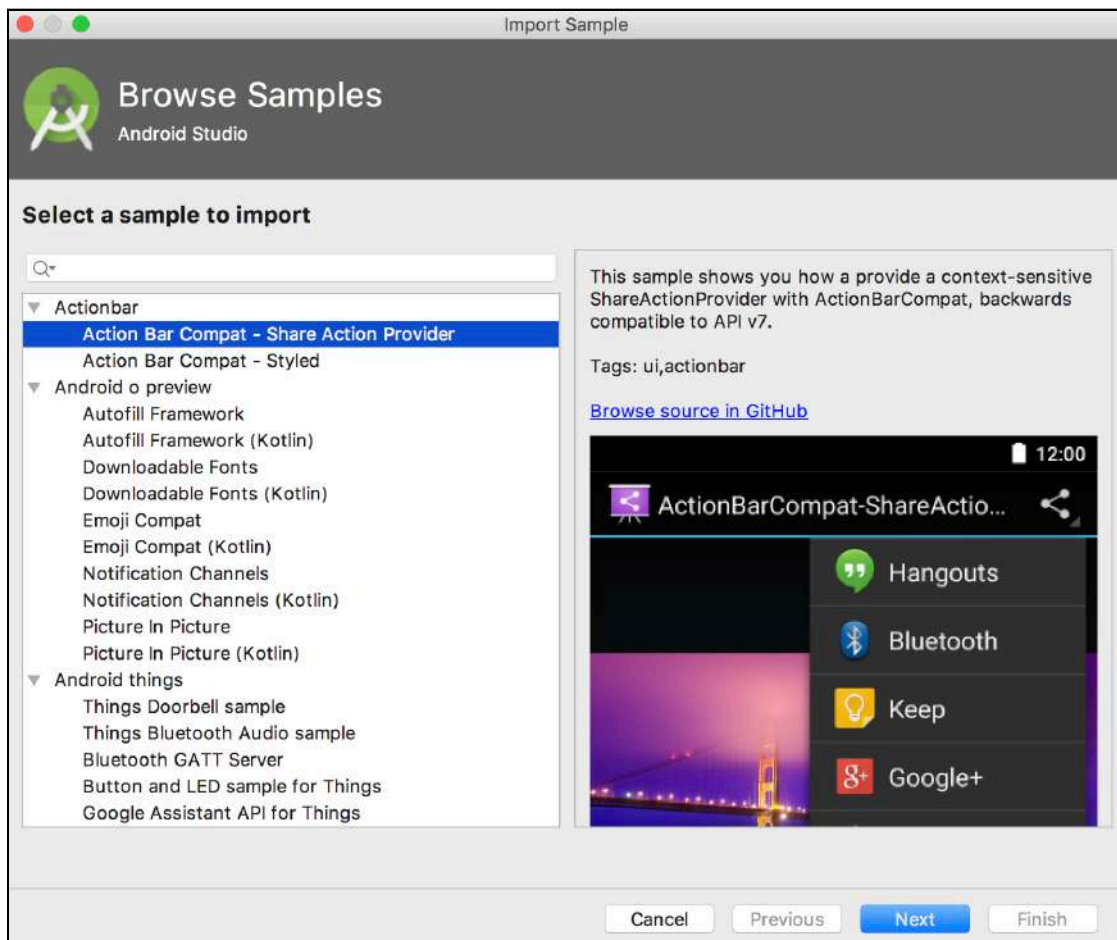
You'll keep it simple this time around. Click **Start a new Android Studio project** to begin creating your own app from scratch!

Creating a new Android project

After clicking the Start Project button, you'll be presented with a new window asking for a few pieces of information:

The first text field at the top is where you enter the name of your app. Type **ListMaker** in this field.

The second textfield is used to identify the packages used within your app. Packages describe how your code is structured, and it's good practice to name them in a way that describes what is inside each package.

This textfield allows you to set the company domain that is prefixed before a package name. This ensures that the package you create is unique to you and will minimize any chances of other external packages from conflicting with your own.

Packages with the same names will cause Android Studio to be confused about where to look for specific code. In this textfield type **raywenderlich.com**. That seems pretty unique!

The final textfield is where your project will be created on your machine. Feel free to change this to wherever you want to save your project. To the right of the textfield is a button with three dots. Clicking this will open up your machine's file system and let you set your save destination.
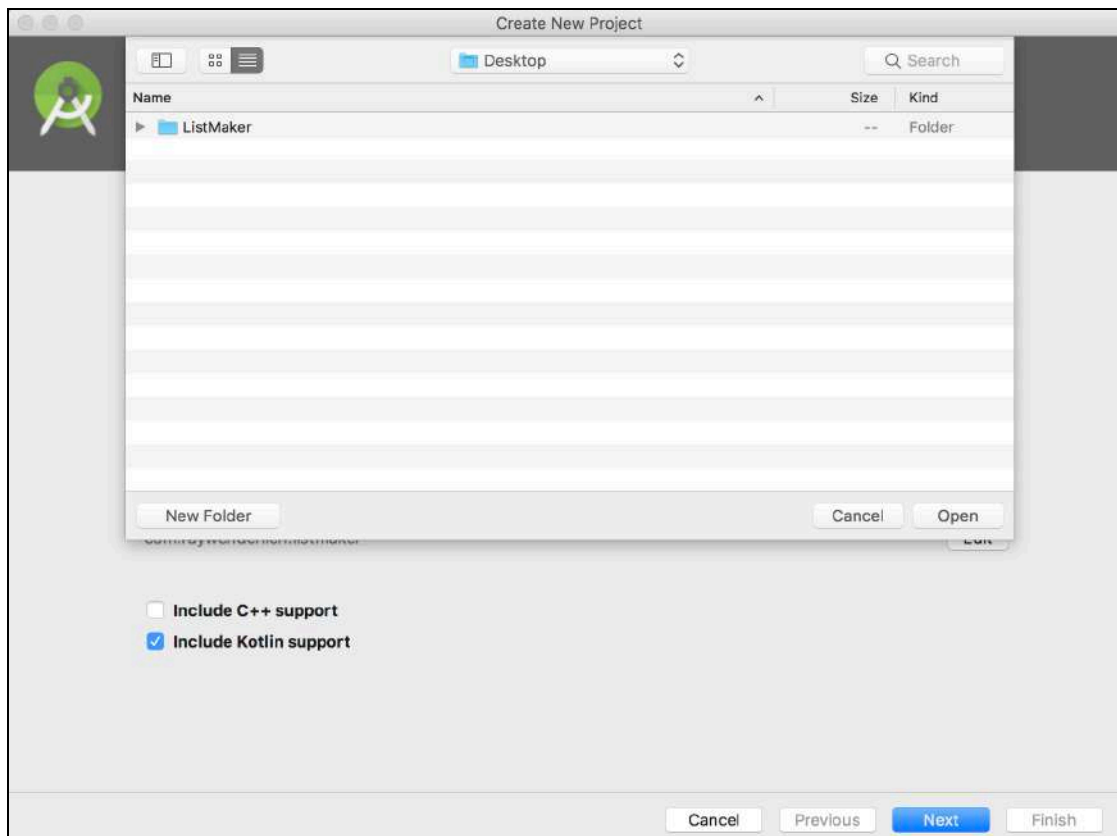
Before you move on, take a look at what else is on this screen. The first thing is **Package Name**. This shows you exactly what your default package name will look like within your app.

By default, this is a combination of your app's name and company domain you entered earlier. If you prefer it to be something else, you can click the **Edit** button to the right, turning the grayed-out package name text into a textfield.

Package name

com.raywenderlich.listmaker | Done |

Unless you have a good reason to change this, it's recommended to leave it as-is. Let's look what else is available.

Below the package name text are two checkboxes. The first one is **Include C++ support**. This gives your Android app the ability to use C++. We won't cover the use of C++ within this book. However, if you want to use C++ in your app, or leverage some C++ code written for other platforms, it's worth checking this box to give you access to some C++ specific tools in your new project.

The final checkbox is labelled **Include Kotlin support**. Make sure this box is checked so you can use Android Studio's included Kotlin language support within your apps.

Previously, you could only use Java to create Android apps, since that's the language used to build the Android libraries. However, Google has now officially adopted Kotlin as an additional language that can be used to create Android apps. Thanks to the clever thinking at JetBrains, the company that created the Kotlin language, you can use Kotlin in your apps alongside Java.

This is a setting you definitely want to keep checked, especially since we use Kotlin in this book!

Now that everything is covered, click **Next** to go to the next step.

Targeting Android devices

The next screen is the **Target Android Devices** screen. This gives you the opportunity to tell Android Studio what kind of Android devices you want to support in your project.

Yes, you heard that right. Android extends beyond just phones and tablets! It also runs on various wearables such as watches and fitness trackers, television sets, automotive systems within your car, and even various electronics grouped under that wide umbrella you know as the "internet of things".
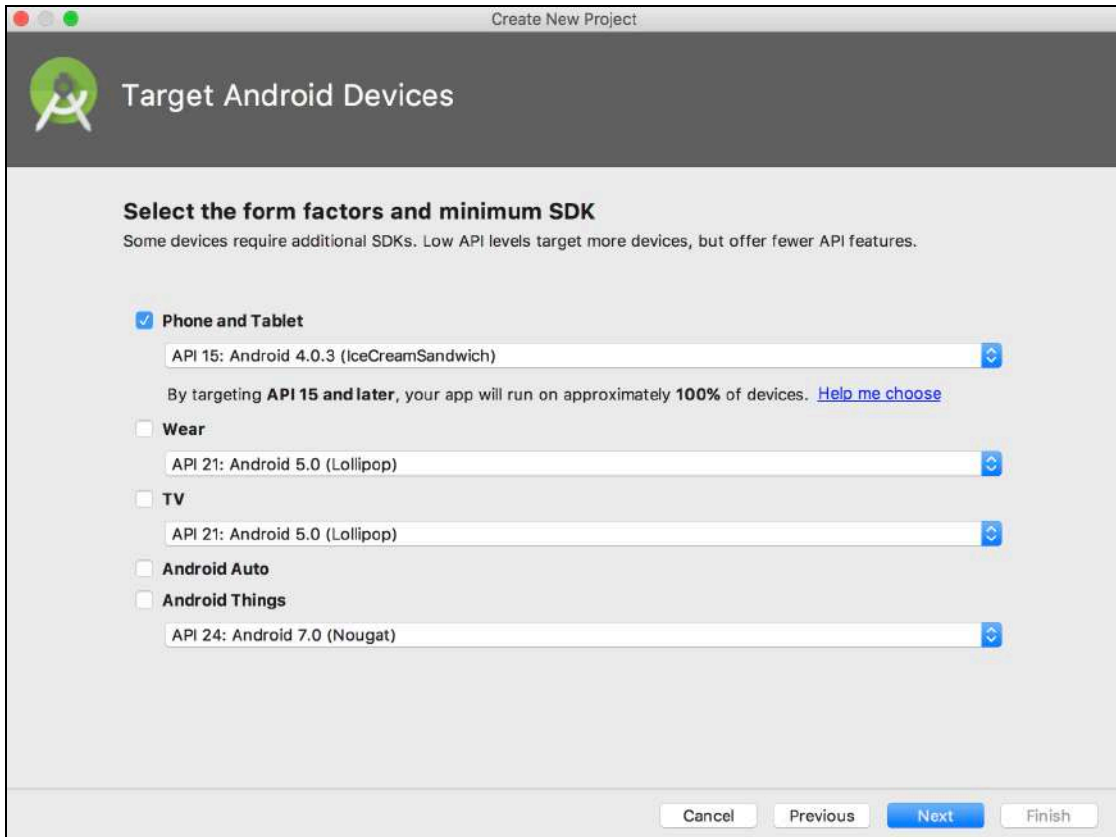
Ticking each of the various checkboxes associated with each type of device will instruct Android Studio to configure your starter project to contain everything you need to run a basic app on that device. You also have the ability to set the minimum SDK you want to support — in other words, the minimum version of Android you want your app to run on.

It can be hard to decide what minimum version of Android you want to support. More recent versions of Android support more features, but settling on one of those means you risk cutting off large numbers of users running older devices.

Conversely, choosing an older version means supporting more users, but having to make hard choices on whether to use older features only and avoid newer features found in more modern versions of Android. If only there was a way to help make this decision!

Fortunately, there is. Underneath the **Phone and Tablet** dropdown, click **Help me choose**, and you'll be presented with the Android Platform Distribution window.
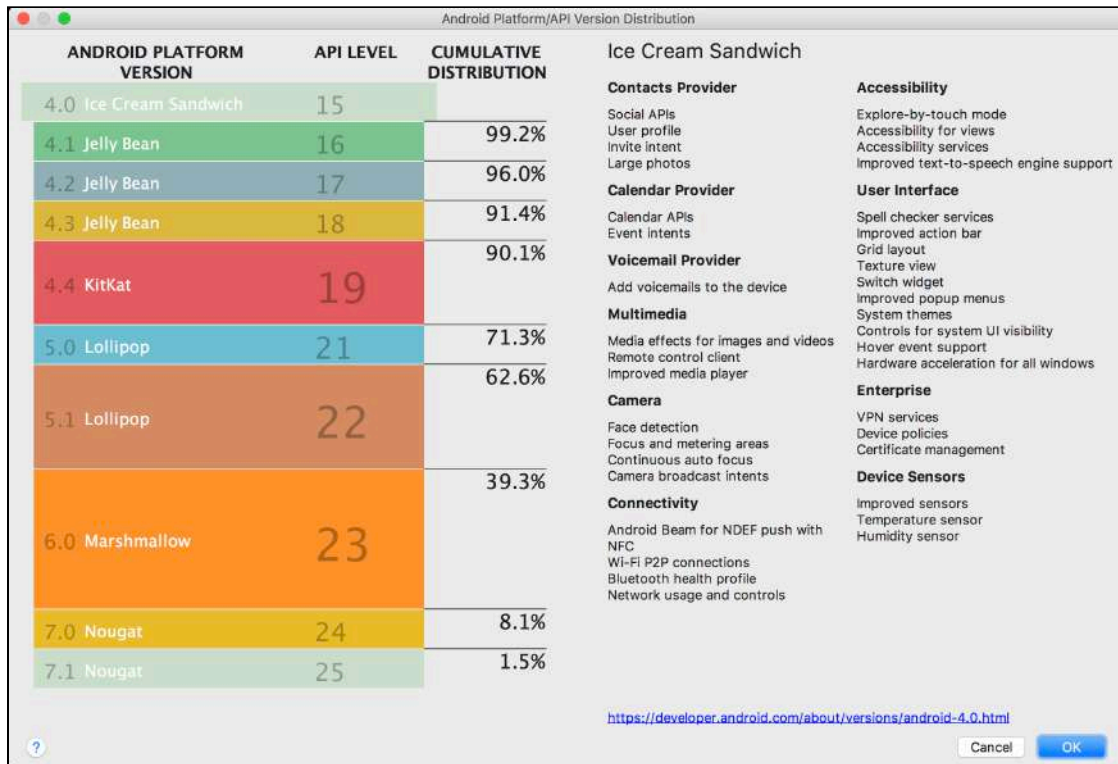
This is the **Android Platform Distribution** window. It shows a rough distribution of the versions of Android running throughout the world. This gives you the opportunity to make an informed decision on which versions of Android to support.

The distribution works on a cumulative basis, as shown by the percentages running alongside the colored boxes on the left. What this means is the earlier the Android version you choose to support, the more Android devices there are in the world that will be able to run your app.

Distribution isn't the only thing this window provides: It also provides a handy overview of what features each version of Android supports. Android Ice Cream Sandwich is selected by default, but if you click each of the colored boxes you can see what features each Android version provides.

If you need more in depth information, then the link in the bottom right of the window will take you to the About page on the Android developer site for that Android version.

This is an extremely useful page, and I recommend that you use it whenever you're trying to decide which versions of Android to support.

For now, you will use the default selected version of Android: Ice Cream Sandwich. Click **Cancel** to go back to the Target Android Devices screen. Then click **Next** in the bottom right to go to the next screen.

Creating an Activity

Now that you've set up the name of your app and chosen the version of Android you want to support, your next crucial choice is to choose what kind of **Activity** you want to start with.
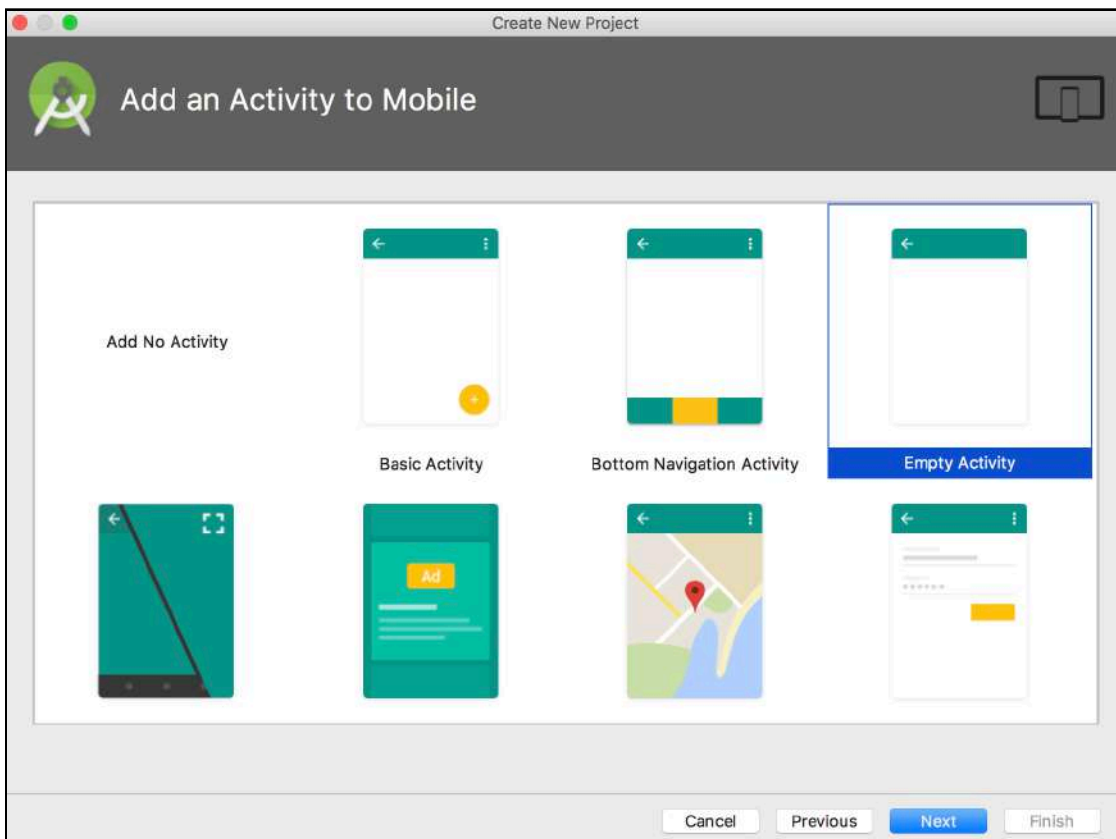
Android Studio provides you with a number of templates to help get you started as quickly as possible, such as an Activity that has a map embedded, or an activity with a basic login layout ready for you to customize. The list is quite extensive.

If you had elected to support other devices in your project such as Android Wear, TV or Auto, you would have multiple opportunities to choose the best template to start with for each device.

For now though, you only have one choice to make for your phone app. You want something simple so you can get started as quickly as possible. The **Basic Activity** will

suit your needs for reasons that will become clear later on. Click the **Basic Activity** image and then click **Next**.

Welcome to the final screen before you get to see your new project!
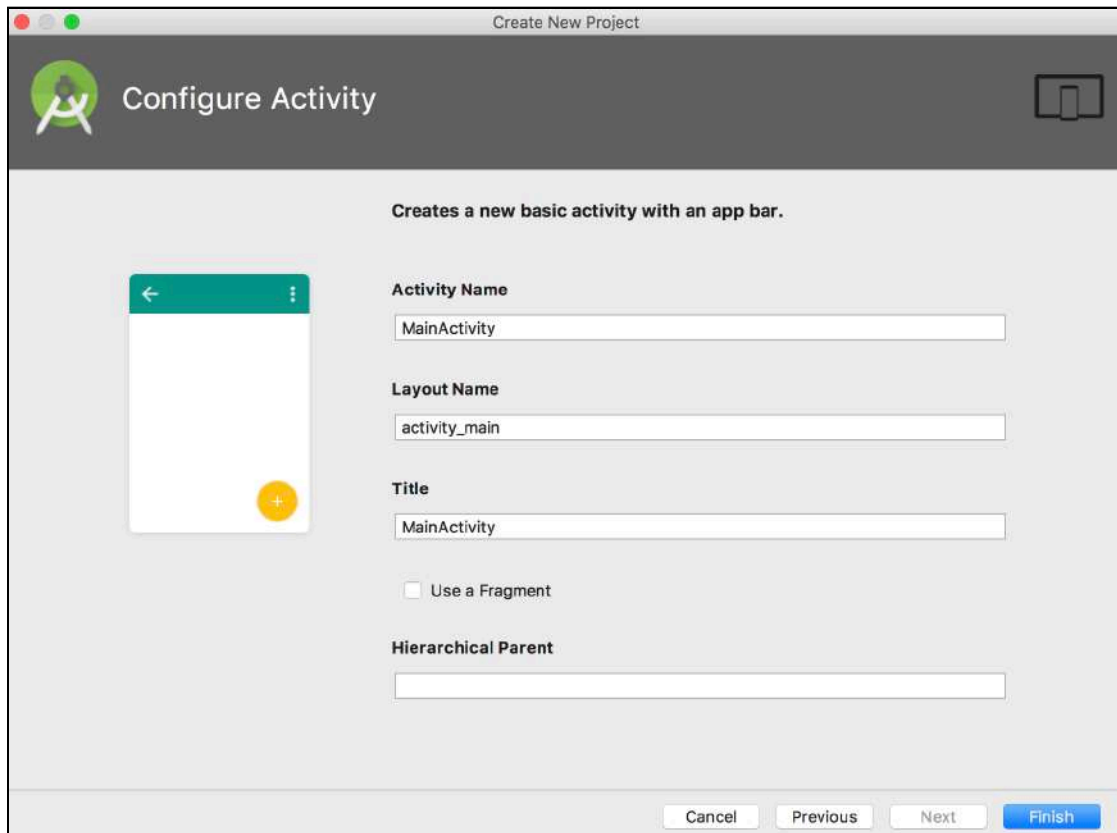
Now that you've chosen what kind of Activity you want to begin with, you just need to supply a little more information about the Activity.

The first text field is for the **Activity Name**. This will be the class name for your Activity where you will create all the logic for that screen. You want your Activity to be called **MainActivity** for now, so let's leave it as is.

The second text field is for the **Layout Name**. Remember, the Layout is the file used in association with an Activity to describe how the screen looks. Change this one to something more descriptive: Type **activity_list** into the textfield.

> **NOTE:** When creating layout files, convention dictates that the type of object the file is associated with is usually the first part of the name, followed by what it does. So similar to `activity_list.xml`, if you were creating a `FooView`, you would name a layout associated with it `view_foo.xml`.

The third text field is for the **Title** of your activity. This is the name shown in the colored bar at the top of your Activity screen. It makes sense for it to show the name of your app, so type in **ListMaker**.

That's all for this screen, but before you run off and click Finish, take a look at what's left on this screen that you didn't touch.

First, there's the **Use a Fragment** checkbox right below the title text field. Leave this unchecked, since you'll explore what **Fragments** are later in the section. For now, all you need to know is that Fragments allow you to split your Activity up into smaller independent pieces.

The final text field is called **Hierarchical Parent** and lets you provide the name of a parent Activity for your new Activity. When you set this, tapping the "Up" button on your Android device will automatically navigate from your current Activity to its parent Activity. Pretty handy! However, since this is the first Activity in the app, leave this blank.

Click **Finish**. Android Studio will take all your project settings and begin to create a new project for you.
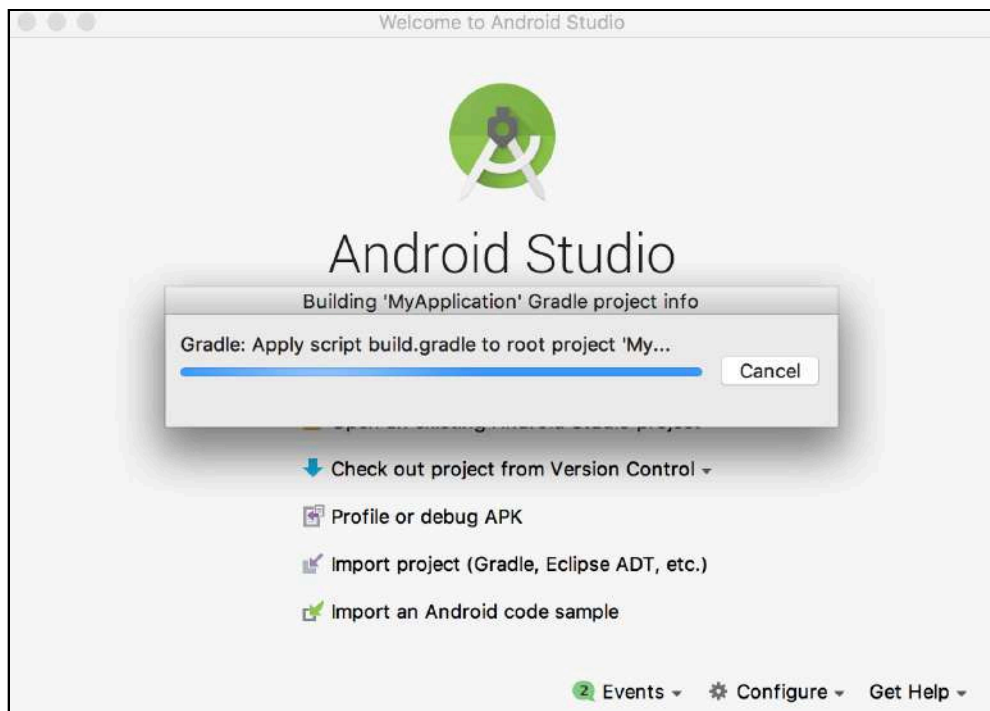
Once it's finished, you'll be presented with your project, and Android Studio will open up with your new layout ready for editing.
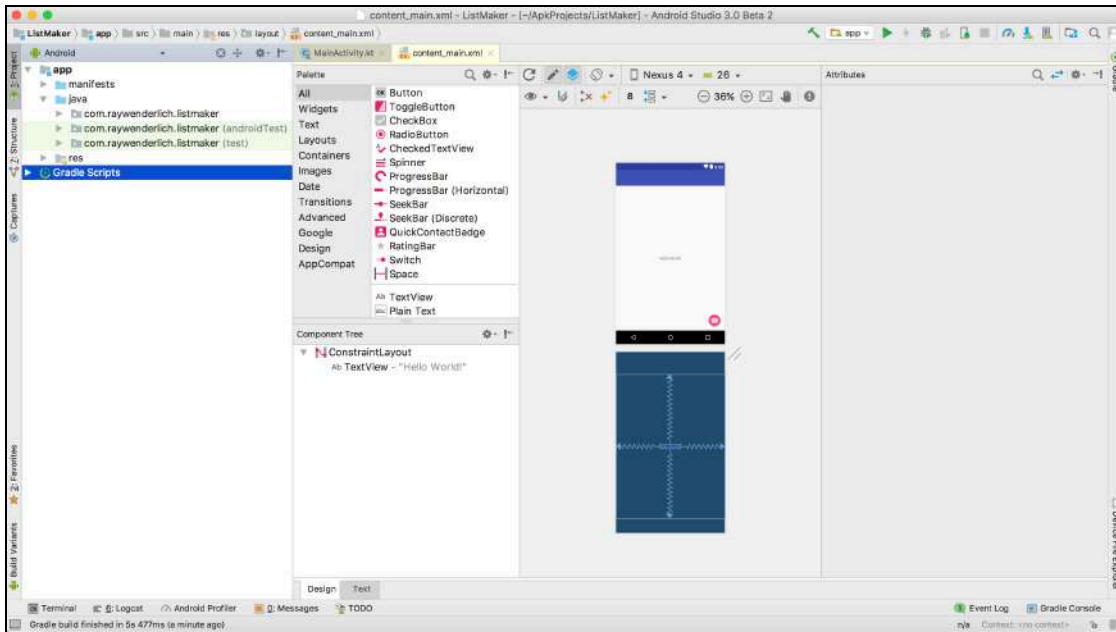
Where to go from here?

That's it for this chapter. As you've seen, Android Studio provides a sophisticated way to set up a new project with a number of templates to get you up and running as quickly as possible. Sample code is only a click away on the welcome screen, and you've got the ability to work with a number of exciting variants of Android with just a few clicks.

In the next chapter you'll begin to build the ListMaker app. Let's get to it!

7Chapter 7: RecyclerViews

By Darryl Bayliss

In this chapter, you're going to build a brand new app called **ListMaker**: an app to help you organize all of your to-do lists in one handy place.

Lists are a common visual design pattern in apps, allowing a developer to group together collections of information and display them in a way that allows users to scroll through and interact with each item in the list.
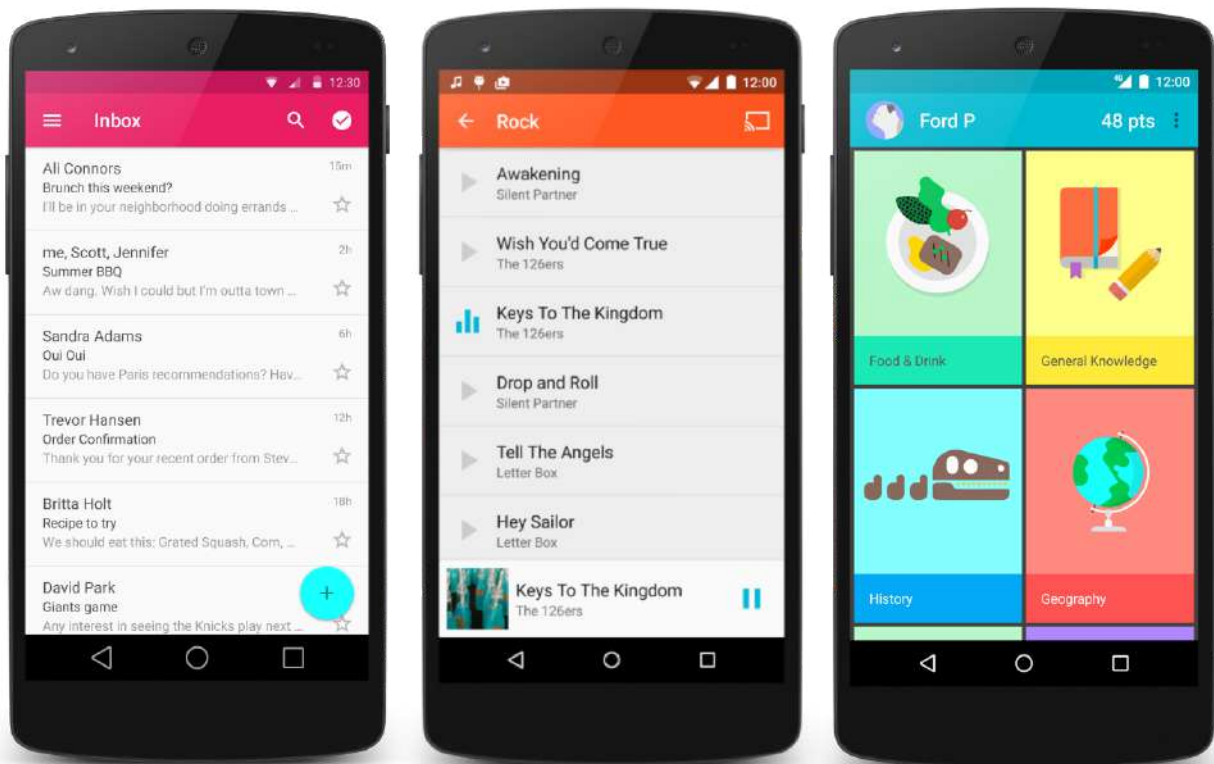
These apps are all using RecyclerView

An item in a list can be anything from a line of text; to something more complex, such as a video with comments below, as you can see in nearly any social media app today.

In Android development, the simplest way to implement lists in your app is to use a class named **RecyclerView**. As part of this chapter, you'll learn about the following:

1. How to get started with `RecyclerView`.

2. How to set up a `RecyclerView Adapter` to populate your list with data.

3. How to setup a `ViewHolder` to handle the layout of each item in your list.

Getting started

If you are following along with your own app, open it up. If not, don't worry. Locate the **projects** folder for this chapter and open the **ListMaker** app under the **starter** folder. The first time you open the project, Android Studio takes a few minutes to set up your environment and update dependencies.

Open up your newly created Android Studio project and examine the project structure. In particular, have a look at the following files:

• **MainActivity.kt**, found in the **java** folder.

• **activity_list.xml** and **content_main.xml**, found in the **layout** folder.

Kotlin (`.kt`) files drive the logic of your app. **MainActivity.kt** will contain some familiar-looking boilerplate code related to the Activity and Menu lifecycles.

In previous chapters, you learnt how to use layout files to build up the user interface of your app. However, there's two layout files in this project: **activity_list.xml** and **content_main.xml**. Previously, you only had one layout file from setting up an activity. Why are there two?

The answer can be found by looking at **activity_list.xml**. Open it up and examine the **Component Tree** to see what is going on:
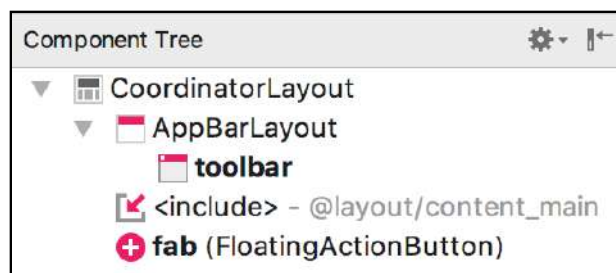
You have a **Toolbar** to display menu items you create, as well as a **FloatingActionButton**. You've used buttons before, so there's no surprises so far.

Keep scanning, and you'll see a final component named **include**. This is where **content_main.xml** comes into play: the **activity_list.xml** layout includes the layout defined in **content_main.xml**.

While it might seem odd to take this approach, this can be useful when you use a layout in multiple places in your app, or when your layout is complex enough to benefit from being split up into multiple files.

Open **content_main.xml** and have a look at its contents:
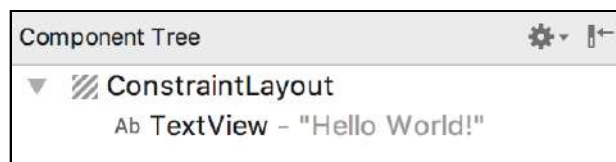
Nothing but a single `TextView`! That seems straightforward enough.

Now that you know how everything is strung together, click **Run app** to see that glorious `TextView`:
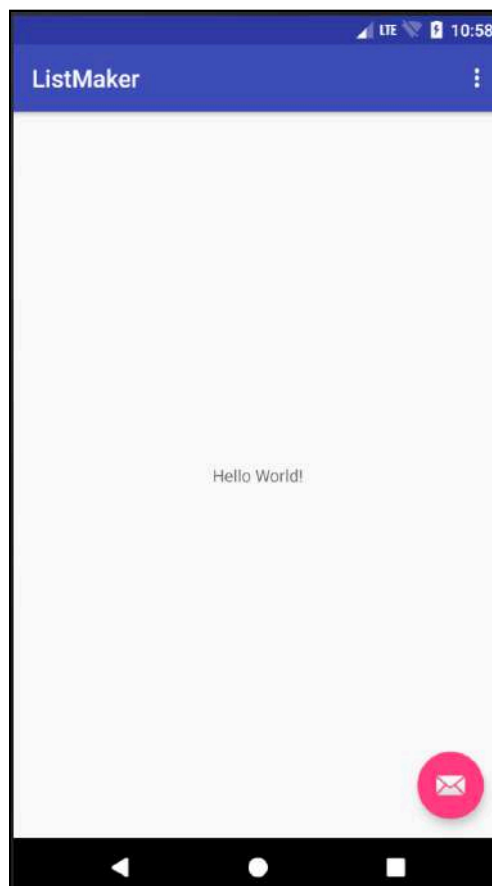
There's something important missing from **ListMaker**: lists! There isn't any way to show a list, let alone your master list of lists. It's like *Inception*, but…*Listception* instead. You'll fix this in the next section.

Adding a RecyclerView

Open **content_main.xml**. First, select the `TextView` and delete it. Next, go to the **Palette** and click **AppCompat**:
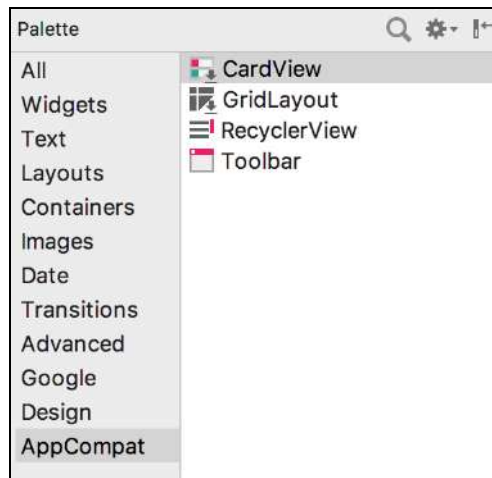
Click and drag a **RecyclerView** from the list of components into the middle of the layout:
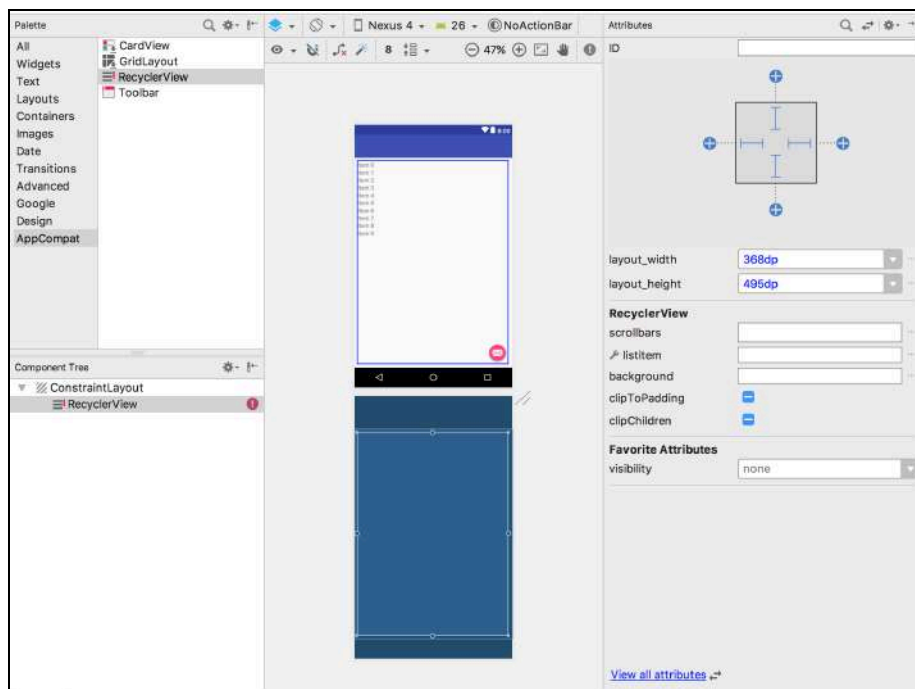
Once you've dropped the `RecyclerView` into the Layout, head over to the **Attributes** window and change the **ID** to **lists_recyclerview**. This will let you reference the `RecyclerView` in your Kotlin file. Next, change the **layout_width** and **layout_height** to **match_parent**.
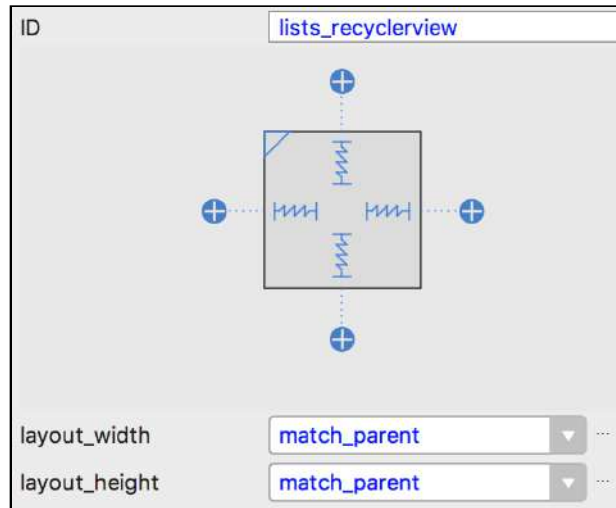
Now it's time to use the `RecyclerView` you just added.

The components of a RecyclerView

`RecyclerView` let you display large amounts of data. In turn, each piece of data is treated as an item within the `RecyclerView`. Each of these items in turn make up the contents of the `RecyclerView`. Sounds a lot like a list, doesn't it?

RecyclerView has two required components that help display your list of items: **Adapter** and **ViewHolders**. The following diagram shows how these components work together:
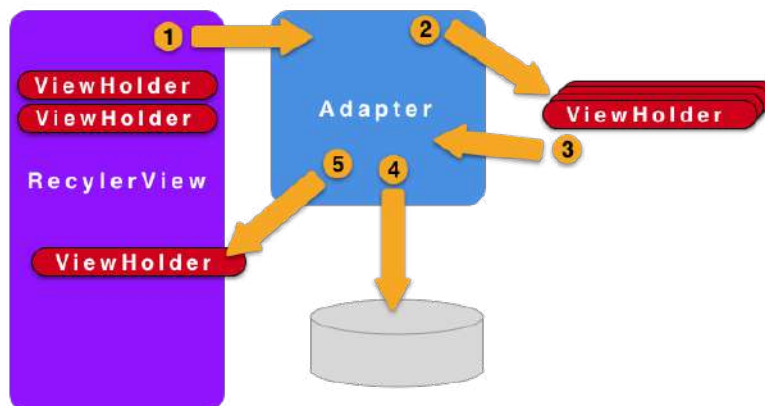
Let's break down the flow of each component:

1. The `RecyclerView` asks the `Adapter` for an *item*, or a `ViewHolder` at a given position.

2. The `Adapter` reaches into a pool of `ViewHolders` that have been created.

3. Either a a new `ViewHolder` is returned, or a new one is created.

4. The `Adapter` then binds this `ViewHolder` to a data item at the given position.

5. The `ViewHolder` is returned back to the `RecyclerView` for display.

In general, **Adapters** give your `RecyclerView` the data it wants to show. They have a clever way to calculate how many rows of data you want to show, which you'll cover shortly.

ViewHolders are the visual containers for your item. Think of them as cells in the table. This is where you tell your `RecyclerView` what each item should look like. These are basically little tiny layout items used to display the data at any given position in the list of data.

As you scroll through a `RecyclerView`, instead of creating new `ViewHolders`, RecyclerView will *recycle* `ViewHolders` that have moved offscreen and populate them with new data, ready to be shown at the bottom of the list. This process repeats endlessly as you scroll through your `RecyclerView`. This *recycling* of `ViewHolder` to display list items helps to avoid *janking* in your app.

> **Note**: Janking is common terms used to refer to dropped or missed frames while rendering. As an app user, you might have experienced stuttering while scrolling long lists. This is affectionately known as jank.

That concludes the whirlwind tour of `RecyclerView`! Now it's time to get coding.

Hooking up a RecyclerView

Open up **MainActivity.kt** and add the following line to the top, just above `onCreate(savedInstanceState: Bundle?)`:

```
lateinit var listsRecyclerView: RecyclerView
```

Here you use the `lateinit` keyword to tell the compiler that a `RecyclerView` is going to be created sometime in the future.

Next, add the following lines to the bottom of `onCreate()`:

```
// 1
listsRecyclerView = findViewById<RecyclerView>(R.id.lists_recyclerview)
// 2
listsRecyclerView.layoutManager = LinearLayoutManager(this)
// 3
listsRecyclerView.adapter = ListSelectionRecyclerViewAdapter()
```

Here's what you're doing:

1. Set `listsRecyclerView` by referencing the ID of the `RecyclerView` you set up in your layout.

2. Let the `RecyclerView` know what kind of layout you want to present your items in. This is similar to the Layouts you can use with your XML layouts, and you'll need something to arrange your items in a linear format. The `LinearLayoutManager` will work perfectly for this. You also pass in the Activity so the layout manager can access its `Context`.

> **Note**: LinearLayoutManager isn't the only layout provided by RecyclerView. Out of the box, RecyclerView provides the GridLayoutManager and StaggeredGridLayoutManager.

3. This lets the `RecyclerView` know what to use as its adapter.

You'll notice that this will show an error in Android Studio. This is because `ListSelectionRecyclerViewAdapter` doesn't exist. You'll create this now.

Setting up a RecyclerView Adapter

Right-click on the **com.raywenderlich.listmaker** package in the **Project Navigator**. In the floating options that appear, hover over **New**. In the next set of options that appear, click **Kotlin File/Class**.
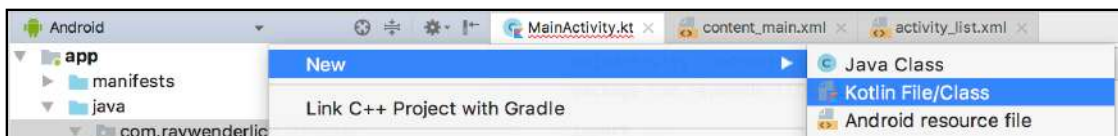
In the **Name** text field, type **ListSelectionRecyclerViewAdapter** and change the **Kind** dropdown to Class. Click **OK**.
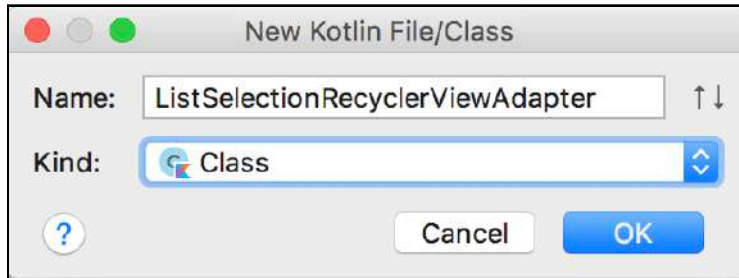
Android Studio will create your class for you. While you're at it, create a `ViewHolder` class as well.

Once again, create a new Kotlin class and name it **ListSelectionViewHolder**.
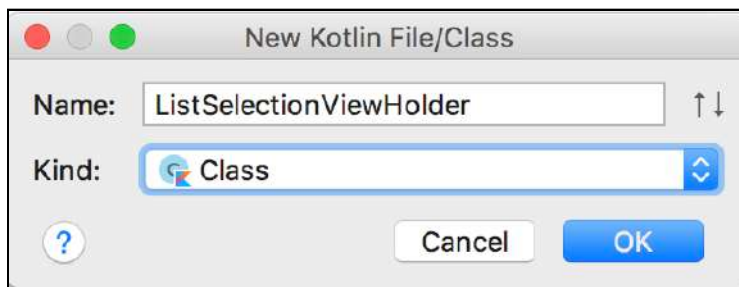
Now you'll turn these classes into recycling machines! In **ListSelectionViewHolder.kt**, add the following primary constructor to the class so you can pass in the View for your ViewHolder and have it extend `RecyclerView.ViewHolder`:

```
class ListSelectionViewHolder(itemView: View?) :
  RecyclerView.ViewHolder(itemView) {
}
```

Open up **ListSelectionRecyclerViewAdapter.kt** and extend the class so it inherits from `RecyclerView.Adapter<ListSelectionViewHolder>()`:

```
class ListSelectionRecyclerViewAdapter :
  RecyclerView.Adapter<ListSelectionViewHolder>() {
}
```

Here you pass in the name of the `ViewHolder` you want your `RecyclerView` Adapter to use. This makes the `RecyclerView` aware of the `ViewHolder` so you can reference it in a few methods you'll implement shortly.

Notice that the name of your class is now underlined with red. Move your mouse cursor over it and Android Studio will tell you the reason for this:
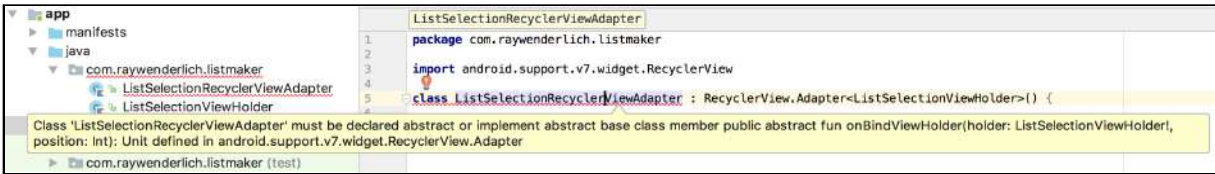
This tells you that since your class inherits from `RecyclerView.Adapter`, it needs to implement some further methods so it knows what to do when used in conjunction with a RecyclerView. Fortunately, this is easy to do.

With your cursor over the class name, press **Option + Enter** to get a selection of options. These will help resolve the issue Android Studio has raised.

Note: This keystroke assumes you're using a Mac for Android development; however, Windows and Linux versions of Android Studio provide an equivalent shortcut through **Alt + Enter**.
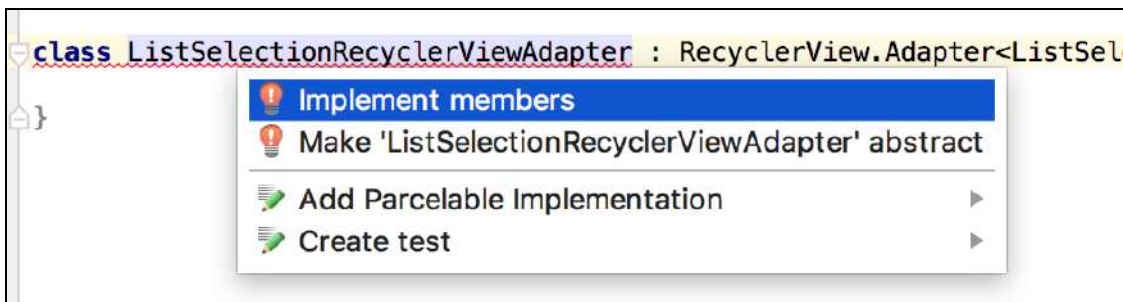
Click the **Implement Members** option, and a new window will appear with options for various methods to implement. Since your Recycler Adapter needs each one, you'll take them all. Make sure `onCreateViewHolder()` is highlighted, then **Shift + click** on the bottom most available member.
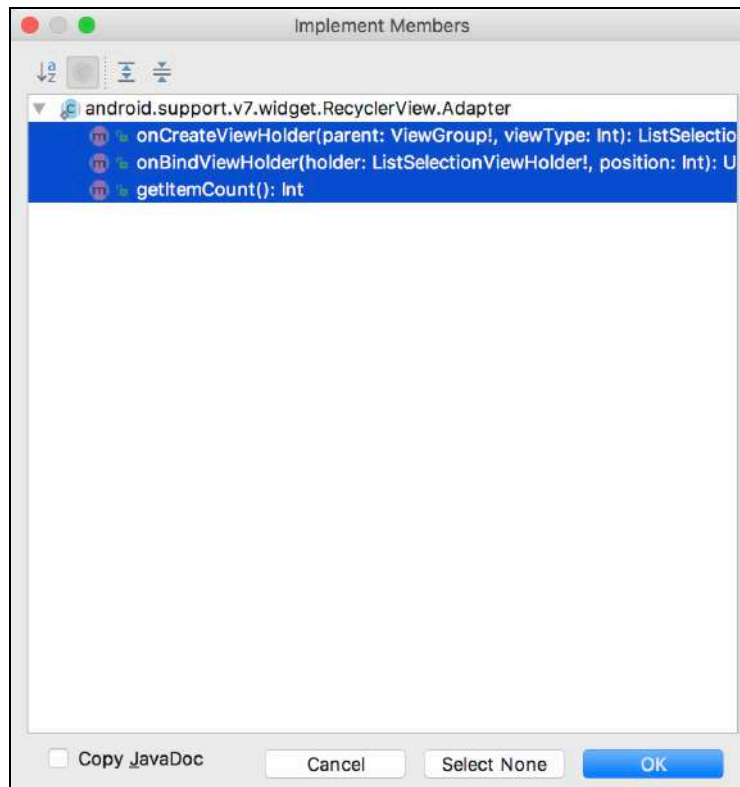
Finally, click **OK** and Android Studio will do the rest of the work for you. Make sure those methods are implemented in your `ListSelectionRecyclerViewAdapter`:

```kotlin
class ListSelectionRecyclerViewAdapter :
        RecyclerView.Adapter<ListSelectionViewHolder>() {

  override fun onCreateViewHolder(parent: ViewGroup?,
                                  viewType: Int):
                                  ListSelectionViewHolder {
    TODO("not implemented") //To change body of created
    //functions use File | Settings | File Templates.
  }

  override fun onBindViewHolder(
          holder: ListSelectionViewHolder?,
          position: Int) {
    TODO("not implemented") //To change body of created
    //functions use File | Settings | File Templates.
  }

  override fun getItemCount(): Int {
    TODO("not implemented") //To change body of created
    //functions use File | Settings | File Templates.
  }
}
```

Filling in the blanks

Now that you've got the basics of your Adapter and `ViewHolder` set up, it's time to put the pieces together. First, you need some sort of content for your `RecyclerView` to show. For now, you'll add some mock titles to show off the `RecyclerView`.

You also need to create a Layout for your `ViewHolder` so the `RecyclerView` knows how each item within it should look. Finally, you need to bind your titles to the `ViewHolder` at the right time depending on what position it has within the `RecyclerView`.

You'll implement those mock list titles first. In your `ListSelectionRecyclerViewAdapter` class, add the following value above the methods you've implemented:

```kotlin
val listTitles = arrayOf("Shopping List", "Chores", "Android Tutorials")
```

Here you're creating an array of strings that will be used for your list titles. In future chapters, you'll change this to something more sophisticated. For now, an array will do.

`getItemCount()` determines how many items the `RecyclerView` will have. You want the size of your array to be the size of your `RecyclerView`, so you'll return that.

In `getItemCount()`, return the size of your array like so:

```kotlin
override fun getItemCount(): Int {
  return listTitles.size
}
```

Next you need to create the Layout that your `ViewHolder` will use for each item in the RecyclerView.

Creating the ViewHolder

In the Project Navigator on the left, right-click on the layout folder and create a new layout:
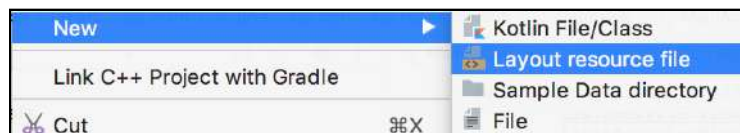
In the new window that appears, enter **list_selection_view_holder** into the **File Name** text field, enter **LinearLayout** for the **Root element**, and click **OK**.
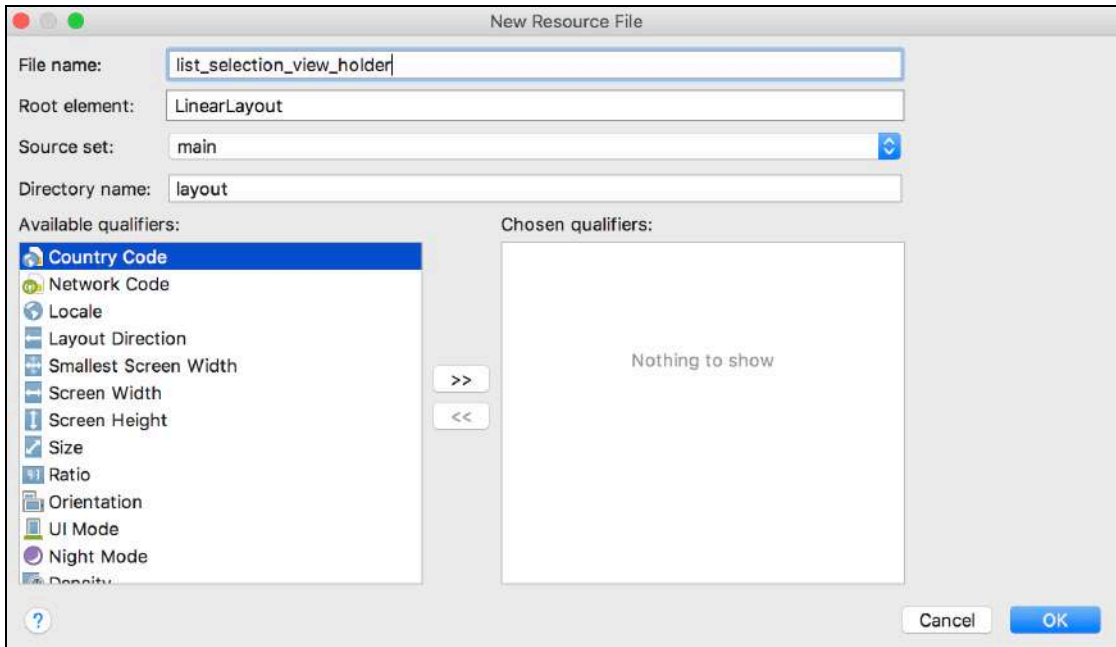
Android Studio will open your new layout, ready for you to add the Views you want your `ViewHolder` to contain. You need two TextViews here: one to tell you the position of the list in the `RecyclerView`, and one to tell you the name of the list.

With the **Design** tab open, drag a TextView on to the layout.
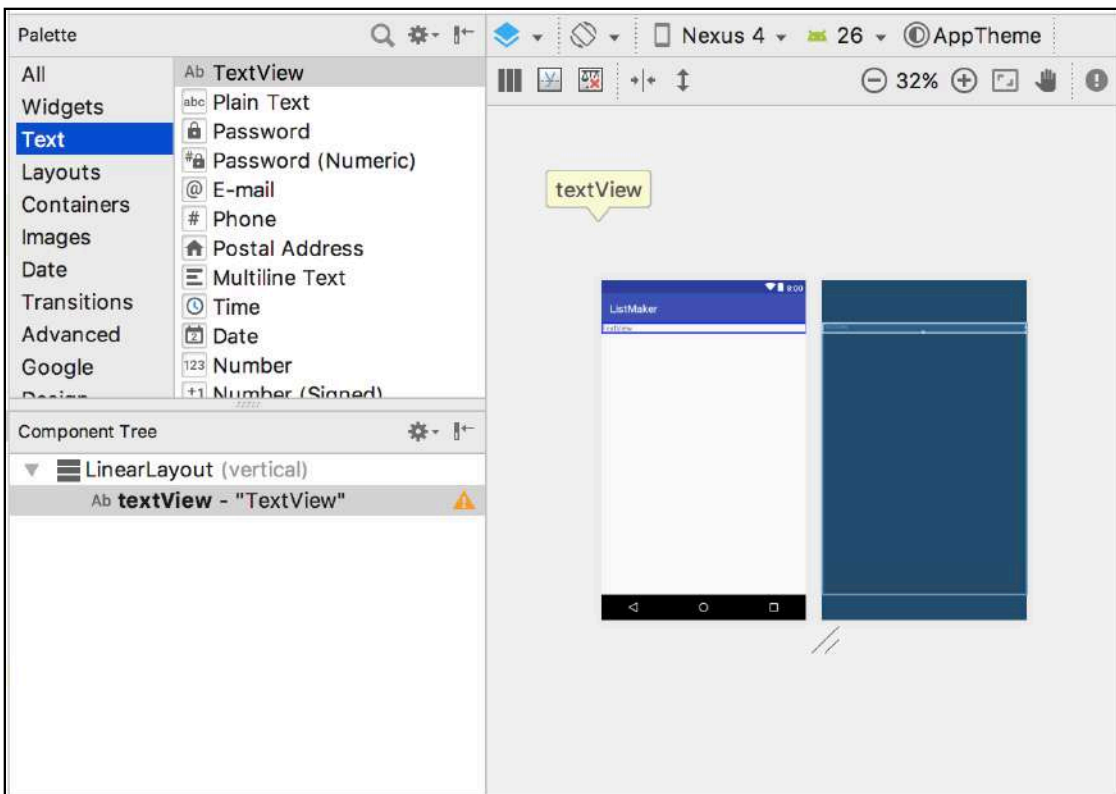
In the Attributes window in the right of Android Studio, change the **ID** of the TextView to **itemNumber**. Also change the **layout_width** and **layout_height** attributes to **wrap_content** and remove the placeholder text from the **text** attribute:

To make sure the text isn't sitting right next to the edge of the screen, you also want to give it a bit of space on its left edge. Click the **View All Attributes** button at the top of the Attributes window:

This brings up a list of attributes you can change for your TextView. Feel free to take a look at what attributes you can change. There's quite a lot of them!

You need to set the **all** parameter of the **Layout_Margin** attribute to add a bit of padding to this text view. First find the **Layout_Margin** attribute and click the arrow next to it to reveal a drop down for each of the parameters:
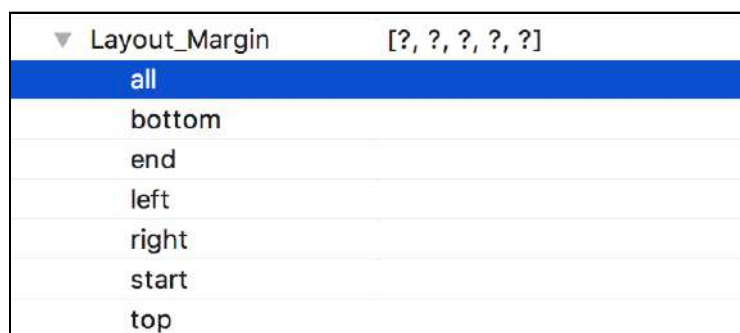
In the **all** text field, type **16dp**. This tells the `TextView` to pad itself by 16 **density pixels** (dp) on all sides.

> **Note**: What's a density pixel? They're a virtual unit of measurement Android uses when laying out your View relative to the size of the device screen. Because devices have many different screen sizes, trying to use absolute pixels isn't feasible as your screen will render differently from device to device. To learn more, head on over to https://developer.android.com/guide/practices/screens_support.html.

With that done, you've set up your first `TextView`! Time to create the next one. Repeat the process: drag another `TextView` into the layout so it's underneath your first `TextView`.
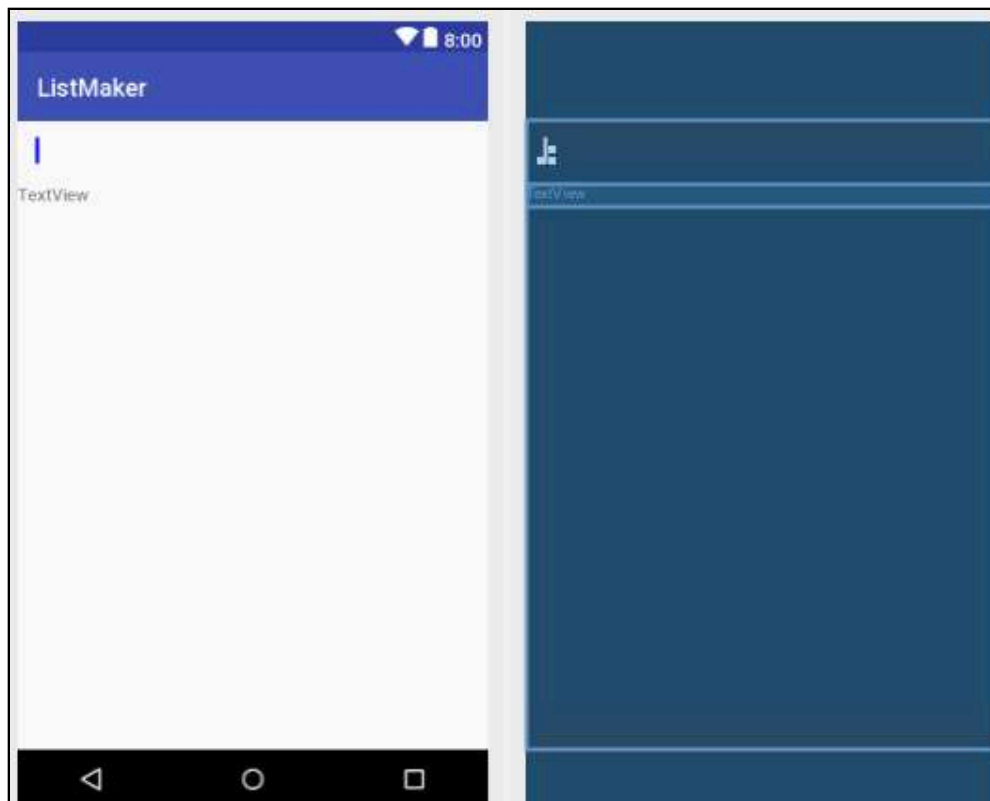
Then just as you did before, change the **ID** of your `TextView`, this time to **itemString**. Change the **layout_width** and **layout_height** to **wrap_content**. Remove the placeholder text from the **text** attribute and change the **all** parameter in the **Layout_Margin** attribute to **16dp**.

You're nearly done with your layout — there's just one more thing to do.

Currently, your TextViews are laid out in a vertical orientation. A horizontal orientation will better suit your app. Therefore, you'll need to change some attributes on the **LinearLayout** your layout uses.

To change this, first click on the **LinearLayout** in the Component Tree window:
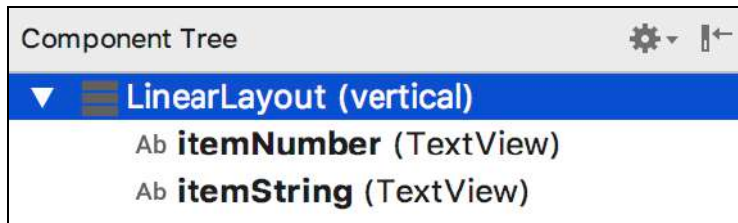
Then, in the Attributes Window, click the dropdown button on the orientation attribute and select **horizontal**.
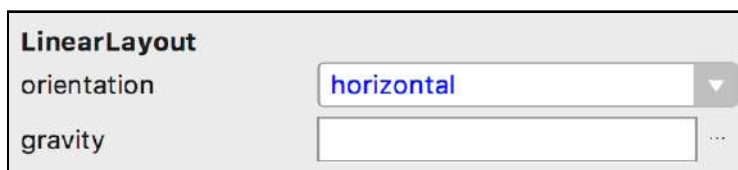
You also need to change the **layout_width** and **layout_height** attributes to **wrap_content** to make your ViewHolder only as big as it needs to be:

With that done, you're now ready to use your Layout! Head back to **ListSelectionRecyclerViewAdapter.kt** and change `onCreateViewHolder()` to the following:

```kotlin
override fun onCreateViewHolder(parent: ViewGroup?,
          viewType: Int): ListSelectionViewHolder {
    // 1
    val view = LayoutInflater.from(parent?.context)
          .inflate(R.layout.list_selection_view_holder,
            parent,
            false)
    // 2
    return ListSelectionViewHolder(view)
}
```

The method does the following two things:

1. First, it uses a **LayoutInflater** object to create a layout programmatically. It uses the context of the Activity to create itself, and then attempts to inflate the layout you want by passing in the layout name and the parent **ViewGroup** so the view has a parent it can refer to. Don't worry about the Boolean value; this is fine as `false` for now.

> **Note**: LayoutInflater is a system utility used to instantiates a layout XML file into its corresponding View objects.

2. Your view is created from the layout and returned from the method. Your RecyclerView Adapter then begins to populate the views in your layout with the relevant information for each item.

Binding data to your ViewHolder

Now that you've created a ViewHolder, you just have to bind your list titles to your ViewHolder. To do this, you need to know what Views to bind your data to. You've already created the TextFields in your ViewHolder layout, but you haven't yet referenced these in code yet as you've done in the past.

Add the following properties to your **ListSelectionViewHolder.kt** class so your ViewHolder has references to your new TextViews:

```kotlin
  val listPosition = itemView?.findViewById<TextView>(R.id.itemNumber) as
TextView

  val listTitle = itemView?.findViewById<TextView>(R.id.itemString) as
TextView
```

Hop back into **ListSelectionRecyclerViewAdapter.kt** and change `onBindViewHolder()` to the following:

```kotlin
override fun onBindViewHolder(holder: ListSelectionViewHolder?, position:
Int) {
  if (holder != null) {
    holder.listPosition.text = (position + 1).toString()
    holder.listTitle.text = listTitles[position]
  }
}
```

This method binds the desired data to the ViewHolder at the appropriate position. This will be called repeatedly as you scroll through your RecyclerView.

For each call, you check the holder passed in to make sure it isn't `null`. It's possible something could have gone wrong during the setting up of the ViewHolder, so it's best to guard against that possibility.

If you have a valid ViewHolder, then you take the TextViews you created in your ViewHolder and populate them with their position in the list and the name of the list from the `listTitles` array.

The moment of truth

Finally! After much coding, you can see the fruits of your labors. Click the **Run App** button at the top of Android Studio and see what happens.
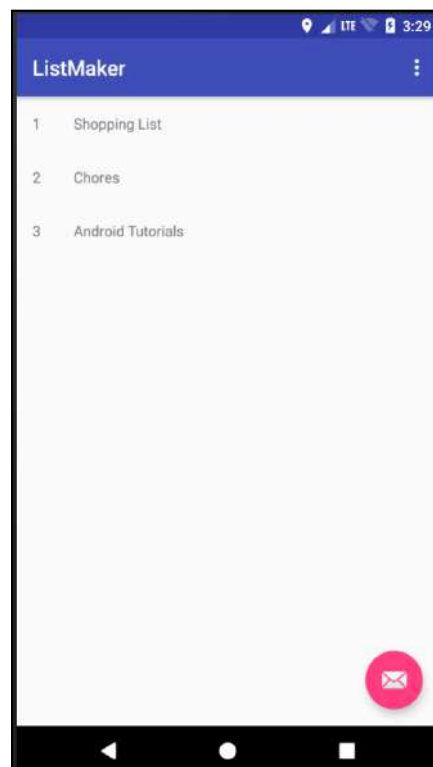
Fantastic — you now have a list of titles and the position they hold in the RecyclerView. Great job!

Where to go from here?

There are a lot of moving pieces required to use `RecyclerView` to display a list of data. However, don't be afraid to use `RecyclerView`, as they are an essential construct for creating Android apps that provide fluid and intuitive user experiences. They are as common in apps as Buttons and TextViews!

If you want to learn more about `RecyclerView`, check out the documentation on the developer website https://developer.android.com/guide/topics/ui/layout/recyclerview.html. It dives deeper into the inner workings of `RecyclerView` and even describes how to animate changes to list items.

If you're still looking for more, check out the tutorial on the Ray Wenderlich site https://www.raywenderlich.com/170075/android-recyclerview-tutorial-kotlin which shows you how to use different LayoutManagers, and how to swipe to delete items in your list!

Chapter 8: SharedPreferences

By Darryl Bayliss

In the previous chapter, you set up your Activity to use the powerful `RecyclerView`. You're still not yet to the point where you can use the app to track your lists, since it only shows a few hardcoded titles to act as placeholders.

In this chapter, you'll add functionality to ListMaker to create, save, and delete lists. You'll learn what SharedPreferences are and how you can use them to save and retrieve user data.

Getting started

If you are following along with your own app, open it and keep going with it for this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **ListMaker** app under the **starter** folder. The first time you open the project, Android Studio takes a few minutes to set up your environment and update dependencies.

Open the **ListMaker** project and run the project in the emulator.
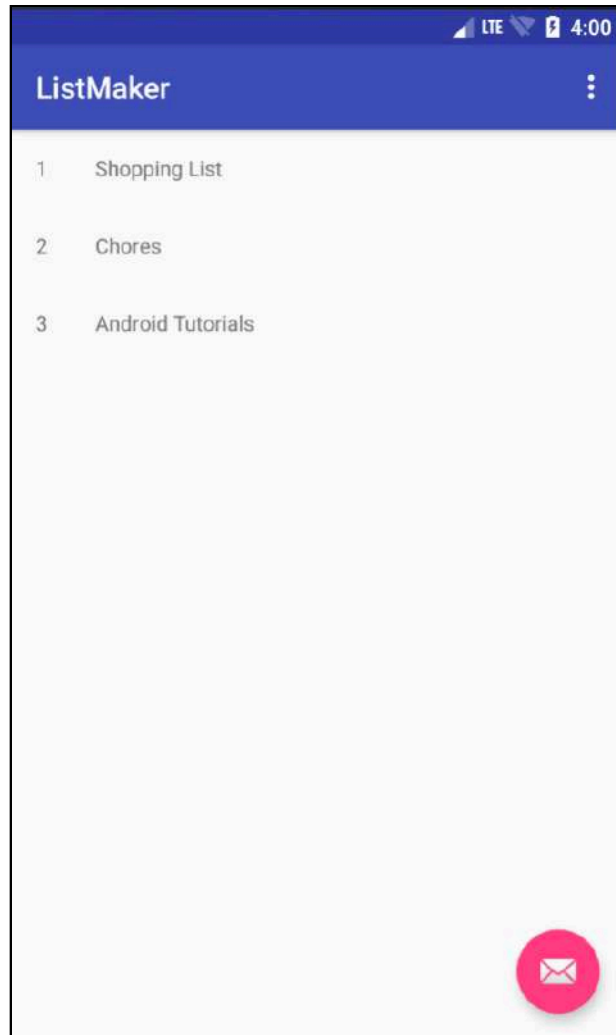
Notice the round pink button at the bottom right? That's called a **Floating Action Button**, better known as a **FAB**. You'll often use a FAB to highlight an important action on the screen. Since creating lists is the most important action in your app, it makes sense to use that button to add new lists.

First, you'll have to pick a more appropriate icon for the button, since an envelope doesn't convey the action behind the button. Open the **activity_list.xml** layout, and in the **Component Tree** window, select the Floating Action Button.
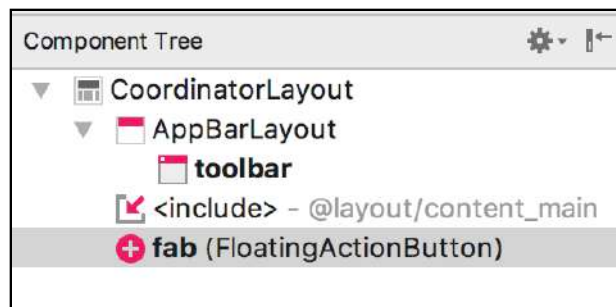
In the **Attributes** window on the right hand side of Android Studio, find the **srcCompat** text field. This is where you assign the image to the button. Right now, it should have a value of **@android:drawable/ic_dialog_email**.

Change the value in the **srcCompat** text field to **@android:drawable/ic_menu_add** and press Enter. The image in the FAB will change to a more appropriate plus sign icon.

Now that users will understand what the button is for, you'll need to add add some code to let the users create a new list.

Adding a Dialog

Tapping the FAB in **ListMaker** will open a dialog to let users enter the name of the list they wish to create. The dialog will also contain some labels to prompt users on what to enter in the field. Instead of hardcoding the dialog strings in the app, open **strings.xml** and add the strings as shown below:

```
<string name="name_of_list">What is the name of your list?</string>
<string name="create_list">Create</string>
```

It's a good idea to keep these in a separate **strings.xml** file in case you want to localize your app for different languages later on.

Open **MainActivity.kt** and add the following method to the bottom of the file:

```kotlin
private fun showCreateListDialog() {
  // 1
  val dialogTitle = getString(R.string.name_of_list)
  val positiveButtonTitle = getString(R.string.create_list)

  // 2
  val builder = AlertDialog.Builder(this)
  val listTitleEditText = EditText(this)
  listTitleEditText.inputType = InputType.TYPE_CLASS_TEXT

  builder.setTitle(dialogTitle)
  builder.setView(listTitleEditText)

  // 3
  builder.setPositiveButton(positiveButtonTitle, { dialog, i ->
    dialog.dismiss()
  })
```

```
  // 4
  builder.create().show()
}
```

Here's what you're doing in the previous code snippet:

1. Retrieve the strings you defined in **strings.xml**.

2. Create an `AlertDialogBuilder` to help construct your Dialog and an `EditText` View
    as well. This is the input field where the user will enter the name of the list.

    You also change the `inputType` of your `EditText` to `TYPE_CLASS_TEXT`. Specifying the
    input type tells Android what the most appropriate keyboard form is. In this case,
    you want a text-based keyboard and not a numerical one, since the numerical one
    would be more appropriate for entering things like telephone numbers.

    You then set the title of your Dialog by calling `setTitle`. You also have to pass in the
    content View of your dialog. In this case, you pass in the `EditText` View by calling
    `setView`.

3. Inform the Dialog Builder that you would like a **positive button** to be added to the
    Dialog, which will tell the Dialog that a positive action has occurred and something
    should happen. You can also use **negative buttons** for doing things that you
    consider negative in your app, such as canceling an action.

    In this case, a positive button makes sense since you're creating something. You
    pass in `"Create"` as the label you want for your button and then implement a
    `onClickListener` for your button. For now, you'll simply dismiss the dialog. You'll
    handle the resulting actions behind the button in the next section.

4. Finally, you instruct `DialogBuilder` to create your dialog and display it onscreen.

Now that you have some code to show the dialog, you'll need to call it when the user
taps the FAB. Locate the `setOnClickListener` called on the `fab` inside of `onCreate`.
Replace the contents of the OnClickListener with the call to your new dialog method:

```
fab.setOnClickListener { view ->
    showCreateListDialog()
}
```

Run your app and click on the pink FAB in the bottom right of your screen. You should see your create list dialog appear as expected.
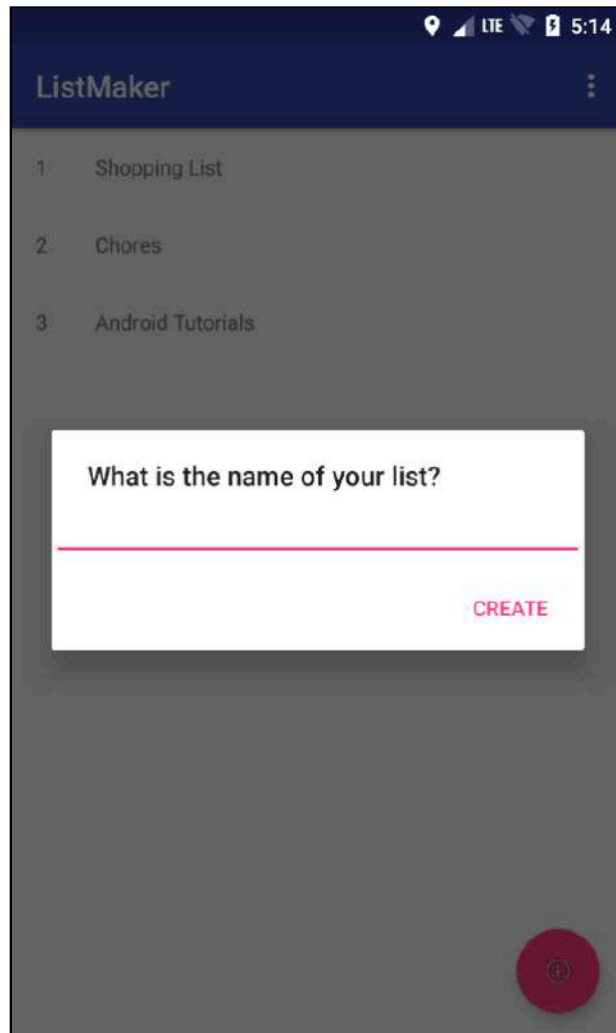
Type in a name for your list, click **Create** and...nothing happens. That's because you need to add some code to handle the creation of the list inside the `onClickListener` of the positive button for the Dialog.

Creating a list

Before you can begin to work with a list, you need to let your App know what exactly a list...is! You'll start by creating a model for a list which you'll use throughout the app.

Right-click on the package containing all of your **.kt** files; you'll find it in the **com.raywenderlich.listmaker** folder. In the options that appear, select **New ▸ Kotlin File/Class**:
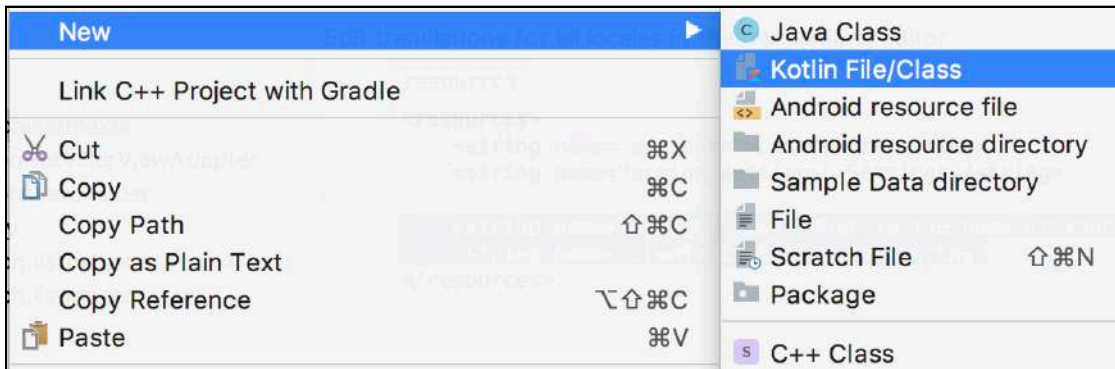
In the window that appears, name your new Kotlin file **TaskList**, change the kind to **Class** and select **OK**.
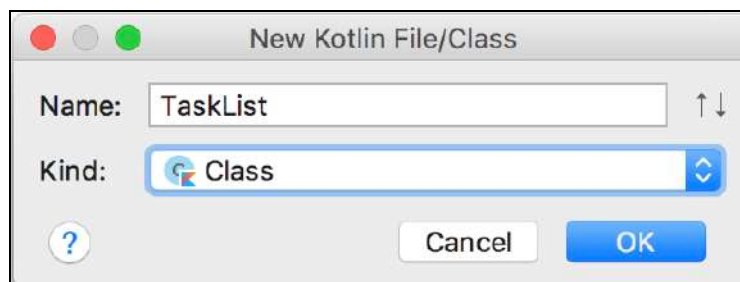
Android Studio will create and display your new Class. Next, add a primary constructor to TaskList so it can be given a name and a list of associated tasks:

```
class TaskList(val name: String, val tasks: ArrayList<String> =
ArrayList<String>()) {

}
```

Next you need a way to save your list to the device. This is where **SharedPreferences** comes into play.

SharedPreferences lets you save small collections of key-value pairs that you can retrieve later. If you need a way to quickly save small bits of data in your app, SharedPreferences is one of the first solutions you should consider.

Behind the scenes, SharedPreferences writes your key-value pairs to a single file, or you can even configure it to write to multiple files for more complex apps. You can also place access controls for other apps around your own app's SharedPreferences store, if you think that other apps might want to access your data.

> **Note**: SharedPreferences is a quick way to persist and retrieve data, but in later chapters you'll see that there are some better alternatives to SharedPreferences when you have complex data needs.

You'll need another class to manage your lists, so create a new Class and name it **ListDataManager**. Create the primary constructor for your class as follows:

```
class ListDataManager(val context: Context) {
  fun saveList(list: TaskList) {
    // 1
    val sharedPreferences =
PreferenceManager.getDefaultSharedPreferences(context).edit()
    // 2
    sharedPreferences.putStringSet(list.name, list.tasks.toHashSet())
    // 3
    sharedPreferences.apply()
  }
}
```

You're passing a `Context` into `ListDataManager` and adding a method named `saveList(list: TaskList)` to persist your list. Here's what's going on:

1. Get a reference to the app's default SharedPreference store via `PreferenceManager.getDefaultSharedPreferences(context)`. With the SharedPreference objects you get back, call `.edit()` to get a `SharedPreferences.Editor` instance. This lets you to write key-value pairs to SharedPreferences.

2. Write `TaskList` to SharedPreferences as set of Strings, passing in the name of your list as the key. Since the tasks in `TaskList` is an array of strings, we can't store it directly in a string. So you convert the tasks in `TaskList` to a **HashSet** which we can then pass as the value to be saved.

3. You then instruct your SharedPreferences Editor instance to `apply` the changes. This writes the changes to your SharedPreferences file.

That takes care of saving your lists, but that's only half the solution. You also need a way to retrieve your lists. Add the following method underneath `saveList`:

```
fun readLists(): ArrayList<TaskList> {
  // 1
  val sharedPreferences =
PreferenceManager.getDefaultSharedPreferences(context)
  // 2
  val sharedPreferenceContents = sharedPreferences.all
  // 3
  val taskLists = ArrayList<TaskList>()
```

```
  // 4
  for (taskList in sharedPreferenceContents) {
      val itemsHashSet = taskList.value as HashSet<String>
      val list = TaskList(taskList.key, ArrayList(itemsHashSet))
      // 5
      taskLists.add(list)
  }

  return taskLists
}
```

Going through this step-by-step:

1. Grab a reference to the default SharedPreferences file. This time, you don't request a `SharedPreferences.Editor` object since you only need to read from the file, not write to it.

2. Call `sharedPreferences.all` to get the contents of your SharedPreferences file as a `Map`.

> **Note**: A `Map` is a collection that holds pairs of objects (keys and values) and supports efficiently retrieving the value corresponding to each key. Map keys are unique; a map holds only one value for each key.

3. Create an empty `ArrayList` of type `TaskList`. You'll use this to store the lists you retrieve from SharedPreferences.

4. Iterate over all the items in the `Map` you received from SharedPreferences using a `for` loop. In each iteration, take the value of the object and attempt to cast it to a `HashSet<String>`. Recall from the `SaveList` earlier that you could not store a `TaskList` object directly as a string, so we had converted that object into a **HashSet**. You perform the reverse to retrieve an object back. Then recreate the list object by passing the `key` of the `sharedPreference` object as the name of the `TaskList` and then convert the `HashSet` into an `ArrayList` of tasks back to a structured object.

5. Finally, add your newly reconstructed list into the empty `ArrayList` you created earlier.

Once you've iterated over the entire set of items you retrieved from SharedPreferences, return the contents of `taskLists` to the caller of your method.

Hooking up the Activity

In the previous section, you created `ListDataManager` to read and write the lists you create. Time to put that to use. Open up `MainActivity.kt` and add the following line to the top of the class:

```
val listDataManager: ListDataManager = ListDataManager(this)
```

This will create a new `ListDataManager` as soon as your Activity is created. Next, update the positive button's `onClickListener` in `showCreateListDialog` as follows:

```
builder.setPositiveButton(positiveButtonTitle, { dialog, i ->
  val list = TaskList(listTitleEditText.text.toString())
  listDataManager.saveList(list)

  val recyclerAdapter = listsRecyclerView.adapter as
ListSelectionRecyclerViewAdapter
  recyclerAdapter.addList(list)

  dialog.dismiss()
})
```

You take the name of the list and create an empty `TaskList` to save to SharedPreferences. You then get the adapter of the RecyclerView and cast it as the custom adapter **ListSelectionRecyclerViewAdapter** that you created earlier.

Pass the newly created `TaskList` into the adapter using `addList` so it knows it has something to show. Don't worry about the **Unresolved reference** error on `addList` since you'll create this method shortly.

That's the background work done for this feature. Now you need to call these methods from somewhere. Back on the `onCreate:` method, replace the setup code for your RecyclerView starting with:

```
// 1
val lists = listDataManager.readLists()
listsRecyclerView = findViewById<RecyclerView>(R.id.lists_recyclerview)
listsRecyclerView.layoutManager = LinearLayoutManager(this)

// 2
listsRecyclerView.adapter = ListSelectionRecyclerViewAdapter(lists)
```

This is fairly straightforward code:

1. You get a list of TaskLists from your DataManager, ready for use.

2. Remember that static array of list titles you added earlier? Here you update that with some dynamic content, changing the last line in the above code snippet so that

the list of tasks is passed directly into the Adapter. Ignore the **Too many arguments** error since you're shortly going to update `ListSelectionRecyclerViewAdapter` to accept a parameter.

Now that your RecyclerView Adapter has a source of information to display, there's a few changes you need to make to ensure everything can work with your new lists.

Open `ListSelectionRecyclerViewAdapter.kt` and update your Class definition as follows:

```
class ListSelectionRecyclerViewAdapter(val lists : ArrayList<TaskList>) :
RecyclerView.Adapter<ListSelectionViewHolder>() {
```

Now your Class can accept the list you want to pass in.

Find `onBindViewHolder` and update it to use the list to populate the `ViewHolder` instead of the static array of strings. The updated code is:

```
override fun onBindViewHolder(holder: ListSelectionViewHolder?, position:
Int) {
    if (holder != null) {
        holder.listPosition.text = (position + 1).toString()
        holder.listTitle.text = lists.get(position).name
    }
}
```

Modify `getItemCount` method so it gets the size of your list as shown:

```
override fun getItemCount(): Int {
    return lists.size
}
```

Finally, create the `addList` method you call from your Activity to let the adapter know you have a new list to display. Add it to the bottom of your Adapter Class as follows:

```
fun addList(list: TaskList) {
  // 1
  lists.add(list)

  // 2
  notifyDataSetChanged()
}
```

1. Here, you update the `ArrayList` with the new `TaskList`.

2. You then call `notifyDataSetChanged` to inform the adapter that it should query its underlying data and update the `RecyclerView`. In this case, the underlying data (your dataset) is the `ArrayList` passed into the `ListSelectionRecyclerViewAdapter` called `list` of type `TaskList`, and any necessary `ViewHolders` will be created to populate each `View` with the right data for each position.

With that done, you can remove the `listTitles` array at the top of the
`ListSelectionRecyclerViewAdapter` class as you no longer need it.

Run your app and tap the FAB to display the Create dialog and enter any name you
wish:
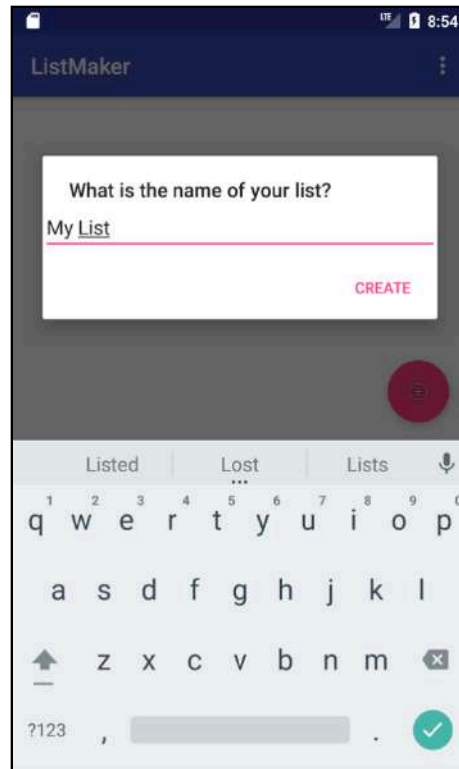
Tap **Create** and you should see your new task list name in the RecyclerView:
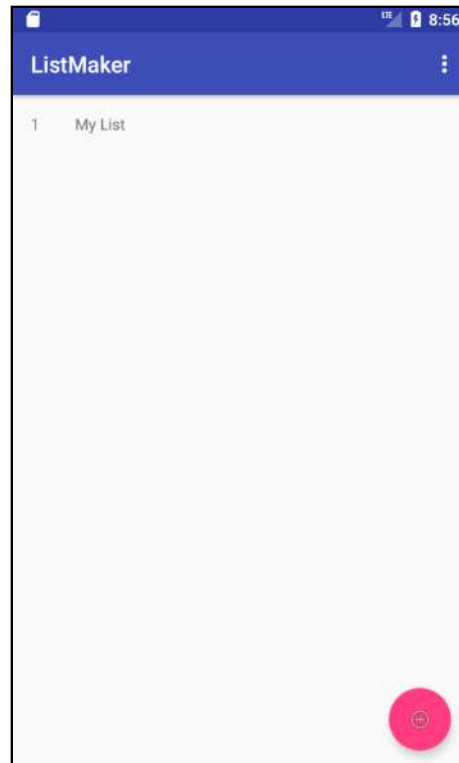
You're not quite done — there's one thing left to verify. Does your list stick around after you stop and restart the app? Click the **Stop** button in Android Studio; it's the big square in the top right corner.

Your device will stop running the app and head back to the home screen. Run your app from Android Studio and, you should see your list return.

Now you can be certain that your app persisted your list to SharedPreferences and loaded it once your app relaunched. Great job!

Where to go from here?

SharedPreferences is a common way to persist values in any Android app, so it's worth keeping in your toolbox. You've learned how to write and read values from SharedPreferences and put that knowledge to good use in ListMaker so your users can save and load their lists.

The next logical step would let the user add items to their lists. And that's exactly what you'll do in the next chapter.

Chapter 9: Communicating Between Activities

By Darryl Bayliss

So far in this book, you've made use of a single **Activity** for your apps. It's a pragmatic choice if your app is simple and doesn't require much screen space.

As your app gets more complex, trying to cram more visual elements into a single screen becomes difficult, and can make your app confusing for users. Keeping an Activity dedicated to a single task removes this problem.

In this chapter, you'll do just that. As you may have realized while building ListMaker, there is no way to add items to the lists you create. This is a perfect task to have in a separate Activity. By the end of the chapter you'll have learned about the following:

1. How to create another Activity.

2. How to communicate between Activities using an Intent.

3. How to pass data between Activities.

Getting started

If you are following along with your own app, open it and keep going with it for this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **ListMaker** app under the **starter** folder. The first time you open the project, Android Studio takes a few minutes to set up your environment and update dependencies.

> **Note**: If you added a few lists in the prior chapter, you will see them inside the app still. If you want to start fresh, delete the app first from your emulator, then keep going with this chapter. All the previous list data will be deleted when you delete the app.

Open up the ListMaker project and run the app. When the app appears on screen, tap the Floating Action Button in the bottom right, and enter the title **My List** in the dialog that comes up.
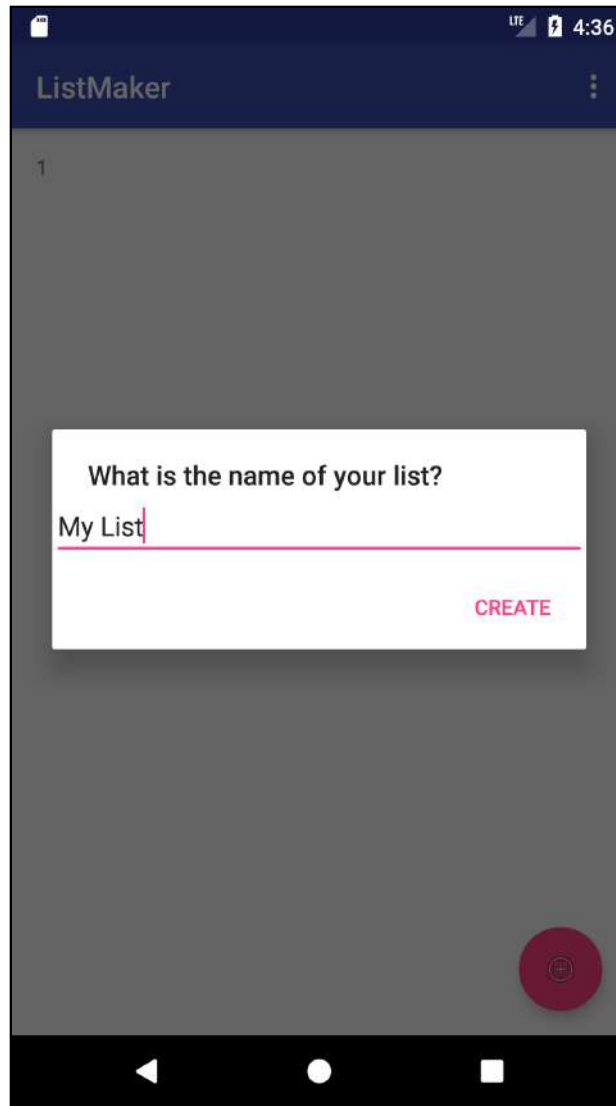
Tap **Create** and your new list name will show up in the RecyclerView as it did before.
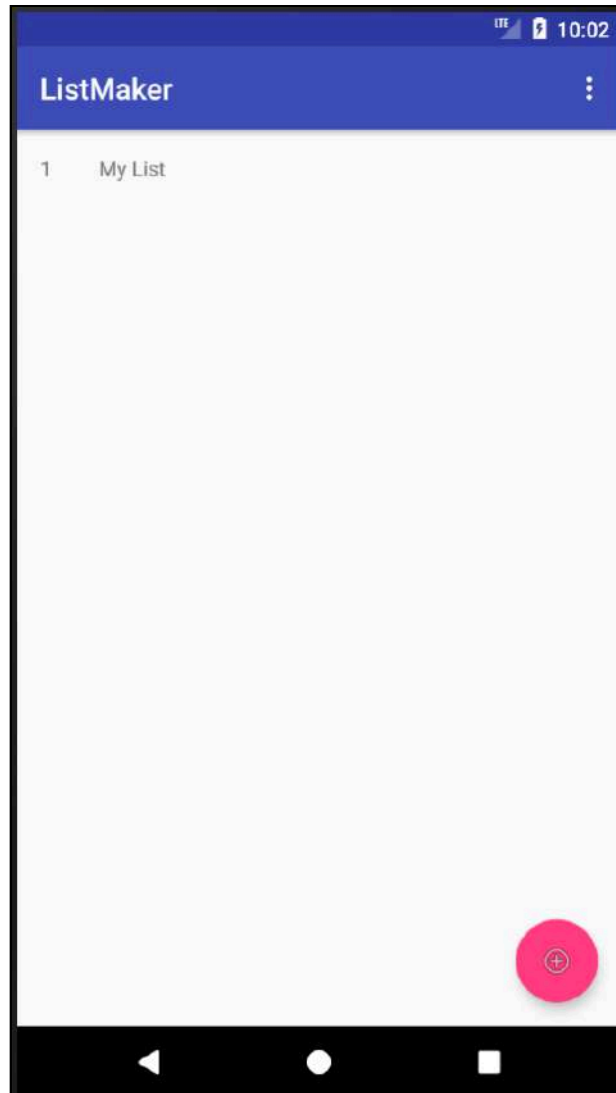
That works, but it's a bit simplistic. In addition, if you tap on the title of your list in the RecyclerView, nothing seems to happen. Right now, that's the end of the story, and it doesn't seem all that useful.

What should be the next logical step in your list making app? Wouldn't it be great to have a more customized screen to enter the name of your list? What about something happening when you tap on the title in the RecyclerView, too?

To do that, you're going to create another Activity. As a good rule of thumb, Activities should focus on a single task. That way, the logic within an Activity stays clean and simple as you build it out. This also benefits your users because navigation between screens becomes more intuitive.

Right-click on the **com.raywenderlich.listmaker** package in the project navigator (highlighted in the screenshot) to the left of Android Studio:
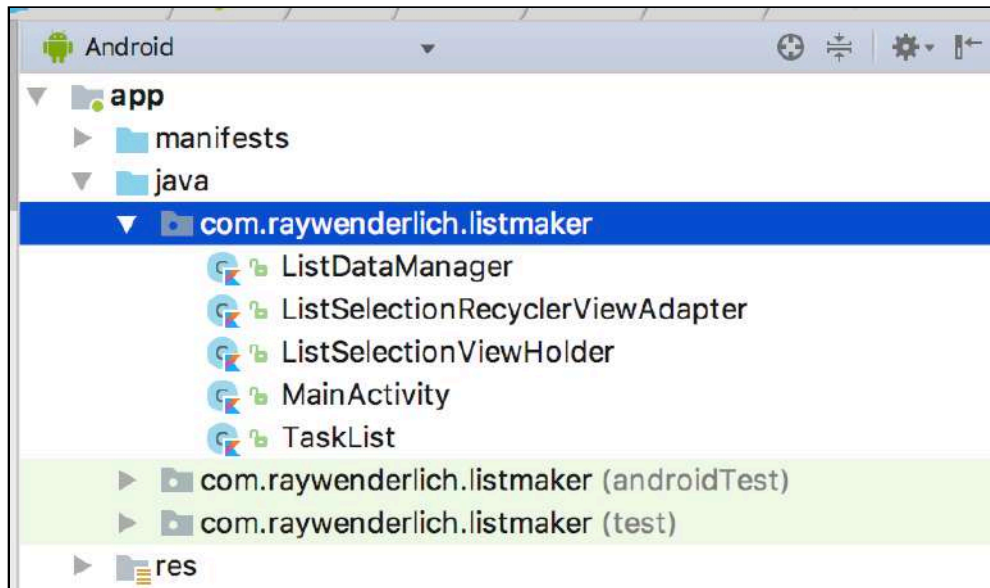
In the floating selection that appears, select **New ‣ Activity ‣ Empty Activity**.
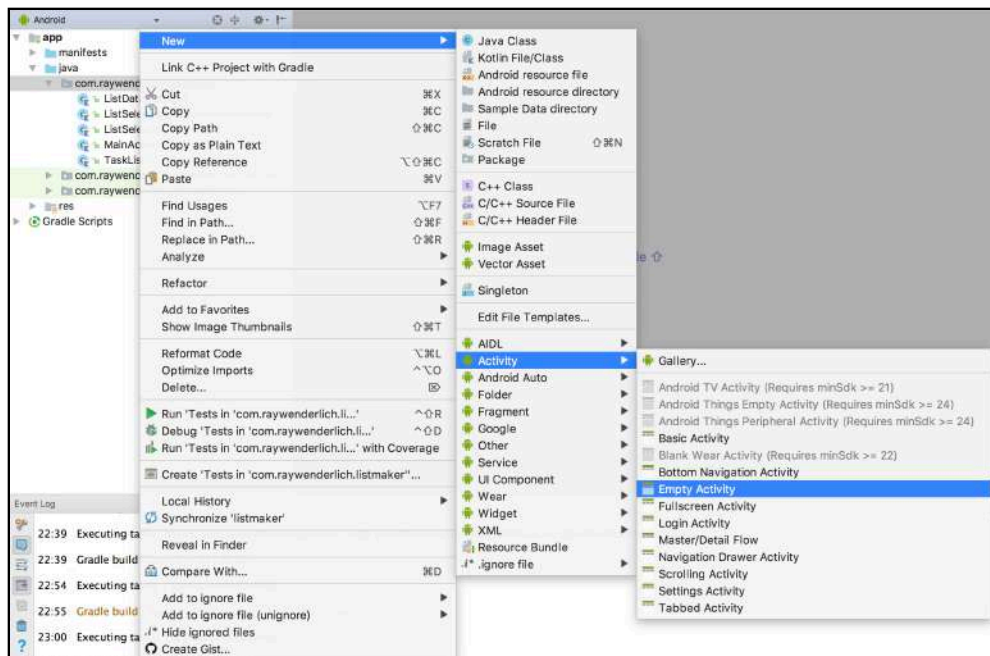
Android Studio will bring up a new window, giving you an opportunity to customize your new Activity before creating it.
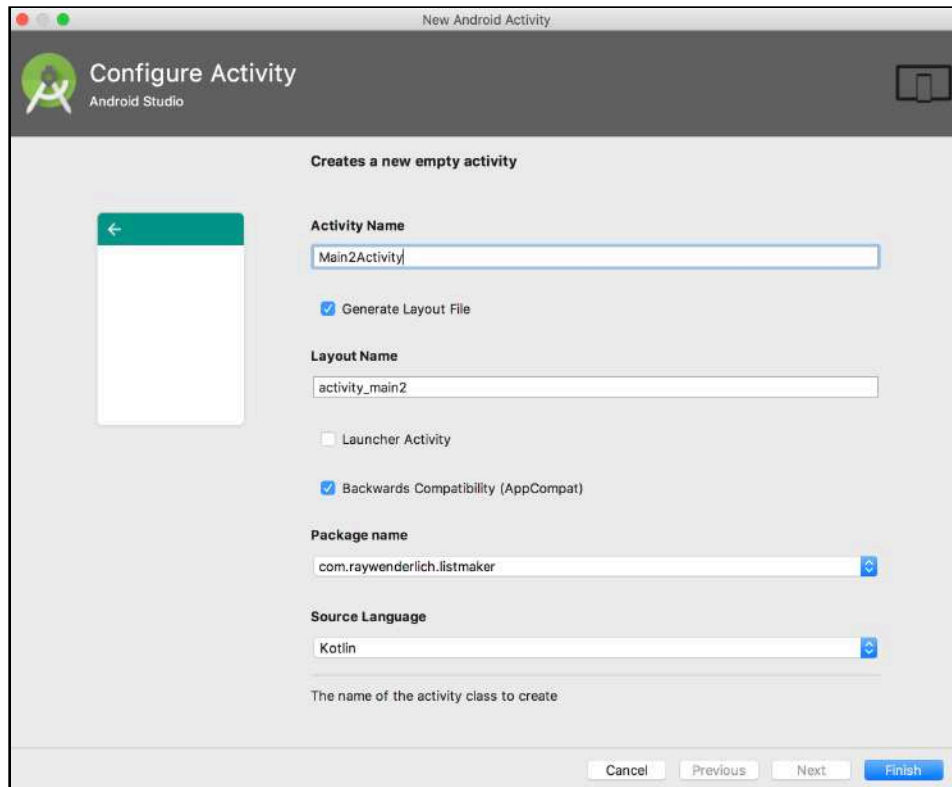
Creating a new Activity

Going through the options one by one:

- **Activity Name** is the name you want to give your Activity. This is used to name the class in Kotlin or Java which is associated with your Activity.

- **Generate Layout File** gives you the option to generate a layout XML file to use in conjunction with your Activity class. This is checked by default, since it's rare that you won't want to create a layout.

- **Launcher Activity** gives you the chance to set your new Activity as the one shown first when your app starts. This is unchecked by default. You'll see how to change the starting Activity later in the chapter.

- **Backwards Compatibility (AppCompat)** ensures that your Activity inherits from `AppCompatActivity`. This ensures backwards compatibility across the many versions of Android. You'll learn more about this topic in Chapter 28, but for now, it's best to leave this checked.

- **Package Name** lets you select the package your Activity class will be created in. Since you only have one package in your project, this will default to **com.raywenderlich.listmaker**.

- **Source Language** lets you choose what programming language the Activity will use. The choices are Java and Kotlin. In this project the default is Kotlin, since you want to be cutting-edge!

> **Note**: If you don't see **Source Language** as a choice, try scrolling inside the area with all the choices. Depending on your screen size, the value for language might not be visible unless you scroll.

Most of the choices here are fine at their defaults. The only thing to change is the Activity Name: Give it the name **ListDetailActivity**.

As you change the name of the Activity, the Layout name also changes to something similar: `activity_list_detail`. Android Studio tries to keep your filenames related to follow Android platform conventions and to make it easier to find your files later.

It's time to create the Activity. Click **Finish** in the bottom right of the window and Android Studio will create your new Activity.

```kotlin
package com.raywenderlich.listmaker

import ...

class ListDetailActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_list_detail)
    }
}
```

Android Studio has even hooked up your layout as well, so you don't have to do this yourself. How nice! But there's still one more piece of the puzzle you need to know about: the app manifest.

The App Manifest

Every Android App has an **app manifest** file. It's important, since it tells Android everything it needs to know about your app.

Android is quite strict about its requirements for a manifest. The file name must be **AndroidManifest.xml** and must be located in just the right spot in your project file hierarchy. Without this file, Android will simply refuse to run your app.

Let's open it up. On the left side of Android Studio in the project navigator, navigate to **app ▸ manifests ▸ AndroidManifest.xml**.

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.raywenderlich.listmaker">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="ListMaker"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".MainActivity"
            android:label="ListMaker"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".ListDetailActivity"></activity>
    </application>

</manifest>
```

> **NOTE**: The **manifests** folder in the sidebar is a virtual folder generated by Android Studio's Android project view, and is not directly related to anything in the file system. The actual file is kept at the root of your app's **main** folder under **app/src**.

As you can see, this is an XML based file containing various tags. The tags in this file are `manifest`, `application` and `activity`, although there are plenty more you will use in

chapters to come.

The `manifest` tag is the root element of the App Manifest. All other tags must be declared within it, and the package where your code sits must also be declared within the tag. This is a security measure to ensure that only your package is associated with this app.

The `application` tag contains app-specific information for the Android system, such as the icon to use for the app, the name of the app, and what theme style it uses. This information tells Android exactly how to present the app on the home screen and how to represent it in other areas such as Settings.

Perhaps the most interesting pieces of information available within the application tag are the `activity` tags. Every Activity within an app should have a corresponding tag within the manifest. This is to ensure that your app only runs Activities from within your app, not any that may have came from elsewhere.

There's a `.MainActivity` declared in there, with another tag — `intent-filter` — inside this declaration. This tells Android that `MainActivity` is the activity to start when the app launches.

This happens thanks to the inclusion of the `action` and `category` tags inside `intent-filter`. You don't need to be concerned about the details behind these tags at the moment; you'll learn more about intents later in this chapter. What you need to know is that the `intent-filter` will be used to set your main activity as the startup activity.

You will also see `.ListDetailActivity`, which is the activity you created in the first part of this chapter. When you create a new project or use the new activity wizard, Android Studio does the hard work of updating the Manifest so you don't have to.

If you prefer, you can edit the manifest manually. You'll actually do this yourself in future chapters. However, it's best if you let Android Studio do the hard work to reduce the chance of human error.

Intents

Now that you have your two Activities, it's time to give your app the ability to navigate between the two. In your MainActivity, you have two main points of entry for your new activity: the first, when a user taps the name of the list in the RecyclerView; the second, when a user enters the name of a new list and taps Create.

You will navigate between the two Activities by using an **Intent**. An intent is an object

that encapsulates some work or action that your app will perform at some point in the future.

The Android OS relies heavily on intents as its primary form of communication to know what to do, so it's best that you use them for your app communication as well. Intents are extremely flexible and can perform a wide range of tasks such as communicating with other apps, providing data to processes or starting up another screen.

Even your app will be launched by the Android system via an intent. Remember `intent-filter` in the app manifest? That filter allows an activity to be picky about what intents it handles. In the case of your MainActivity, it only wants to handle intents that attempt to launch it.
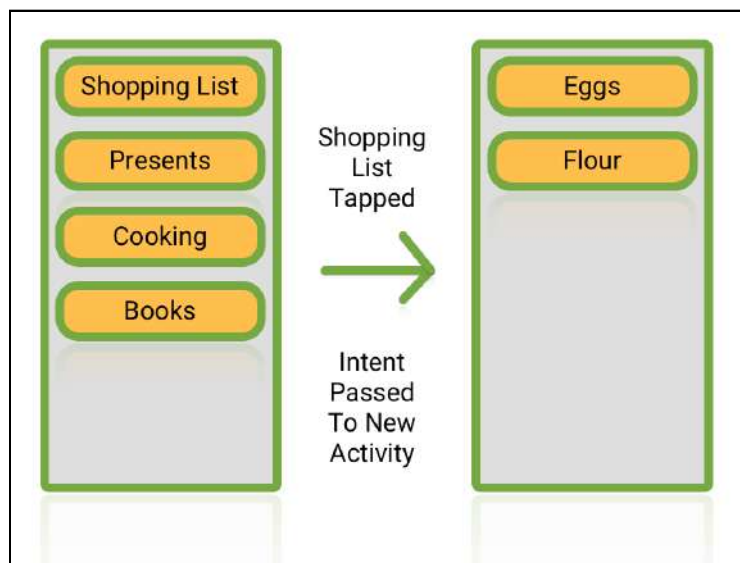
An Intent is created to show another Activity on screen

Let's get to it. In **MainActivity.kt**, add the following method to the bottom of the file:

```
private fun showListDetail(list: TaskList) {
  // 1
  val listDetailIntent = Intent(this, ListDetailActivity::class.java)
  // 2
  listDetailIntent.putExtra(INTENT_LIST_KEY, list)
  // 3
  startActivity(listDetailIntent)
}
```

This is a small but important method in your app: it creates an **Intent** like we discussed above.

This code does the following:

1. You create an intent and pass in the current Activity and Class of the Activity you

want to show on screen. Think of this as saying you are currently on *this* screen, now you want to move to *that* screen.

2. Next you add something called an **Extra**. Extras are keys with associated values that you can provide to intents to give more information to the receiver about the action to be done. In your case, you want to display a list. This is why the method expects a `list` variable to be passed in, which you use as a parameter in the `putExtra` method call.

    You also pass in a constant called `INTENT_LIST_KEY`. This is a string that the receiver of the intent uses as a key to reference the list. You'll add this constant shortly (so it's ok to ignore the **Unresolved reference** error for now).

3. The final line is simply a method call to inform your current activity to start another activity, making use of all the information provided within the intent.

Intents and Parcels

You've already set up the presentation of your new screen. However there is a small problem: `TaskLists` can't be passed through intents, since Android doesn't understand how to handle that type of object. Fortunately, there is a way around that.

Open `TaskList.kt` and change the class declaration so it implements the `Parcelable` interface:

```
class TaskList constructor(val name: String, val tasks: ArrayList<String>
= ArrayList()) : Parcelable
```

`Parcelable` lets you break your object down into types the intent system is already familiar with: Strings, ints, floats, Booleans, and other objects which conform to `Parcelable`. You then can shove all that information into a `Parcel`.

To help transfer data, intents use a `Bundle` object which can contain `Parcelable` objects. This is exactly what you're using to pass your list as an **Extra** in the intent you setup earlier.

Next, you need to implement some required methods so your object can be parceled up. To do that, add the following constructor and methods inside the braces of the `TaskList` class:

```
//1
constructor(source: Parcel) : this(
  source.readString(),
  source.createStringArrayList()
```

```kotlin
  )

  override fun describeContents() = 0

  //2
  override fun writeToParcel(dest: Parcel, flags: Int) {
    dest.writeString(name)
    dest.writeStringList(tasks)
  }

  // 3
  companion object CREATOR: Parcelable.Creator<TaskList> {
      // 4
      override fun createFromParcel(source: Parcel): TaskList =
  TaskList(source)
      override fun newArray(size: Int): Array<TaskList?> =
  arrayOfNulls(size)
    }
  }
```

There's a lot of boilerplate code here, but for now you only need to know about the four most important parts:

1. **Reading from a Parcel**: Here you're adding a second constructor (as opposed to the primary constructor in the class declaration) so that a `TaskList` object can be created from a passed-in `Parcel`.

    The constructor grabs the values from the `Parcel` for the title (by calling `readString` on the `Parcel`) and the list of tasks (by calling `createStringArrayList` on the `Parcel`), then passes them into the primary constructor using `this()`.

2. **Writing to a Parcel**: This method is called when a `Parcel` needs to be created from the `TaskList` object. The parcel being created is handed into this functions, and you fill it in with the appropriate contents using the assorted `write...` functions.

3. **Fulfilling `static` interface requirements**: The `Parcelable` protocol requires you to create a `public static Parcelable.Creator<T> CREATOR` field and override a couple methods in it in Java. However, `static` methods don't exist in Kotlin. Instead, you create a `companion` object meeting the same requirements, and override the appropriate functions within that object.

4. **Actually calling your constructor**: In the `CREATOR` companion object, you override the interface function `createFromParcel`, and pass the parcel you get from this function along to the second constructor you just created, giving back a nice new `TaskList` with all the data from the `Parcel`.

> **Note**: For more information about the `Parcelable` interface, head on over to
> https://developer.android.com/reference/android/os/Parcelable.html.

> There is also an experimental feature in Kotlin initially released with Kotlin 1.1.4
> to allow you to simply annotate your classes with `@Parcelize` instead of writing all
> this boilerplate. It'll certainly be nice when that's out of experimental mode!

Bringing everything together

Now that your `TaskList` is able to be passed around on Android, it's time to add a few
things to make sure everything works as intended. The first is the `INTENT_LIST_KEY`
constant you're using to place the list in the Bundle.

Add the following under the class declaration at the top of **MainActivity.kt**:

```
companion object {
  val INTENT_LIST_KEY = "list"
}
```

This key will be used by your intent to refer to a list whenever it needs to pass one to
your new Activity.

Now you need to hook up `showListDetail` in a few ways. You'll start with the list
creation.

Inside `showCreateListDialog`, head to the bottom of the `setPositiveButton` closure
code. Add a call to `showListDetail` after the dialog is dismissed, so it looks like the
following:

```
builder.setPositiveButton(positiveButtonTitle, { dialog, i ->

  val list = TaskList(listTitleEditText.text.toString())
  listDataManager.saveList(list)

  val recyclerAdapter = listsRecyclerView.adapter as
ListSelectionRecyclerViewAdapter
  recyclerAdapter.addList(list)

  dialog.dismiss()
  showListDetail(list)
})
```

Now when you create a new list, your app will pass that list to your new Activity. Easy!
You also want the same thing to happen if a user taps on an existing list in the

RecyclerView.

Open up **ListSelectionRecyclerViewAdapter.kt** and add the following new interface above `onCreateViewHolder`:

```
interface ListSelectionRecyclerViewClickListener {
  fun listItemClicked(list: TaskList)
}
```

In the class declaration above that, update the constructor to allow passing in a click listener:

```
class ListSelectionRecyclerViewAdapter(val lists: ArrayList<TaskList>,
val clickListener: ListSelectionRecyclerViewClickListener) :
RecyclerView.Adapter<ListSelectionViewHolder>()
```

Finally, edit `onBindViewHolder` to add an `onClickListener` to the View of `itemHolder` so it looks like this:

```
override fun onBindViewHolder(holder: ListSelectionViewHolder?, position:
Int) {

  if (holder != null) {
    holder.listPosition.text = (position + 1).toString()
    holder.listTitle.text = lists.get(position).name
    holder.itemView.setOnClickListener({
      clickListener.listItemClicked(lists.get(position))
    })
  }
}
```

Here you've created an object to listen for any clicks that occur on the rows of your RecyclerView. When a click event happens, this code tells the listener which list was tapped.

Head back to **MainActivity.kt** and update the class declaration to state that it conforms to the `ListSelectionRecyclerViewClickListener` interface you just created. The new class declaration should look like this:

```
class MainActivity : AppCompatActivity(),
ListSelectionRecyclerViewAdapter.ListSelectionRecyclerViewClickListener
```

Then at the bottom of the class, add the following:

```
override fun listItemClicked(list: TaskList) {
  showListDetail(list)
}
```

This `listItemClicked` method calls `showListDetail` and passes in the selected list (the list the user clicked/tapped on).

Next, update the initialization of `ListSelectionRecyclerViewAdapter` in `onCreate` to pass in your Activity as the listener:

```
listsRecyclerView.adapter = ListSelectionRecyclerViewAdapter(lists, this)
```

Now MainActivity is ready to send your list! Before you can see it in action though, you need to do one last thing: Handle the intent in your `ListDetailActivity` class.

Recall that the intent is passing a parcel with the list that the detail activity is operating on. So, we need a local variable to hold the list.

Open up **ListDetailActivity.kt** and add the following variable to the top of the class above `onCreate`:

```
lateinit var list: TaskList
```

Next, in `onCreate`, you need to retrieve the list you passed in as an **Extra**. To do that, change `onCreate` as follows:

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_list_detail)
  // 1
  list = intent.getParcelableExtra(MainActivity.INTENT_LIST_KEY)
  // 2
  title = list.name
}
```

In this code:

1. You use the key assigned to your list in **MainActivity.kt** to reference the list in your intent and assign it to your `list` variable.

2. You assign the title of your Activity to the name of your list to let the user know what list they're adding.

Time to see your hard work in action!

Click the **Run App** button in the top right of Android Studio and select your favorite device or emulator. Once the app is running, create a new list. Tap the Create button, and behold! Your new activity appears on screen — with the new list.
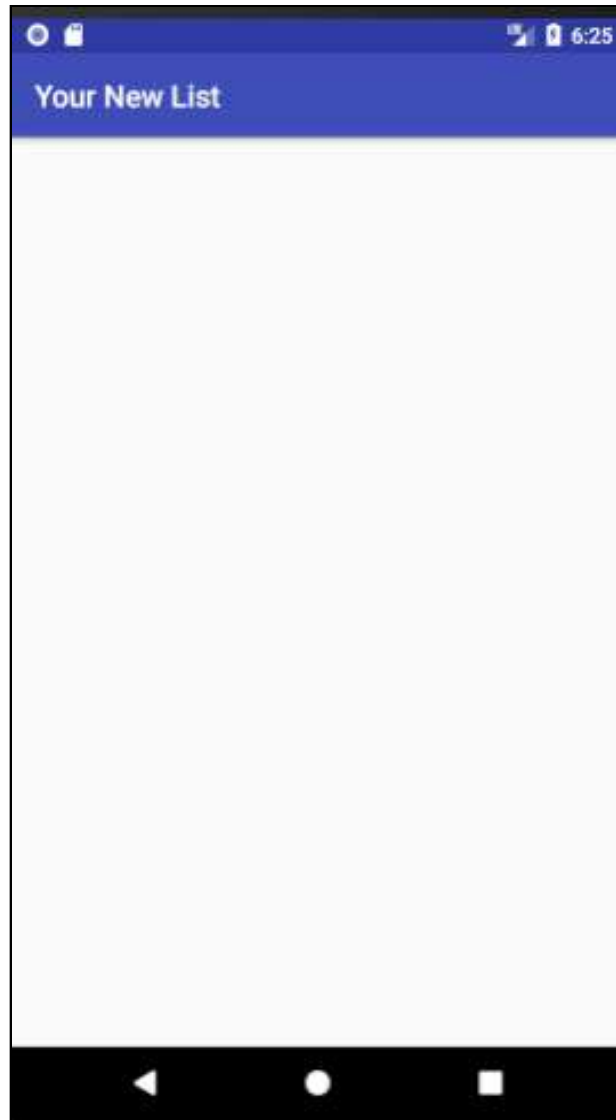
Android took the intent you created in **MainActivity.kt** and passed it to
ListDetailActivity.kt so it could use the list in the new Activity.

Where to go from here?

Intents are another common pattern you'll see in all Android apps, since they're used
for all kinds of purposes beyond starting Activities. Learning the abilities of these
objects and how to use them in your apps is another powerful tool to have in your
Android toolbox.

Chapter 10: Completing the Detail View

By Darryl Bayliss

In the last chapter, you set up a new Activity to display the contents of a list. At the moment, that Activity looks extremely empty.

In this chapter, you're going to build up that Activity, use familiar components such as a **RecyclerView** to display the list, and add a **FloatingActionButton** to add tasks to the list.

You'll also learn how to communicate back to the previous Activity using an **Intent**, similar to what you did in the last chapter when you passed a list to your new Activity.

Getting started

If you are following along with your own app, open it and keep going with it for this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **ListMaker** app under the **starter** folder. The first time you open the project, Android Studio takes a few minutes to set up your environment and update dependencies.

Open **ListDetailActivity.kt** and take a moment to see what is there. At the moment you pass in a list from **MainActivity.kt** via an Intent and set the title of the Activity to the name of the list.

This Activity should do a little more than that to make ListMaker a useful app! It needs to let a user view all the items in the list, and add items to it. To do that, you're going to rely on the faithful RecyclerView to show all the items.

Open up **res\layout\activity_list_detail.xml** from the **layout** folder, making sure the Design tab is selected in the layout window.
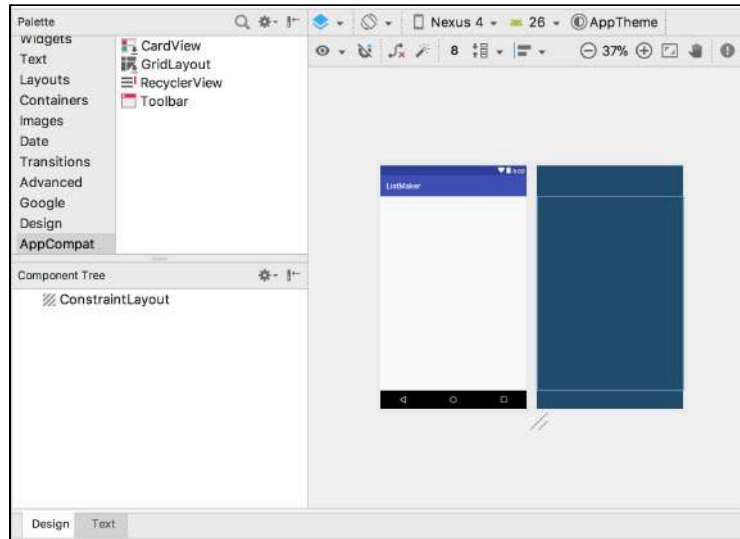
In the Palette window, select the **AppCompat** option in the left-hand list. You will be able to see the RecyclerView available for selection in the right-hand list.
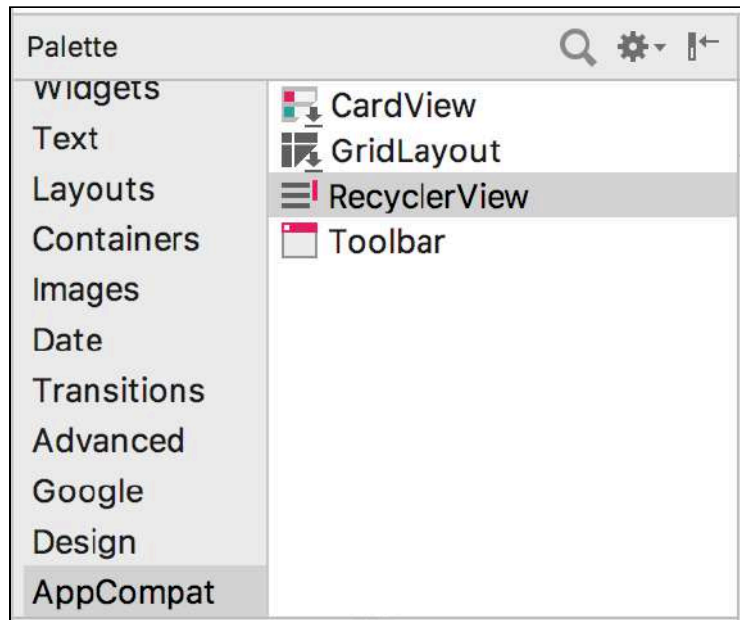
Click and drag the RecyclerView button to the whitespace in the layout shown on the right of the Layout Window.
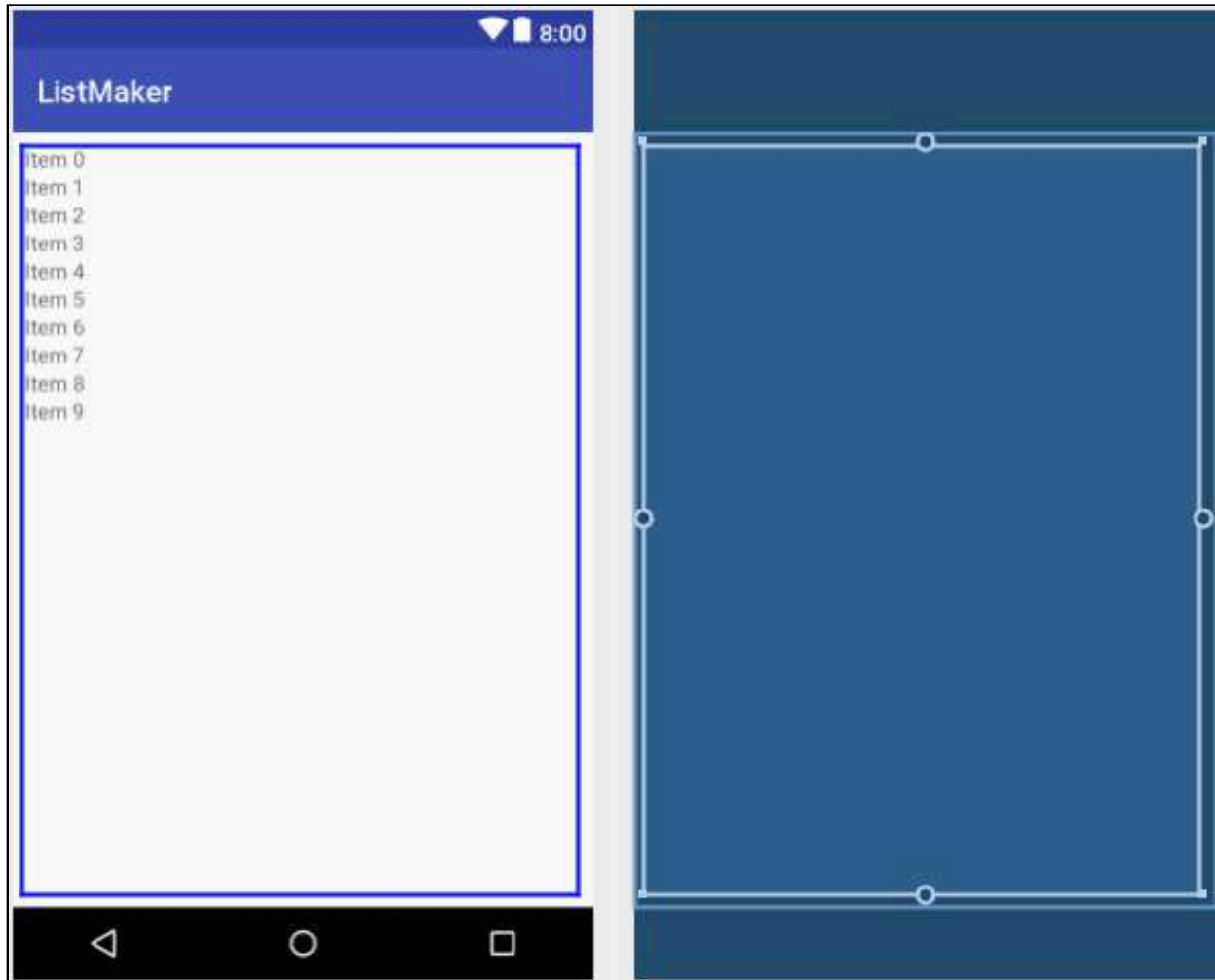
With your RecyclerView set up, it's time to give it an ID and some dimensions. To the right, in the **Attributes** window, change the ID of the RecyclerView to **list_items_reyclerview**:

Then add constraints to the RecyclerView by clicking the four plus buttons around the square underneath the ID textfield. Change the margins for each constraint to 0:
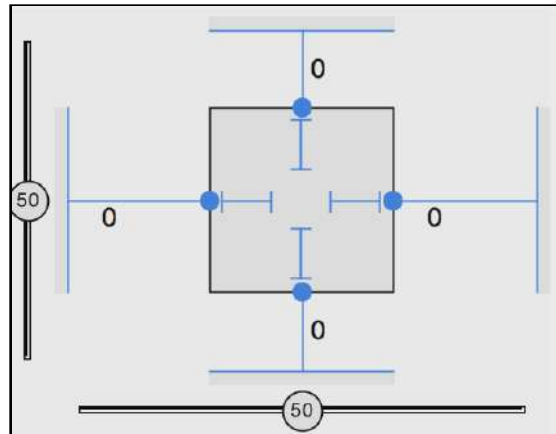
Underneath, update the layout_width and layout_height dropdowns to **match_constraints**.

This will ensure your RecyclerView adheres to the constraints set on it and take up the entire screen.

With your RecyclerView set up, it's time to use it in your code.

Coding the RecyclerView

Open up **ListDetailActivity.kt**. At the top of the class, add the following variable to hold a reference to your RecyclerView:

```
lateinit var listItemsRecyclerView : RecyclerView
```

If offered a choice, choose the `v7.appcompat` version of the RecyclerView component.

At the bottom of `onCreate(savedInstanceState: Bundle?)` add the following code to reference your RecyclerView:

```
// 1
listItemsRecyclerView =
    findViewById<RecyclerView>(R.id.list_items_reyclerview)
// 2
listItemsRecyclerView.adapter = ListItemsRecyclerViewAdapter(list)
```

```
// 3
listItemsRecyclerView.layoutManager = LinearLayoutManager(this)
```

This code:

1. Finds the RecyclerView in the Activity layout and assigns it to our local variable.

2. Assigns the RecyclerView an Adapter, passing the list in. It needs to know about the list so it can tell the RecyclerView what tasks to show. You'll create the adapter shortly so ignore the **Unresolved reference** for now.

3. Assigns the RecyclerView a layout manager to handle the presentation.

With the RecyclerView all set up, let's tackle that Adapter. In the Project Navigator, right click on the `com.raywenderlich.listmaker` package.

In the popup that appears, navigate to **New ▸ Kotlin File ▸ Class**.
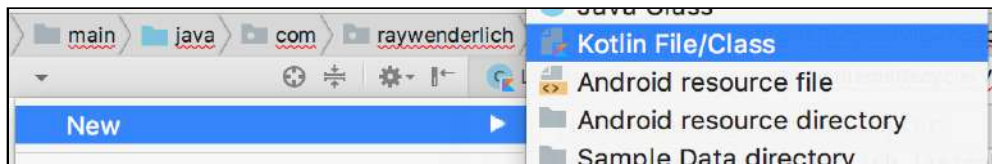
In the window that appears, name your new Class **ListItemsRecyclerViewAdapter**, ensure the **Kind** dropdown is set to **Class** and click **OK**.
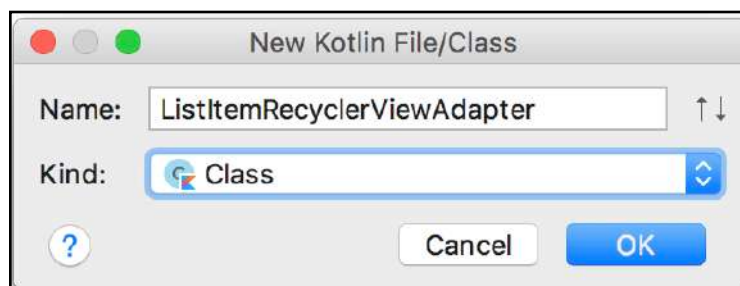
Android Studio will create your new Kotlin Class and present it to you once it's finished.

You'll need to make a few adjustments to this Class. In **ListDetailActivity.kt**, you pass in a list to your RecyclerView Adapter. To be able to use that list, your adapter needs a constructor that accepts a `TaskList`.

You also need to make your Class implement the `RecyclerView.Adapter<ViewHolder>` Interface, so your adapter can create ViewHolders for your RecyclerView and reuse them as necessary. Finally, you need to create a custom `ViewHolder` that you can use to show the tasks in your list.

Let's do this now. First, update the class definition so it has a default constructor that accepts a `TaskList`, and have it conform to `RecyclerView.Adapter<ListItemViewHolder>`:

```
class ListItemsRecyclerViewAdapter(var list: TaskList) :
RecyclerView.Adapter<ListItemViewHolder>()
```

You'll create the `ListItemViewHolder` shortly, so ignore the **Unresolved reference** here too.

You'll need to create another Kotlin Class, just as you did earlier, for the ViewHolder your Adapter will use.

Set the name of the file to **ListItemViewHolder**, making sure to set the **Kind** dropdown to **Class**.
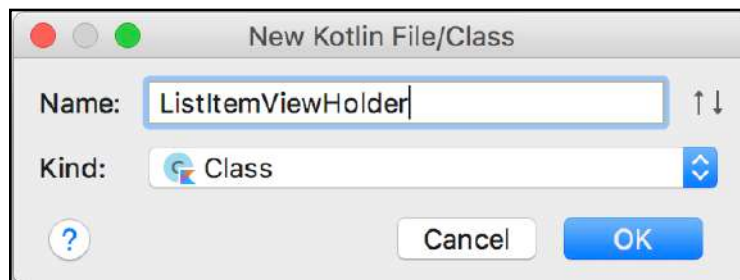
Once Android Studio has created the Class, update the Class definition as follows:

```
class ListItemViewHolder(itemView: View?) :
RecyclerView.ViewHolder(itemView)
```

This code sets up the `ListItemViewHolder` constructor to pass in a View you can use to reference the ViewHolders widgets. It also makes the Class implement the `RecyclerView.ViewHolder(itemView)` interface, passing in the view you want to use.

With the bare bones of your Adapter and ViewHolder set up, you now need to instruct your Adapter how to work with your list of tasks.

Adapting the Adapter

Head back to **ListItemsRecyclerViewAdapter.kt**. You're now ready to begin hooking together the adapter. As you did previously with the RecyclerView Adapter in **MainActivity.kt**, your adapter must implement methods to ensure your RecyclerView knows how to present each task in your list.

There is a way to let Android Studio do most of the work for you. Click on the class name (the part where the red squiggly line is) and press **Alt + Return**:
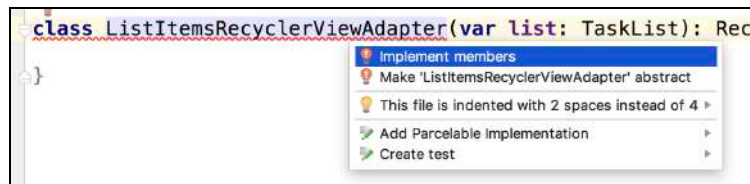
In the popup that appears, you'll see the first option highlighted is **Implement Members**. Press **Return** again and you'll be presented with a window.
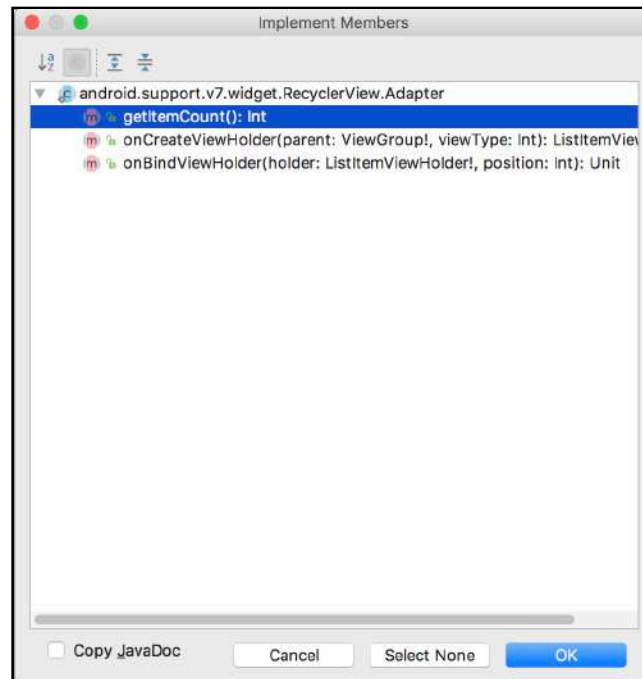
This window shows all the methods you implement to conform to the **RecyclerView.Adapter** Interface, which is the Interface your Class has implemented. You need to implement all these methods. To do this, hold down **Shift** and click on the bottom most method.

All the methods will be highlighted in blue. This means you've selected all the methods you want Android Studio to implement. Exactly what you need! To finish, click **OK**.
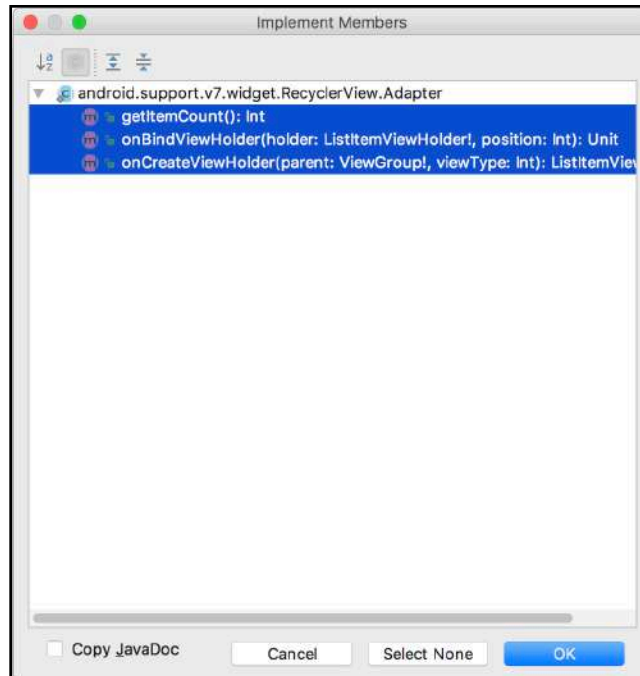
With that, Android Studio will automatically generate the chosen methods for you. It's up to you to write the logic behind each method.

Begin with `getItemCount`. This is the method that tells your RecyclerView how many items it needs to display. You want it to show all the tasks in your list. To do that, update the method so it returns the amount of tasks it contains:

```
override fun getItemCount(): Int {
  return list.tasks.size
}
```

That was easy! Let's move onto something a little more difficult: creating your ViewHolder in `onCreateViewHolder()`. Here, you need to let your RecyclerView know what Layout to use as the ViewHolder.

Since you haven't created the Layout yet, you'll add the code and then look at creating the Layout. Update `onCreateViewHolder()` so it creates a view from the layout using a LayoutInflater, which it passes into your ViewHolder, ready for use:

```kotlin
override fun onCreateViewHolder(parent: ViewGroup?, viewType: Int):
ListItemViewHolder {

  val view = LayoutInflater.from(parent?.context)
            .inflate(R.layout.task_view_holder, parent, false)
  return ListItemViewHolder(view)
}
```

Don't worry about the **Unresolved reference** for `task_view_holder` since we're going to create that next.

With your ViewHolder ready to be used, all you need now is a Layout for the LayoutInflater to inflate. To create the Layout, right click on the **layout** folder inside the **res** folder in the project navigator to the left of Android Studio, move your cursor to **new** and click on **Layout Resource File**.

In the window that appears, enter the file name as **task_view_holder**. Change the Root Element underneath from `android.support.constraint.ConstraintLayout` to `LinearLayout`:
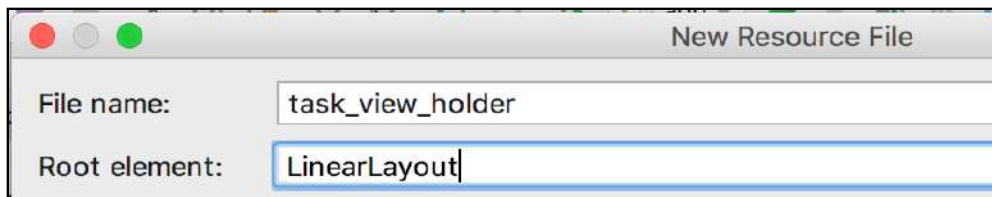
This will make your layout use a **LinearLayout**, which allows you to stack views in a vertical or horizontal direction. For simple views like this **task_view_holder**, it's a lot easier to use than a **ConstraintLayout**.

Finally, click **OK** at the bottom to let Android Studio create your new layout.
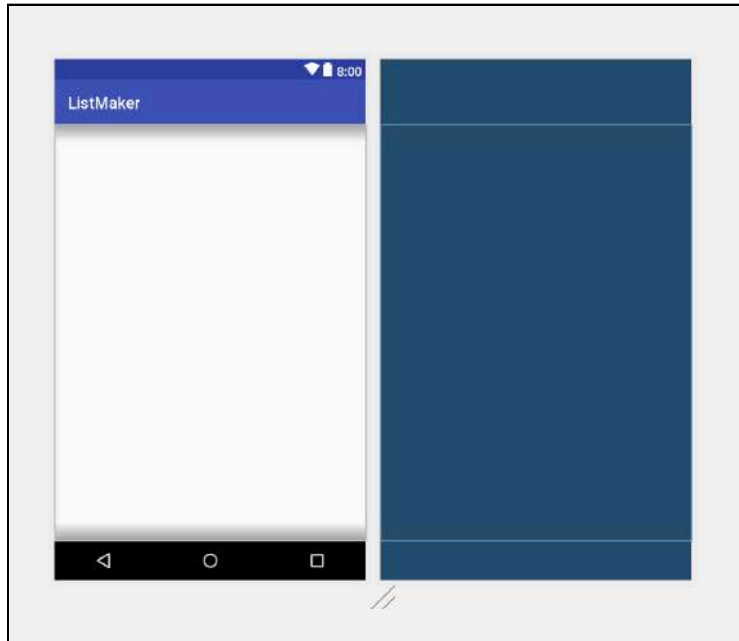
Before you continue, you want to make sure the **LinearLayout** is only as tall as the content within it. Otherwise, every row would be the size of its entire parent - in this case, the **RecyclerView**, which takes up the whole screen!

To do this, update the **LinearLayout**'s **layout_height** to be **wrap_content**:

Now, your layout will only be as big as whatever is inside of it - and you will probably see its height shrink down to nothing as there's nothing in it yet.

To fix this, you need to add the Widgets you want to use with your ViewHolder. In this, case you only need a **TextView** to hold a task in your list.

In the Palette window, click **Text** and then drag a **TextView** into your layout.
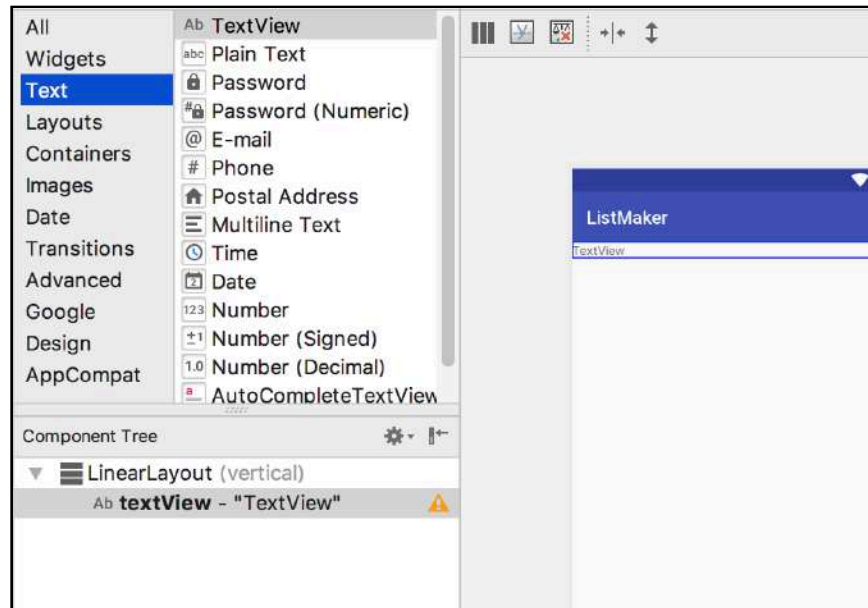
In the **Attributes** window to the right of Android Studio, change the ID of the TextView to **textview_task**, and set the **layout_width** and **layout_height** to **wrap_content**.

There's one final tweak needed here, some margin spacings. To do that, you need to get to the larger list of attributes for the TextView. Click the **View all attributes** text at the bottom of the attributes window.

The attributes will slide across to reveal the entire list of attributes available for tweaking. Click the arrow next to the **Layout_Margin**, and in the left and top textfields, enter **8dp**.

With your Layout ready, it's time to get back to coding some Kotlin!

Visualizing the ViewHolder

Now that you have a layout for your ViewHolder, you need to reference the TextView in the layout in your code. Open **ListItemViewHolder.kt** and add the following line in between the Class brackets:

```
val taskTextView = itemView?.findViewById<TextView>(R.id.textview_task)
    as TextView
```

Now when the ViewHolder is instantiated, it knows exactly how to reference the TextView. Now you need to hook up the data to your ViewHolder.

Open **ListItemsRecyclerViewAdapter.kt**. In onBindViewHolder(), update it as follows:

```
override fun onBindViewHolder(holder: ListItemViewHolder?, position: Int)
{
  if (holder != null) {
      holder.taskTextView.text = list.tasks[position]
  }
}
```

This code grabs a specific task from the list, depending on the position of the row the RecyclerView is intending to show. As before, we wrap this in a null check just in case there was some issue when the class was created.

Great job! You've just finished hooking up your Adapter, and your RecyclerView will now run as expected.

Run the app on the emulator or a device and select one of the lists in the main Activity. It runs but you'll see... not much:

The reason for this is there's no way to add tasks to your lists! That's something that needs to be fixed.

Open **res\layout\activity_list_detail.xml**, making sure the **design** tab is selected. In the list next to the design list, next to the category selector, grab a **FloatingActionButton**.
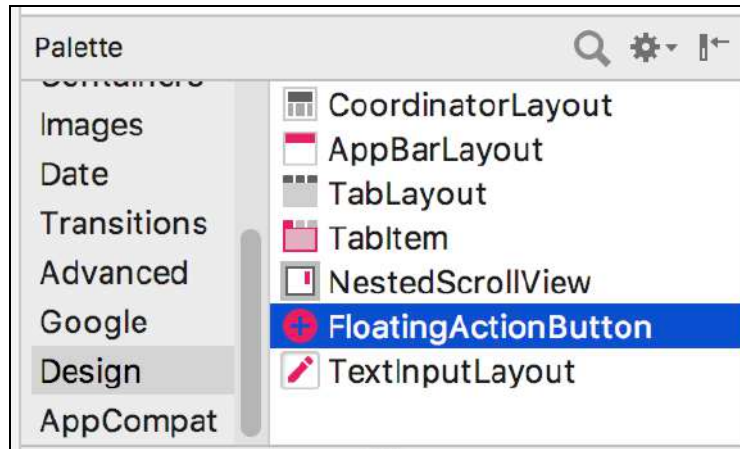
Drag the FAB into the layout. Be careful to drag it into the **ConstraintLayout** and not the **RecyclerView**, otherwise it won't show up. A window will appear, asking you to select a resource for the action button:
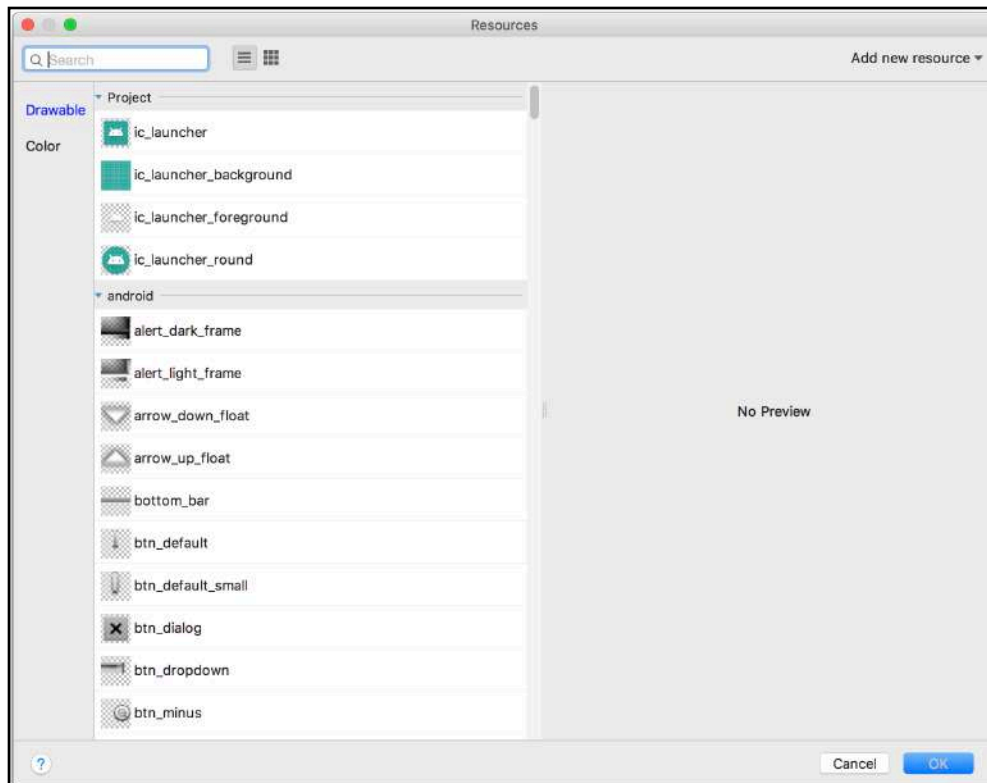
In the search bar, type **add** to filter the list of resources available. Click the familiar-looking **ic_menu_add** resource and click **OK** at the bottom.
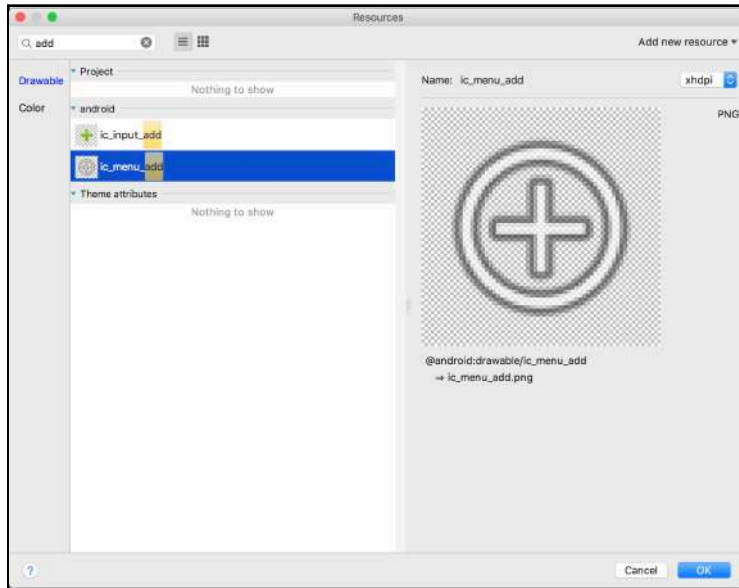
The button will now appear in the layout ready for you to position.

Click on the button. In the blueprint view, drag the right button constraint to the right edge of the layout. Then drag the bottom button constraint to the bottom edge of the layout, so it's positioned at the bottom right.
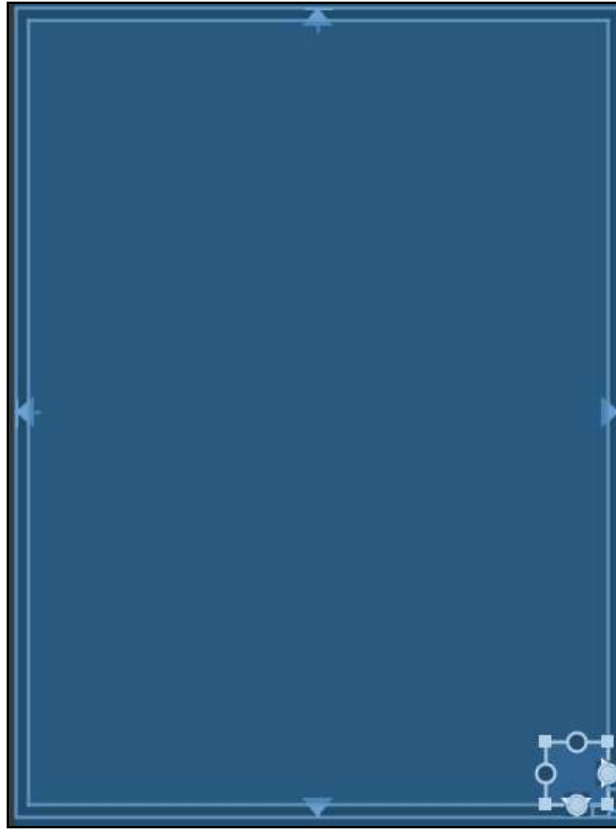
In the attributes window, change the ID to **add_task_button**. In the constraints view underneath the ID textfield, make sure the bottom and right constraints are added. If they're not, add them by using the plus button. Set the margin of the bottom and right constraints to **16**.
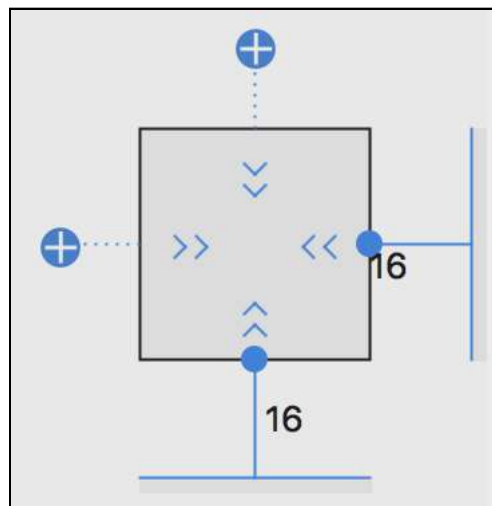
With that, you're ready to use your new action button to add tasks to your list. Open **ListDetailActivity.kt** again and add a new `lateinit` property to the top of the class that will hold the reference for your new button.

```
lateinit var addTaskButton: FloatingActionButton
```

At the bottom of the `onCreate()` method, reference your button and add a new click listener to it:

```
addTaskButton = findViewById<FloatingActionButton>(R.id.add_task_button)
addTaskButton.setOnClickListener {
  showCreateTaskDialog()
}
```

In the click listener, you call a method that will ask the user for the task to be added to the list. You'll create that method next.

Underneath `onCreate()`, add the new method that will prompt your users to add in a new task:

```
private fun showCreateTaskDialog() {
  //1
  val taskEditText = EditText(this)
  taskEditText.inputType = InputType.TYPE_CLASS_TEXT

  //2
  AlertDialog.Builder(this)
    .setTitle(R.string.task_to_add)
    .setView(taskEditText)
    .setPositiveButton(R.string.add_task, { dialog, _ ->
      // 3
      val task = taskEditText.text.toString()
      list.tasks.add(task)
      // 4
      val recyclerAdapter = listItemsRecyclerView.adapter as
ListItemsRecyclerViewAdapter
      recyclerAdapter.notifyItemInserted(list.tasks.size)
      //5
      dialog.dismiss()
    })
    //6
    .create()
    .show()
}
```

The code should look familiar to you, as it's similar to the `showCreateListDialog` method you created in **MainActivity.kt**, but a bit shorter. Here, you've done the following:

1. Created an **EditText** which will receive text input from the user.

2. Created an **AlertDialogBuilder** and used method chaining to set up various aspects of the **AlertDialog**. Method chaining can happen when each method returns a value which can then be used. Here, when any method is called on the Builder, it returns the builder instance, modified with whatever you've just added.

3. In the Positive Button's click listener, accessed the **EditText** which you just created to grab the text input and create a task

4. Still in the click listener, you've notified the **ListItemsRecyclerViewAdapter** that a new item has been added. This gives the adapter a chance to check its datasource (that is, the list), so it can inform the RecyclerView to create any new rows with the new information it has.

5. Finished out the click listener by dismissing the dialog.

6. Back outside the click listener, continued to use method chaining to create then show the **AlertDialog** without ever needing to have the **AlertDialogBuilder** as a separate variable.

As Android Studio will tell you in its angry red text, you're missing some strings. Go to **res\values\strings.xml**. Then add the following new string elements in between the `resources` tags:

```
<string name="task_to_add">What is the task you want to add?</string>
<string name="add_task">Add</string>
```

These will be shown in the app when the user tries to add a new task to a list.

Finally, you need to save any new tasks added to your list. Remember that **ListDataManager** you created when you first began to save lists? It's time to use that again to update your saved list with any new tasks it may have. We'll look at that next.

Getting the list back

ListDataManager is declared and used in **MainActivity.kt**. To save your list with your newly added task you *could* also declare it in **ListDetailActivity.kt** and save your list anytime the user adds a new task.

That would work, but it would mean you've got two separate spots where data is saved, which gives you double the places where bugs could occur. Wouldn't it be better if you could get that list passed into the detail Activity and use the original list data manager?

With a few tweaks and additions, you can do just that! Open up **MainActivity.kt** and edit showListDetail() so it looks like the following:

```
private fun showListDetail(list: TaskList) {
  val listDetailIntent = Intent(this, ListDetailActivity::class.java)
  listDetailIntent.putExtra(INTENT_LIST_KEY, list)

  startActivityForResult(listDetailIntent, LIST_DETAIL_REQUEST_CODE)
}
```

The only change here is the final line: startActivity() has been changed to startActivityForResult(). It may seem small, but the difference is very important. This line starts your detail Activity as intended; however, it also adds the expectation that **MainActivity.kt** will hear back from **ListDetailActivity.kt** once it has finished being onscreen.

Think of it as asking for someone to do something for you, and to report back with the results when they're done. That's exactly what's going on here: You want to hear back about that list you're passing to **ListDetailActivity.kt**.
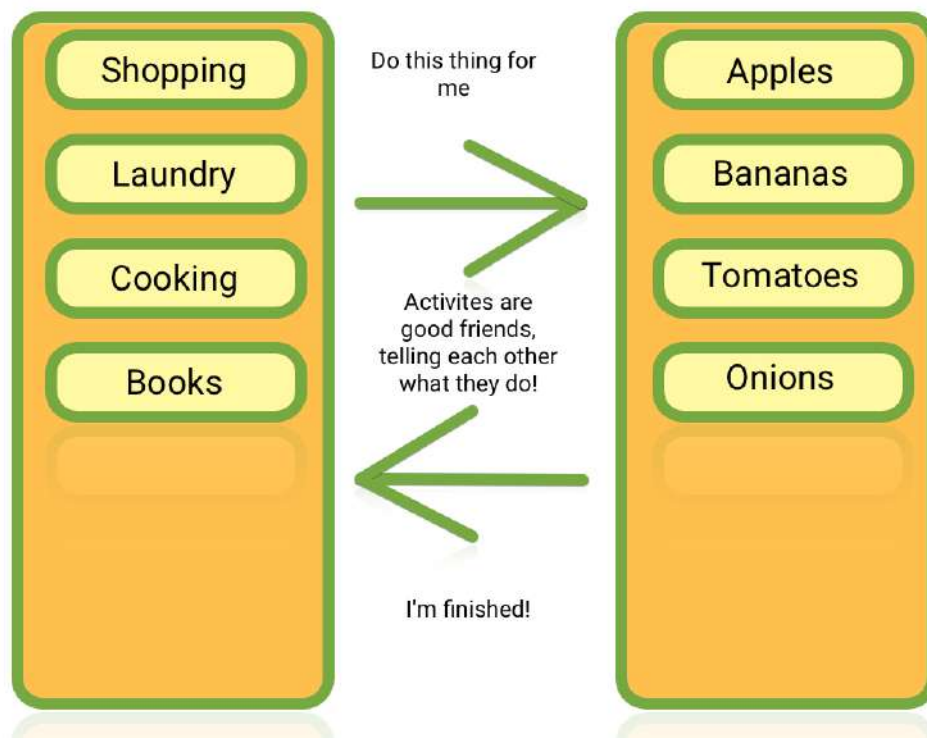
You should notice there is an additional parameter being passed into `startActivityForResult()` too. The second parameter is a request code that lets you know which result you're dealing with.

You could be dealing with multiple Activities that are passing back multiple results, so having a unique way of being able to identify results is really helpful.

Add the request code to the companion object at the top of **MainActivity.kt**, so it looks like the following:

```kotlin
companion object {
  val INTENT_LIST_KEY = "list"
  val LIST_DETAIL_REQUEST_CODE = 123
}
```

Next you need to be able to deal with the returned result. To do that, you need to override a new method in your `MainActivity` named `onActivityResult`. This method allows your Activity to receive the result of any Activities it starts. In this case, it will look for the result that **ListDetailActivity.kt** will provide once it has finished adding tasks to a list:

```kotlin
override fun onActivityResult(requestCode: Int, resultCode: Int, data:
    Intent?) {
  super.onActivityResult(requestCode, resultCode, data)
  // 1
  if (requestCode == LIST_DETAIL_REQUEST_CODE) {
    // 2
    data?.let {
      // 3
      listDataManager.saveList(data.getParcelableExtra(INTENT_LIST_KEY))
      updateLists()
    }
  }
}
```

Let's go through this method step by step:

1. You first check to make sure the request code is the same code you're expecting to get back. You don't want to be dealing with any other requests here.

2. Assuming you're dealing with the request you want, you unwrap the data Intent passed in. It's possible there isn't any data at all here, so it's good to first make sure you have something to deal with.

3. Once you've confirmed there is data to deal with, in this case a new list. You save the list to the list data manager and then call `updateLists()`, which you will create shortly.

> **Note**: You may have noticed the `data?.let` block in the code snippet above. The `.let` function is a short hand method available in Kotlin, allowing you to only execute a block of code if the variable `.let` is used on isn't null.
>
> This is what is meant by unwrapping the data intent in the code snippet. You're trying to unwrap the optional value to get at the actual value.
>
> You can still use a null check like in Java, it's all down to personal preference. All of this stuff falls under the **Null Safety** paradigm of Kotlin, feel free to read more about it here: https://kotlinlang.org/docs/reference/null-safety.html

Underneath `onActivityResult()`, add the `updateLists()` method, like this:

```
private fun updateLists() {
   val lists = listDataManager.readLists()
   listsRecyclerView.adapter =
       ListSelectionRecyclerViewAdapter(lists, this)
}
```

This code will read the saved lists again so your RecyclerView is aware of any new tasks added to a List. It also passes the lists into the detail Activity in case a user decides to perform further updates to lists.

Finally, open **ListDetailActivity.kt** and add a new override method named `onBackPressed()`:

```
override fun onBackPressed() {
   val bundle = Bundle()
   bundle.putParcelable(MainActivity.INTENT_LIST_KEY, list)

   val intent = Intent()
   intent.putExtras(bundle)
   setResult(Activity.RESULT_OK, intent)
   super.onBackPressed()
}
```

This gives you a chance to run some code whenever you tap the back button to get back to the List Activity. In this case, you package up the list in its current state and let **MainActivity.kt** know that everything is OK.

You also quite literally bundle up the passed in list and any new tasks added into a **Bundle** object, and put that into an Intent that will be passed back to **MainActivity.kt**. Finally, you set the result to **RESULT_OK**, informing the Activity that everything happened according to plan.

With that, it's time to test out the app again. Click **Run App** at the top of Android Studio and select your device. Go through the motions of creating a list if necessary, or edit any other list, then once inside a list, add a task on the list detail Activity:
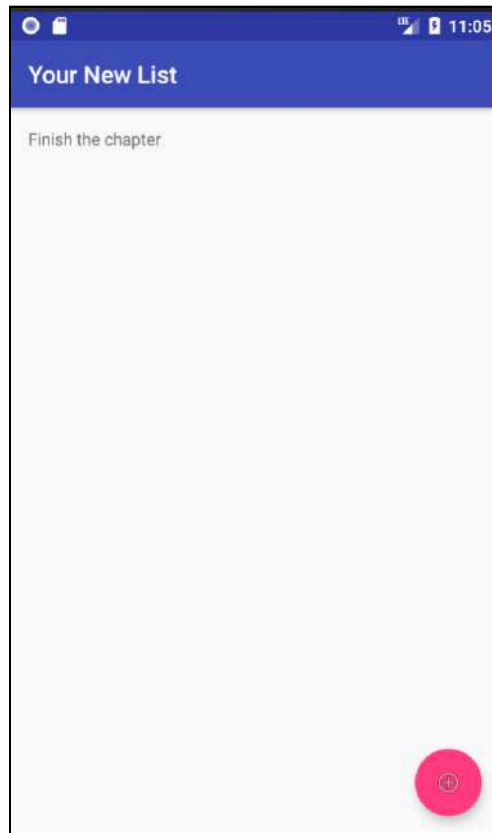
Tap the back button, and then click the list you added a task to. Your task should have persisted and show up proudly! Hopefully you're assured that your list will stay on your phone for a while — even if your battery runs out.

Where to go from here?

Great job! You've reused a lot of your knowledge in this chapter and picked up a few new tricks to reuse code in your apps in a clean way, while also learning how to pass data back and forth between Activities.

You've got yourself a fully functioning list app now, which is perfectly usable. In the next chapter, you'll learn how to take your app and make it work on Android tablets, as well as on Android phones!

Chapter 11: Using Fragments

By Darryl Bayliss

Thanks to the standard set of hardware and software features Android provides across devices, adding new features can be fairly easy. However, when it comes to coding an appealing user interface that adapts across all these devices with varying screen sizes, things can get tricky!

Although you won't be building an app for a fridge just yet (give it time), in this chapter you'll adapt **ListMaker** to make full use of the additional screen space a tablet can provide. Along the way, you'll also learn:

- What Fragments are and how they work with Activities.

- How to split up Activities into Fragments.

- How to provide different layout files for your app depending on the running device screen size.

Getting started

If you are following along with your own app, open it and keep going with it for this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **ListMaker** app under the **starter** folder. The first time you open the project, Android Studio takes a few minutes to set up your environment and update dependencies.

You'll start things off by creating a device that emulates a tablet.

> **Note**: If you have a physical tablet available, feel free to use that instead.

Open up the **ListMaker** project and click the **Android Virtual Device** button along the top of Android Studio.
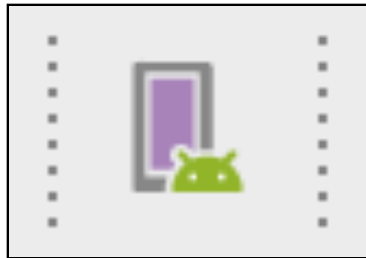
The AVD window will pop up, showing you all the emulators already available on your machine.
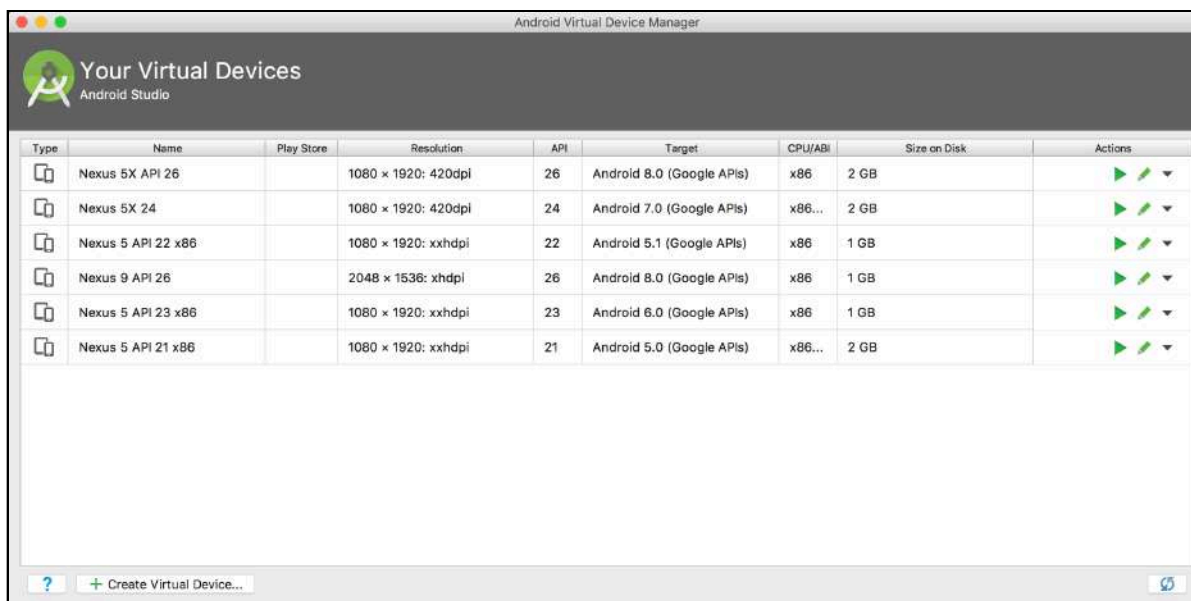
Click the **Create Virtual Device** at the bottom of the window.

A new window will pop up asking what hardware you want your virtual device to emulate.
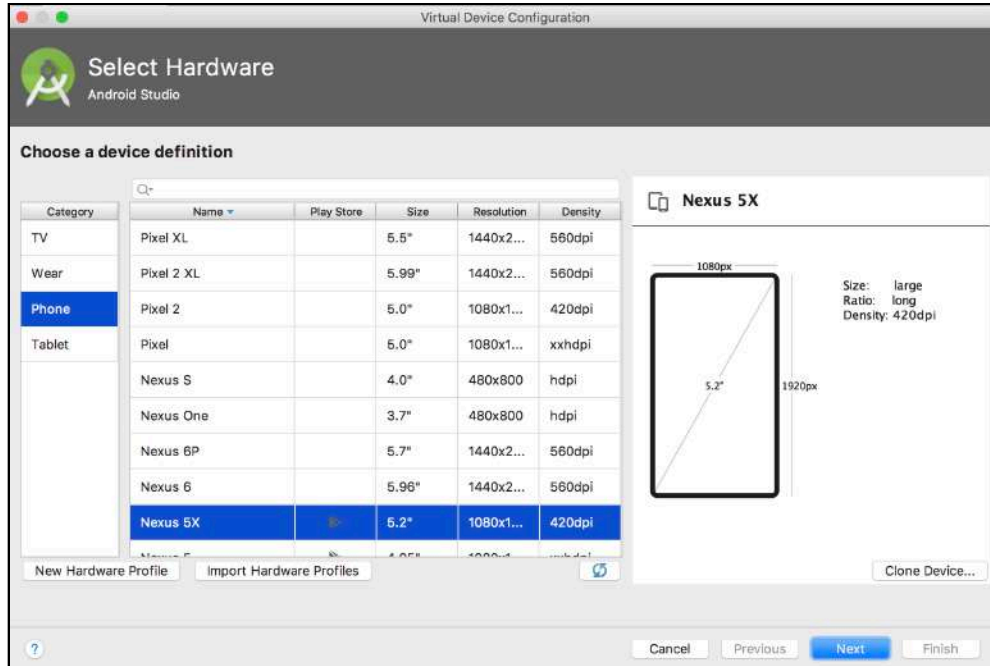
Select the **Tablet** category. Notice the table in the middle of the window has changed to offer a selection of tablets.

The **Pixel C** will do nicely. **Click** the Pixel C row, then in the bottom right, click **Next**.
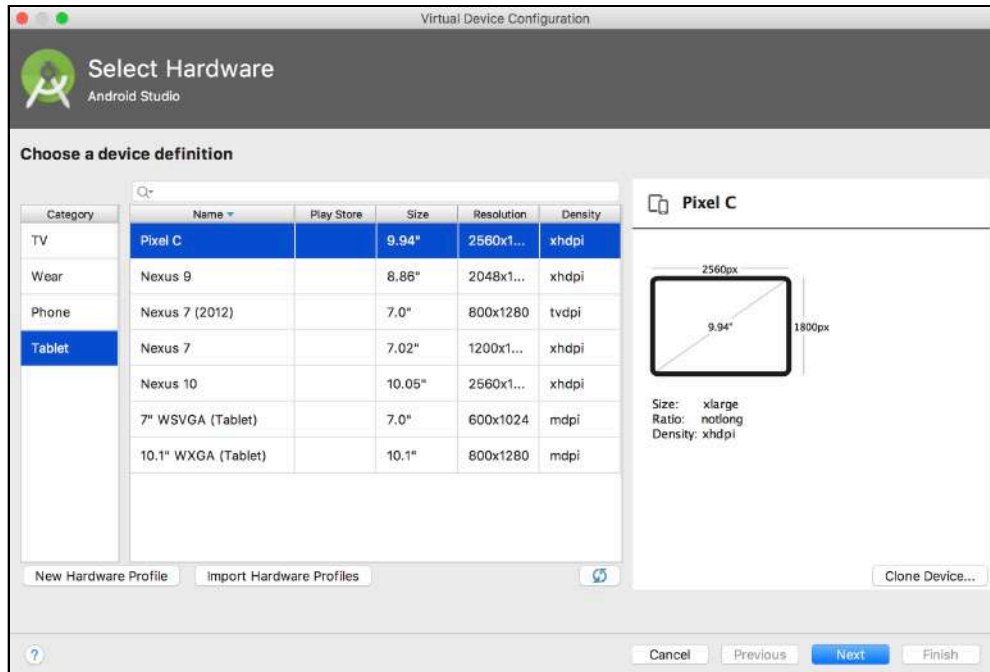
The next screen asks what version of Android you want your device to run. Make sure the highest API level is selected, and click **Next**:
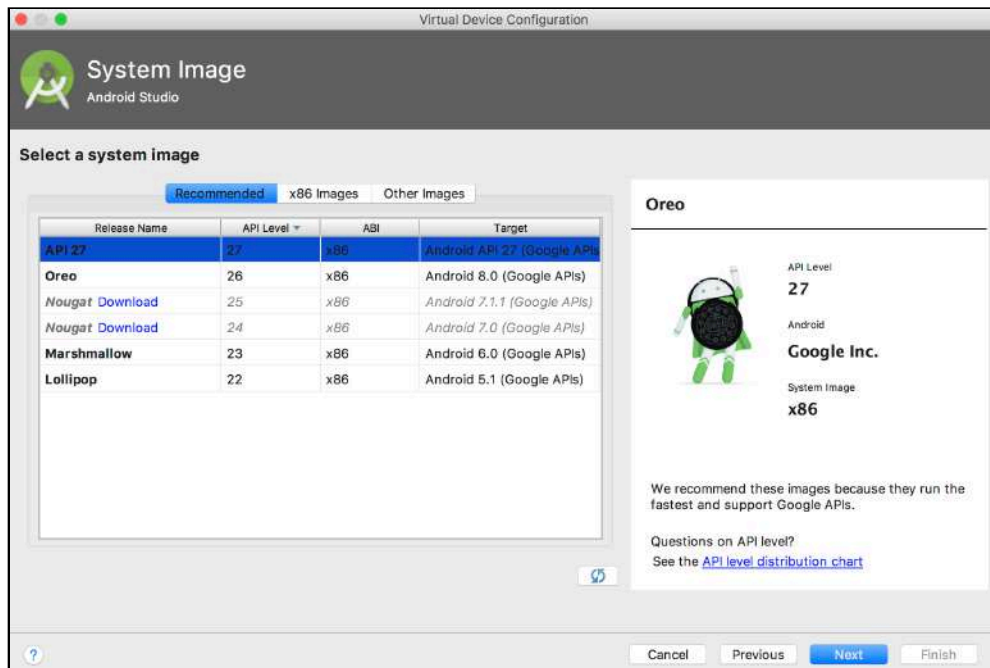
The final screen shows you the configuration for your device, while giving you a chance to change some advanced settings. Don't worry about changing anything here. Click **Finish** to complete setting up your emulator.

Time to run your app on the new emulator. Close the AVD window and click the run app button at the top of Android Studio. In the deployment target window that appears, select the new emulator you created and click **OK**.

When your app loads, you'll be happy to see that it looks exactly as it does on a phone.
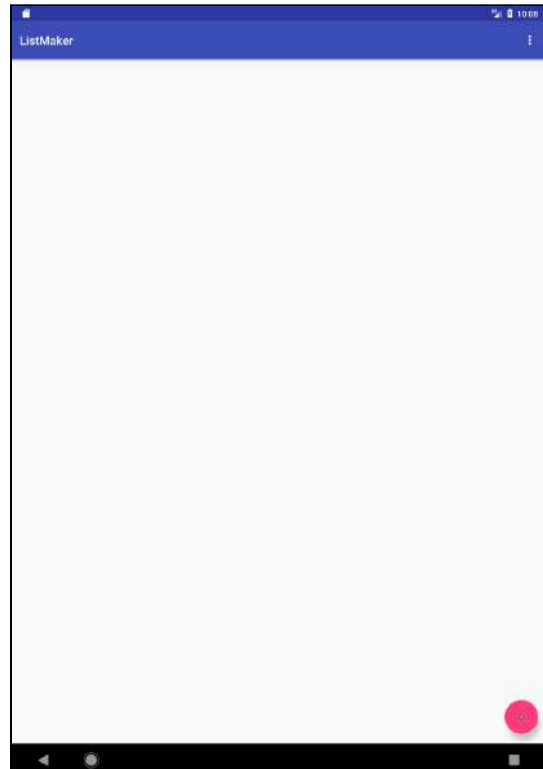
Create some lists and add tasks to each list, taking note of the extra real estate in the app.
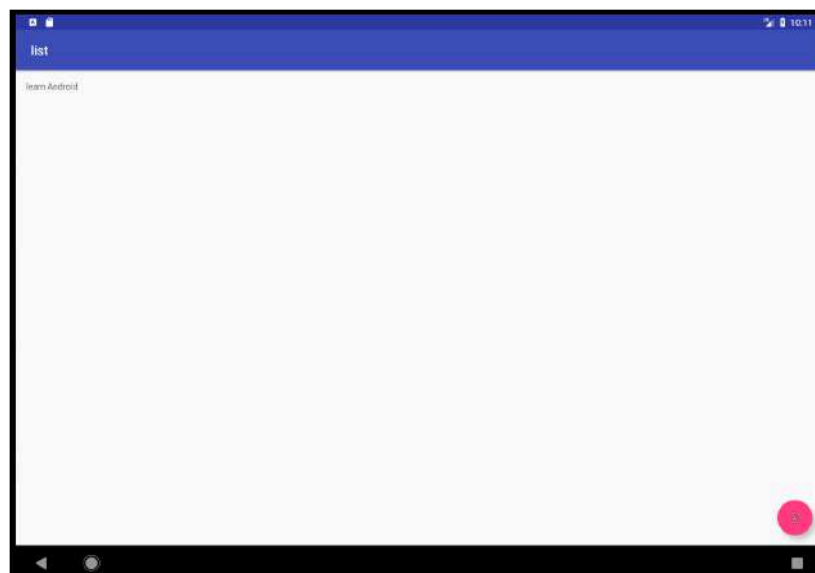
That's a lot of empty space for a simple list app to fill. Although the app works perfectly well on a tablet, its design isn't optimized for the extra space available on the screen. It's situations like this where you need to consider how to make your app adapt to the size of a device's screen.

One solution could be to split the screen in half so one side could show all your lists, while the other half could show the tasks belonging to each list. This would certainly make better use of the real estate available on the tablet screen. Splitting a layout in this way may look better on large screens, but it wouldn't quite work on a small phone screen and could end up being quite unusable.

By the end of this chapter, you will rearrange the code in a way that will support this dual layout based on the device screen size.

This is where the concept of **Fragments** comes in.

Creating a Fragment

In the project navigator to the left of Android Studio, **right-click** on the com.raywenderlich.listmaker package and in the selection dialog that appears, select **New ▸ Fragment ▸ Fragment (Blank)**
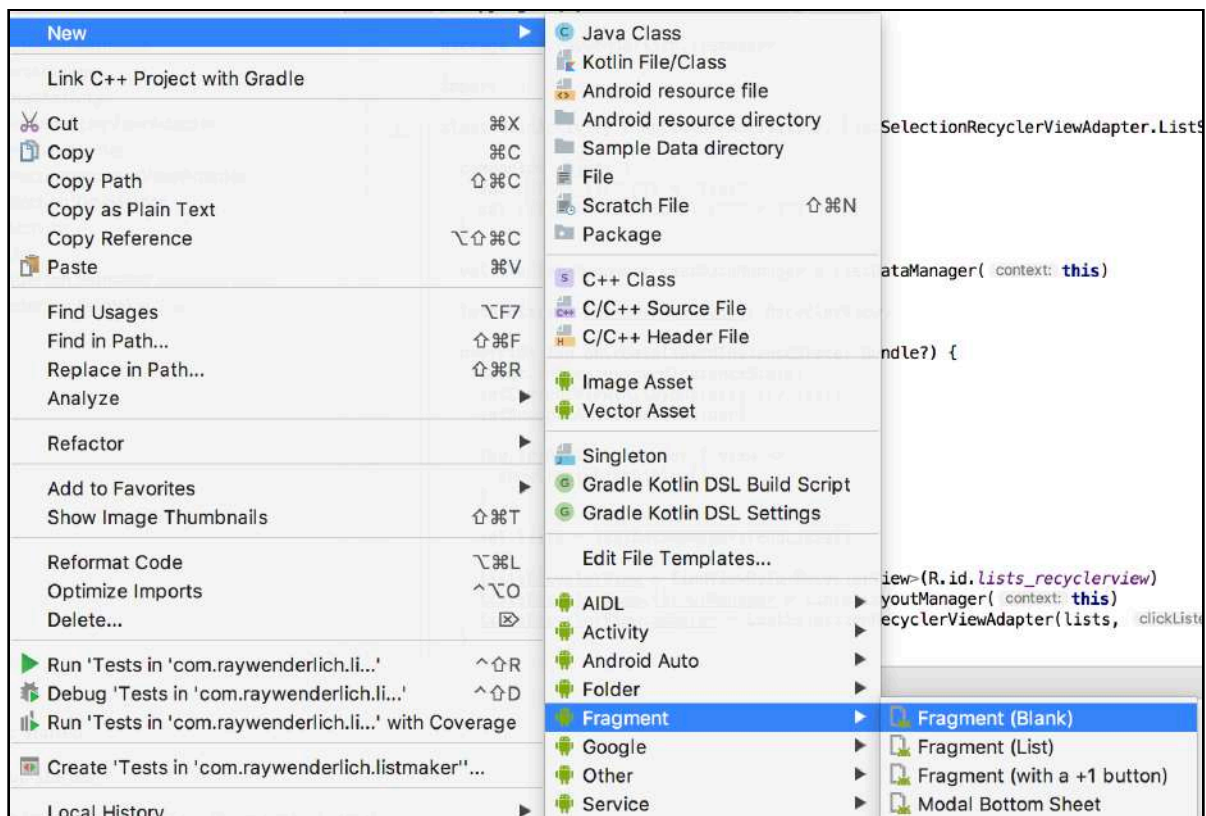
Click Fragment (Blank) and Android Studio will display a new window.
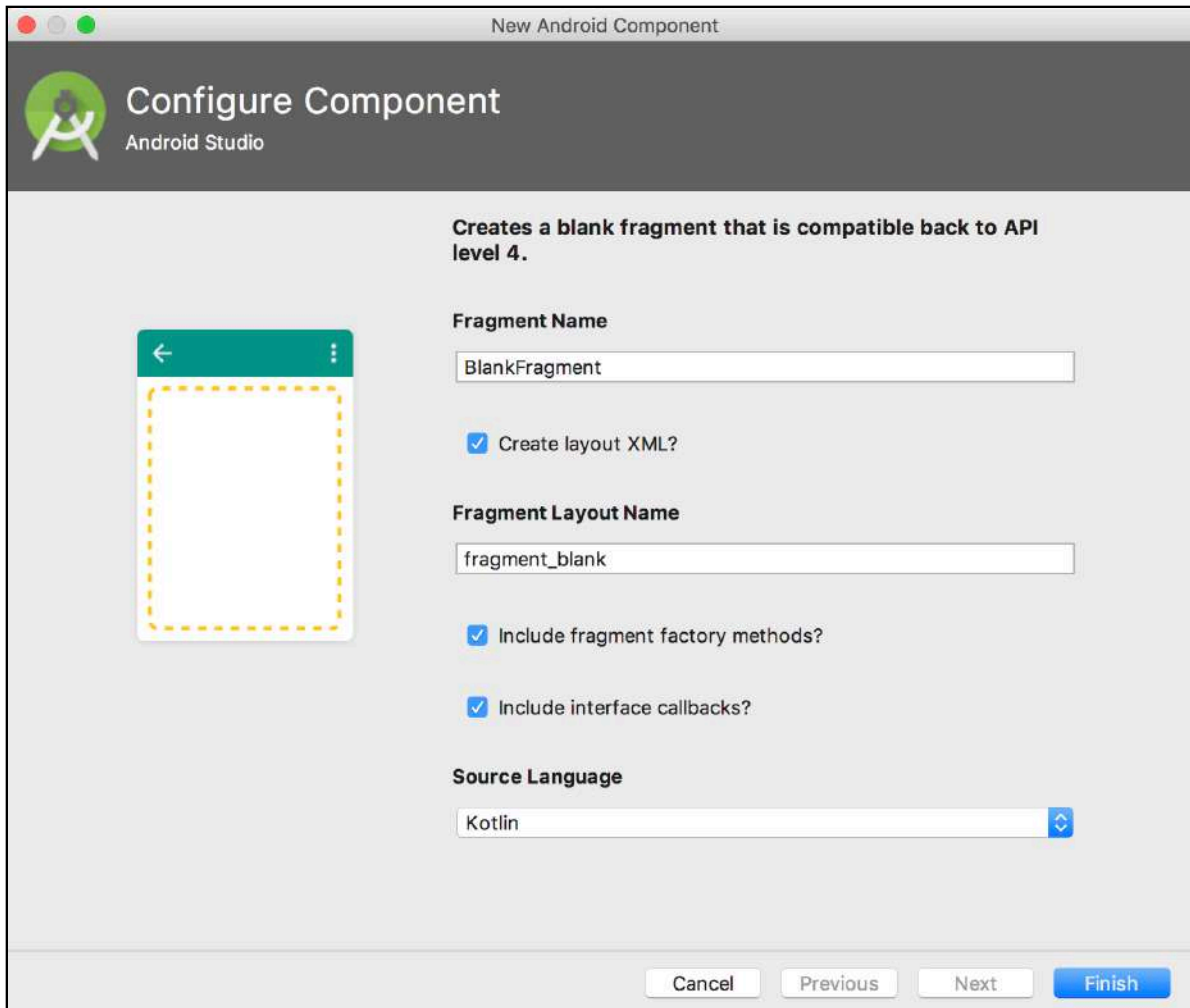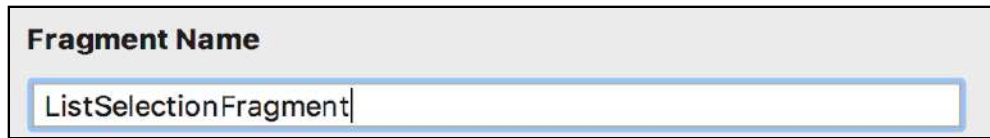
This window is dedicated to creating a new Fragment for your app. Don't worry too much about what a Fragment is for now; just think of it as something similar to an Activity. What you're doing in this window is creating another screen for your app.

Back to the window. Let's go through the options available:
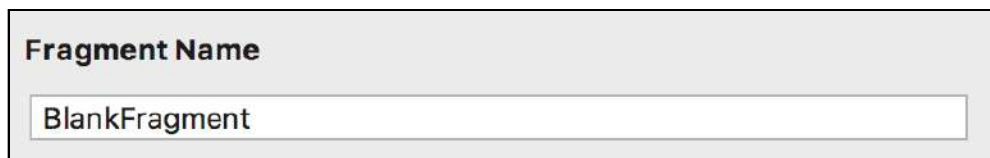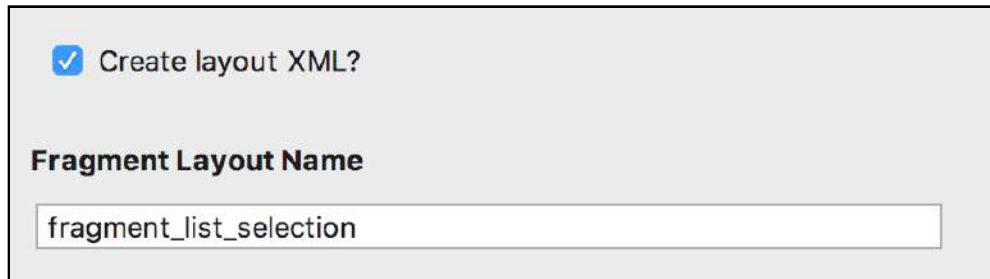
The **Fragment Name** textfield allows you to name your Fragment, just as you can name an Activity.

Change the name in the textfield to **ListSelectionFragment**.

Fragment Name

ListSelectionFragment
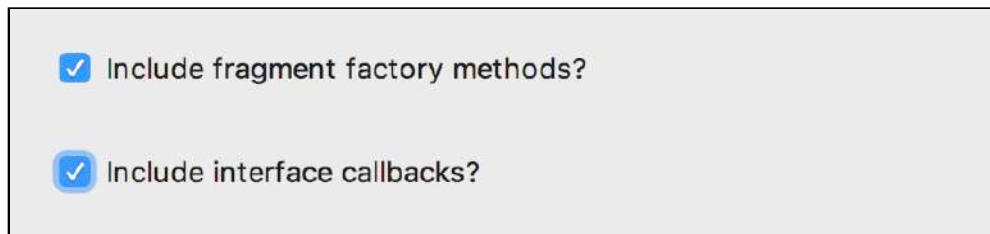
The next options are a checkbox titled **Create Layout XML** and another textfield named **Fragment Layout Name**:

☑ Create layout XML?

Fragment Layout Name

fragment_list_selection

Again, just like an Activity, you can have Android Studio create a layout file for you. You can edit the layout file to customize the screen shown by the Fragment. The **Create Layout XML** checkbox is already checked and Android Studio has filled in the textfield with the name **fragment_list_selection** which is based on the **Fragment Name** you entered.

The next options are a little more complicated:

☑ Include fragment factory methods?

☑ Include interface callbacks?

The first checkbox, **Include fragment factory methods**, tells Android Studio to generate code to instantiate the Fragment.

The second checkbox, **Include interface callbacks**, tells Android Studio to generate an interface to allow other objects to receive callbacks from the Fragment.

You'll learn more about why factory methods and callbacks are useful for a Fragment a little later. The important thing to know now is they are both required for your scenario.

The final option is the **Source Language** option box:

This option tells Android Studio whether you want the generated code for your Fragment to be in Kotlin or Java. Select Kotlin and click the **Finish** button in the bottom right of the window.

What is a Fragment?

It's time to explain the mystery behind Fragments. What exactly are they?

A fragment is frequently a part of an activity's user interface and contributes its own layout to the activity.

This lets you dynamically add and remove pieces of the user interface from the app while it's running. You can use this to your advantage to decide how many Fragments an Activity should have depending on the size of a screen for instance.

If **ListMaker** is running on a tablet, you could have an Activity display two Fragments: one dedicated to selecting a list, and another to display the selected list. If **ListMaker** is running on a phone, you could show only one of the Fragments in an Activity and show the next Activity when a list is clicked.

Fragments give you a lot of power to ensure you're using as much of the available space as possible.

Fragments have their own Lifecycles that work alongside the Activity's lifecycle in which they are embedded. Since it's unknown whether a Fragment will be displayed at runtime, it's important that they be as self contained as possible.

This is why you were given the option earlier of generating a callback interface when you created your first Fragment. This is the way your Fragment communicates with the outside world, without having to rely on anything besides itself.

> **Note**: If you want to read more about Fragments, check out the official documentation available at https://developer.android.com/guide/components/fragments.html

To see how Fragments fit into the bigger picture, you'll start with some code. Open up **ListSelectionFragment.kt** so you can clean up the generated file to be easier to understand.

Change the file so it looks like the following:

```kotlin
class ListSelectionFragment : Fragment() {
  // 1
  private var listener: OnListItemFragmentInteractionListener? = null

  interface OnListItemFragmentInteractionListener {
    fun onListItemClicked(list: TaskList)
  }

  // 2
  companion object {

    fun newInstance(): ListSelectionFragment {
      val fragment = ListSelectionFragment()
      return fragment
    }
  }

// 3
override fun onAttach(context: Context) {
  super.onAttach(context)
  if (context is OnListItemFragmentInteractionListener) {
    listener = context
  } else {
    throw RuntimeException(context.toString() + " must implement
OnListItemFragmentInteractionListener")
  }
}

  // 4
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
  }

  // 5
  override fun onCreateView(inflater: LayoutInflater, container:
ViewGroup?,
             savedInstanceState: Bundle?): View? {
    return inflater.inflate(R.layout.fragment_list_selection, container,
false)
  }

  // 6
  override fun onDetach() {
    super.onDetach()
    listener = null
  }
}
```

There's a lot of code here with various duties, let's go through the file bit by bit:

1. You define a private `OnListItemFragmentInteractionListener` variable to hold a reference to an object that implements the Fragment interface. The interface is also defined underneath, requiring a single method to be implemented informing objects that a list has been clicked. **MainActivity** will implement this interface.

2. You define a companion object here with a `newInstance()` method inside. This will be used by any object that wants to create a new instance of the Fragment.

3. This is the first lifecycle method run by a Fragment. Fragments have lifecycle methods very similar to like Activities. This method is run when the Fragment is first associated with an Activity, which gives you a chance to set up anything required before the Fragment is created. In this method, you assign the context of the Fragment to the `listener` variable. This context will be the **MainActivity** that will contain the Fragment.

4. The next overridden lifecycle method is `onCreate(savedInstanceState: Bundle?)`. This functions similarly to the method of the same name in an Activity, except it's used when a Fragment is in the process of being created.

5. Another lifecycle method, this one named `onCreateView()`. This is where the Fragment acquires the layout it must have in order to be presented within the Activity. Here, a layout inflater is used to inflate the layout and pass it back to the Fragment.

6. This is the final lifecycle method to be called by a Fragment. This is called when a Fragment is no longer attached to an Activity, which could be due to the Activity being destroyed or the Fragment being removed. At this point within the method, `listener` is set to null as the Activity is no longer available.

From Activity to Fragments

With the code cleaned up, the next job is to move parts of **MainActivity.kt** and its associated layout to your new Fragment.

Remember that splitting up your code into individual, isolated Fragments makes them reusable. It's essential that the Fragment needs nothing inside the Activity.

Open **MainActivity.kt** and move the following member variables:

```
val listDataManager: ListDataManager = ListDataManager(this)
lateinit var listsRecyclerView: RecyclerView
```

To the top of **ListSelectionFragment.kt** with a slight modification:

```
lateinit var listDataManager: ListDataManager
lateinit var listsRecyclerView: RecyclerView
```

You should notice that we're no longer initializing listDataManager inline since Fragments do not extend from Context. This means you'll have to initialize listDataManager at the earliest moment you get a Context, which, just happens to be in onAttach.

Update onAttach() in the Fragment so it instantiates the ListDataManager when the Activity is attached:

```
override fun onAttach(context: Context) {
  super.onAttach(context)
  if (context is OnListItemFragmentInteractionListener) {
      listener = context
      listDataManager = ListDataManager(context)
  } else {
      throw RuntimeException(context.toString() + " must implement
OnListItemFragmentInteractionListener")
  }
}
```

Wonderful — your ListDataManager works exactly the same, except it now gets the Context via the fragment. You will notice errors in **MainActivity.kt** after you remove the last two variables. Let's fix that up.

In onCreate() from **MainActivity.kt**, copy then remove the following lines (we'll recreate them shortly inside the Fragment):

```
val lists = listDataManager.readLists()

listsRecyclerView = findViewById<RecyclerView>(R.id.lists_recyclerview)
listsRecyclerView.layoutManager = LinearLayoutManager(this)
listsRecyclerView.adapter = ListSelectionRecyclerViewAdapter(lists, this)
```

You need to move these into a new lifecycle method in the Fragment, named onActivityCreated(). This method runs when the Activity the Fragment is attached to has finished running onCreate(). This ensures you have an Activity to work with and something to show your widgets.

Add the complete onActivityCreated(). to **ListSelectionFragment.kt**:

```
override fun onActivityCreated(savedInstanceState: Bundle?) {
  super.onActivityCreated(savedInstanceState)

  val lists = listDataManager.readLists()
  view?.let {
    listsRecyclerView =
```

```
it.findViewById<RecyclerView>(R.id.lists_recyclerview)
    listsRecyclerView.layoutManager = LinearLayoutManager(activity)
    listsRecyclerView.adapter = ListSelectionRecyclerViewAdapter(lists,
this)
  }
}
```

The next item is to move is the `ListSelectionRecyclerViewClickListener` interface implementation. This is the interface `ListSelectionRecyclerViewAdapter` provides to inform any interested objects that it detected a list was selected. Since the `ListSelectionRecyclerViewAdapter` no longer exists in **MainActivity.kt**, you can move that to your new Fragment. Your Activity, however, still needs to be aware of the list click event.

This is because only Activities can start other Activities. Fragments, being isolated views, should only inform Activities of any events it should handle.

You may recall the `ListSelectionFragment` provides an interface you can use to talk back to its Activity. At the top of the **MainActivity.kt** class, change the class declaration as follows:

```
class MainActivity : AppCompatActivity(),
ListSelectionFragment.OnListItemFragmentInteractionListener {
```

The only change here is the Activity now implements your Fragments interface.

In `MainActivity`, replace `listItemClicked()` from `ListSelectionRecyclerViewClickListener` with `onListItemClicked()` from `OnListItemFragmentInteractionListener`:

```
override fun onListItemClicked(list: TaskList) {
   showListDetail(list)
}
```

Similar to the `ListSelectionRecyclerViewClickListener` interface, when this method runs, it shows the detail of the `TaskList` in another Activity.

The `ListSelectionRecyclerViewClickListener` interface method now needs to be moved into the Fragment. The Fragment also needs to implement the `ListSelectionRecyclerViewClickListener` interface so it can receive the list item click to pass up to the Activity.

Update the Fragment declaration so it implements the interface:

```
class ListSelectionFragment : Fragment(),
ListSelectionRecyclerViewAdapter.ListSelectionRecyclerViewClickListener {
```

Add the following method inside the Fragment Class:

```
override fun listItemClicked(list: TaskList) {
  listener?.onListItemClicked(list)
}
```

When the method receives an item click from the RecyclerView Adapter, it uses the `listener` variable to inform the Activity that it has received a item click.

This in turn allows the Activity to receive the list and to start up a new Activity to show the list while keeping your app logic intact.

So far so good. There are still a few things to move over to your Fragment, so keep at it. The next piece of logic to move over to your Fragment is adding a list to the Data Manager.

The data manager is now handled by your Fragment, but you still need to maintain the previous logic that was relied on by the Activity. To make sure you can still use the logic your Fragment will be responsible for, you need a reference to your Fragment.

At the top of the Activity, add the following line:

```
  private var listSelectionFragment: ListSelectionFragment =
ListSelectionFragment.newInstance()
```

This line creates a new instance of your Fragment when your Activity is created. In the `showCreateListDialog` method of your Activity, change the positive button click listener to the following:

```
builder.setPositiveButton(positiveButtonTitle, { dialog, i ->

  val list = TaskList(listTitleEditText.text.toString())
  listSelectionFragment.addList(list)

  dialog.dismiss()
  showListDetail(list)
})
```

The change is subtle, but important. What you've done is taken out the lines that added the list to the Data Manager, and replaced them with a method call on your Fragment.

The logic that was once here needs to be moved to the Fragment, so let's do that now. In **ListSelectionFragment.kt**, add the following method:

```
fun addList(list : TaskList) {

  listDataManager.saveList(list)

  val recyclerAdapter = listsRecyclerView.adapter as
ListSelectionRecyclerViewAdapter
  recyclerAdapter.addList(list)
}
```

Next, you'll have to save a list that is returned from the List Detail Activity. Again this needs to be handled by your Fragment.

In **MainActivity.kt**, change `onActivityResult` so the parcelable extra is passed into a method provided by your Fragment:

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data:
Intent?) {
  super.onActivityResult(requestCode, resultCode, data)

  if (requestCode == LIST_DETAIL_REQUEST_CODE) {
    data?.let {

listSelectionFragment.saveList(data.getParcelableExtra<TaskList>(INTENT_L
IST_KEY))
    }
  }
}
```

In **ListSelectionFragment.kt**, add the following method so your Fragment saves the updated state of the list and updates the RecyclerView:

```
fun saveList(list: TaskList) {
  listDataManager.saveList(list)
  updateLists()
}
```

Finally move `updateLists()` from the Activity, straight into your Fragment.

```
private fun updateLists() {
  val lists = listDataManager.readLists()
  listsRecyclerView.adapter = ListSelectionRecyclerViewAdapter(lists,
this)
}
```

Showing the Fragment

So far you've spent most of your time moving logic from your Activity to your Fragment. If you recall, you also need to adjust your layouts so your `RecyclerView` is provided by the Fragment.

You also need to ensure your Activity layout is aware it needs to show the Fragment. This means you're going to have to delve into the not-quite-so pretty side of layouts.

Open up **content_main.xml** available in **res ▸ layout**. If not already selected, select the **Text** tab in the bottom left corner of the layout editor:

```
Design        Text
```

Your editor should update to look like something that resembles code, rather than a user interface:

```xml
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout 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:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="android.support.design.widget.AppBarLayout$ScrollingVie..."
    tools:context="com.raywenderlich.listmaker.MainActivity"
    tools:showIn="@layout/activity_list">

    <FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/fragment_container"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

</android.support.constraint.ConstraintLayout>
```

Up until now, you've used the design tab to create your layouts. For this part, you need to copy widgets across various files. For this it's easier to work with XML representation of your widget. Let's begin.

Copy the entirety of the `android.support.v7.widget.RecyclerView` tag:

```xml
<android.support.v7.widget.RecyclerView
    android:id="@+id/lists_recyclerview"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:layout_editor_absoluteX="8dp"
    tools:layout_editor_absoluteY="8dp" />
```

Make sure you delete the `RecyclerView` tag as well from **content_main.xml**. Open up the **fragment_list_selection.xml** layout, select the **text** tab and paste the `RecyclerView` over the generated textview:

```
<android.support.v7.widget.RecyclerView
    android:id="@+id/lists_recyclerview"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:layout_editor_absoluteX="8dp"
    tools:layout_editor_absoluteY="8dp"  />
```

With the `RecyclerView` in its new home, it's time to give the Fragment a home. Back in **content_main.xml**, add a **FrameLayout** in between the ConstraintLayout tags:

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/fragment_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

FrameLayout lets you allocate space for a single item. This is perfect for something like a Fragment that could take up an entire screen. You also give it an ID so you can reference it in your Activity.

Open up **MainActivity.kt** and add a variable to hold a reference to the FrameLayout at the top of the file:

```
private var fragmentContainer: FrameLayout? = null
```

Update `onCreate()` in your Activity so it grabs the reference to the FrameLayout via the ID you assigned it in your layout. Add this code just before `fab.setOnClickListener`:

```
fragmentContainer = findViewById(R.id.fragment_container)

supportFragmentManager
    .beginTransaction()
    .add(R.id.fragment_container, listSelectionFragment)
    .commit()
```

Notice the use of `supportFragmentManager`? A **FragmentManager** is an object that lets you dynamically add and remove Fragments at runtime. This gives you a powerful tool to make your UI as fluid as possible for various screens.

The `FragmentManager` uses a **FragmentTransaction** to begin manipulation of any Fragments that need to change. This is performed via `beginTransaction()`.

Once the transaction has begun, you then call `add()` which adds a Fragment into a container view that will hold the Fragment.

The `add()` method takes two parameters to do this: the ID of the container view, and an instance of the Fragment to be shown. You pass in the ID of the FrameLayout and the instance of the ListSelectionFragment your Activity has.

Once the transaction has been defined, `commit()` informs the FragmentManager that it should begin manipulating the Fragments as intended.

Finally, build and run your app. Click the **Run App** button at the top of Android Studio, making sure you run the app on the Tablet Emulator created earlier.
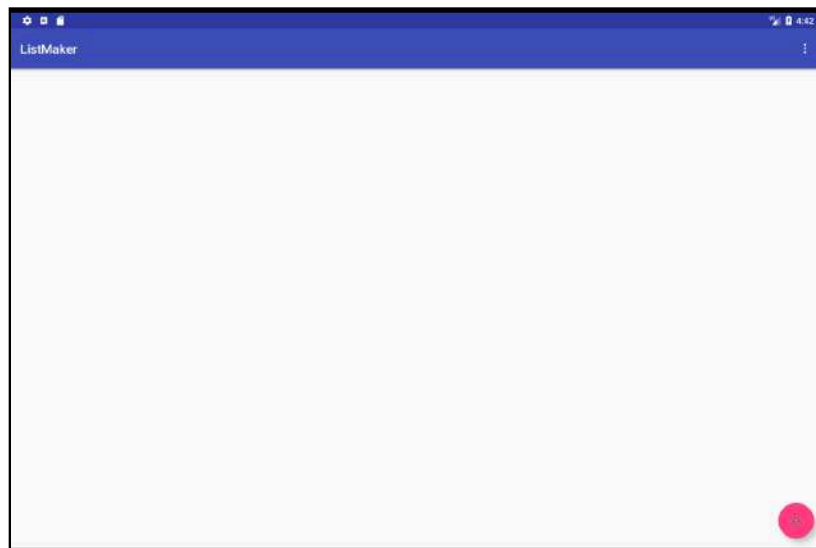
Your app doesn't look any different at this point, but under the hood you are now using an Activity that contains a Fragment. This is a good foundation to start making use of all that space on the tablet screen.

The next step is to replicate the `ListDetailActivity` screen into its own Fragment.

Creating your next Fragment

Right click on the `com.raywenderlich.listmaker` package in the project navigator, and select **New ▸ Fragment ▸ Fragment (Blank)**.

The "Create Fragment" window you used earlier will pop up. Change the fragment name to **ListDetailFragment** and click **Finish** in the bottom right.

Android Studio will create a **ListDetailFragment.kt** and a **fragment_list_detail.xml** file for your Fragment. Clean up the Fragment of any generated code that isn't needed.

Your final **ListDetailFragment.kt** should look like the following:

```kotlin
class ListDetailFragment : Fragment() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
  }

  override fun onCreateView(inflater: LayoutInflater?, container:
ViewGroup?,
                            savedInstanceState: Bundle?): View? {
    // Inflate the layout for this fragment
    return inflater!!.inflate(R.layout.fragment_list_detail, container,
false)
  }

  companion object {

    fun newInstance(list: TaskList): ListDetailFragment {
      val fragment = ListDetailFragment()
      val args = Bundle()
      args.putParcelable(MainActivity.INTENT_LIST_KEY, list)
      fragment.arguments = args
      return fragment
    }
  }
}
```

The main change here to the original code is the bundle arguments passed in via
`newInstance()`. It now expects a `TaskList` to be passed in, since this Fragment is
responsible for showing your list.

Next, copy some of the variables in **ListDetailActivity.kt** over to your new Fragment.
From the top of the Activity, copy the following lines over to the top of the Fragment:

```kotlin
lateinit var list: TaskList

lateinit var listItemsRecyclerView: RecyclerView
```

Update `onCreate()` in the Fragment so it grabs the list from the bundle passed in:

```kotlin
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    list = arguments.getParcelable(MainActivity.INTENT_LIST_KEY)
}
```

Change `onCreateView()` to set up the RecyclerView via the ID in the Layout and
initialize the Adapter and LayoutManager:

```kotlin
override fun onCreateView(
  inflater: LayoutInflater?,
  container: ViewGroup?,
  savedInstanceState: Bundle?): View? {

  // Inflate the layout for this fragment
  val view = inflater!!.inflate(R.layout.fragment_list_detail, container,
false)

  view?.let {
    listItemsRecyclerView =
it.findViewById<RecyclerView>(R.id.list_items_reyclerview)
    listItemsRecyclerView.adapter = ListItemsRecyclerViewAdapter(list)
    listItemsRecyclerView.layoutManager = LinearLayoutManager(activity)
  }

  return view
}
```

Finally, add a method named `addTask` to the Fragment. This method allows the
Fragment to add new tasks to the list:

```kotlin
fun addTask(item: String) {

    list.tasks.add(item)

    val listRecyclerAdapter =  listItemsRecyclerView.adapter as
ListItemsRecyclerViewAdapter
    listRecyclerAdapter.list = list
    listRecyclerAdapter.notifyDataSetChanged()
}
```

You'll use this method later to instruct your Fragment to add tasks to your list.

Now that your Fragment is using the `RecyclerView`, you also need to make sure that it
exists in the Fragment Layout. Repeating your approach from previous Activity, you
need to copy over the `RecyclerView` from the `ListDetailActivity` layout to the
`ListDetailFragment` layout.

With the **Text** editor open, in **layout ‣ activity_list_detail.xml**, copy the following lines from the Layout:

```
<android.support.v7.widget.RecyclerView
    android:id="@+id/list_items_reyclerview"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Open **layout ‣ fragment_list_selection.xml** and paste the `RecyclerView` in between the FrameLayout tags, once again replacing the **TextView** that was auto generated when you created the Fragment.

Remove the lines are begin with **app:layout_constraint**. You no longer need these now that your RecyclerView is sitting within a FrameLayout.

Bringing the Activity into action

So far you've focused on transferring code over from your old Activities to your new Fragments. Remember though, that Fragments need to exist within an Activity to be of any use. It needs to be able to coordinate how it communicates with the Fragment and when it appears on the screen.

Your final job for this chapter is to make sure that **MainActivity.kt** is able to intelligently show your new Fragments at the right time, that it provides information to each Fragment, and lets your app shift its appearance depending on the device it runs on.

First, you need to a create a layout that works for a large screen. In the project navigator, **right-click** on the layout folder, then select **New ‣ Layout resource file**:
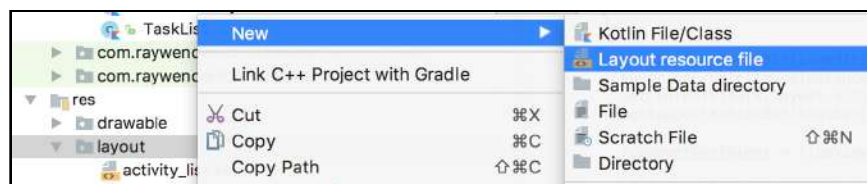
Click the highlighted option to show the new resource file window:
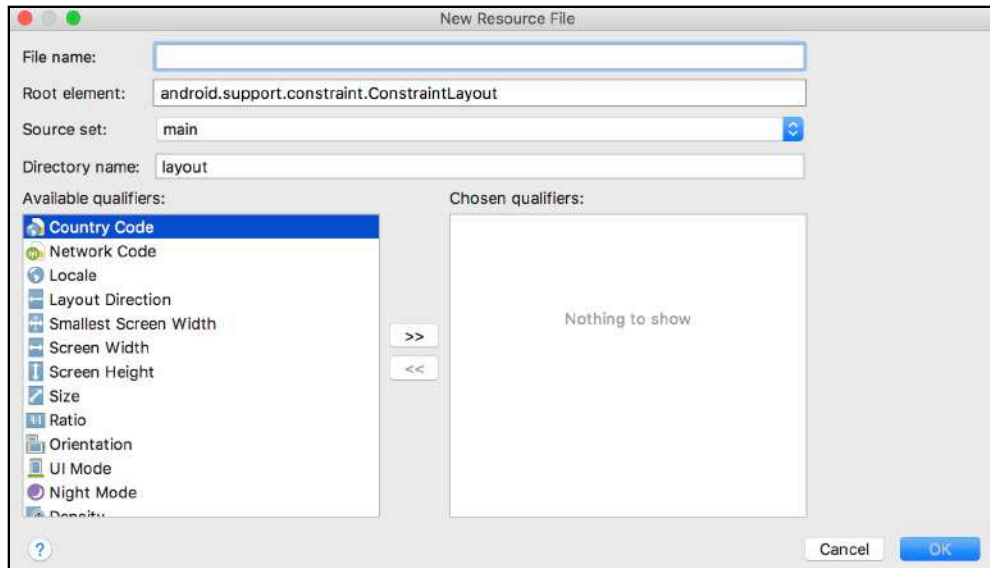
You're creating a new layout file as you've done before. There's a tiny difference here though: you're creating a version of the **content_main.xml** layout that will only display on large screens.

This gives you the option to customize your UI for various sizes of screens. Android is even intelligent enough to distinguish which layout should be used as well. Very helpful!

In the **File name** textfield, name your layout **content_main**. In the **Available qualifiers** list, select the **size** option and click the **>>**.
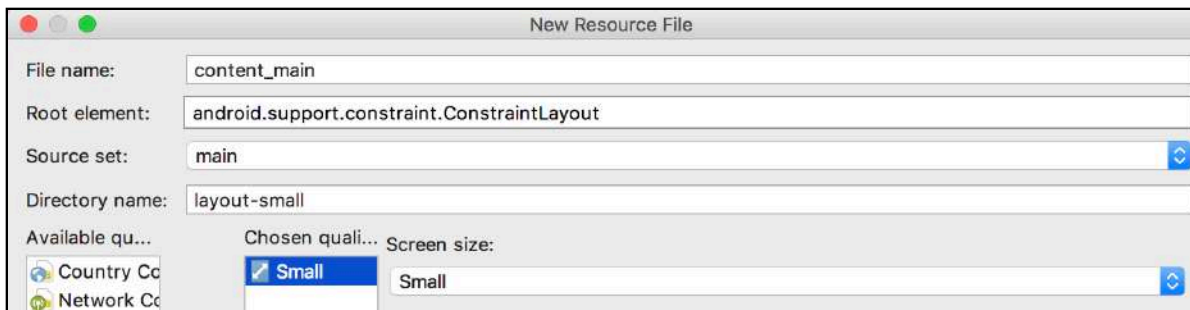
From here, you can select various screen sizes to determine at which sizes your layout will be used. In the screen size dropdown, choose the **X-Large** option.
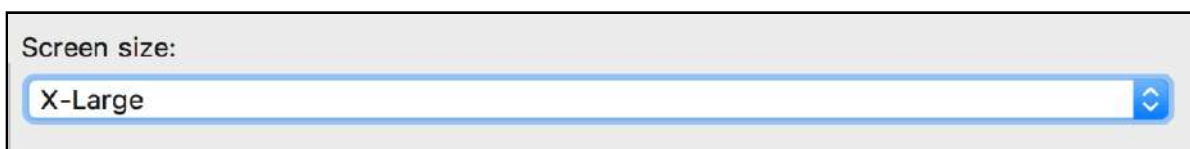
Click **OK** in the bottom right and Android Studio will create your new layout for you. Take a moment to look at the project navigator to the left:
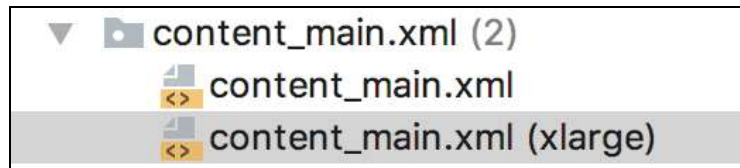
Android Studio shows both your layout files together, and even shows the qualifier you set to distinguish between the two. Now you just have to populate it with the layout you'd like.

You're going to use the Text editor again for this, as it's faster for this particular task. Ensure the text editor tab is selected at the bottom of the layout editor window, and replace the existing XML with the following code for the entire layout:

```xml
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout 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:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:context="com.raywenderlich.listmaker.MainActivity"
    tools:showIn="@layout/activity_list">
    <!-- 1 -->
    <fragment
        android:id="@+id/list_selection_fragment"
        android:name="com.raywenderlich.listmaker.ListSelectionFragment"
        android:layout_width="300dp"
        android:layout_height="match_parent"
        android:layout_marginBottom="0dp"
        android:layout_marginLeft="0dp"
        android:layout_marginStart="0dp"
        android:layout_marginTop="8dp"
        android:layout_weight="1"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
    <!-- 2 -->
    <FrameLayout
        android:id="@+id/fragment_container"
        android:layout_width="0dp"
        android:layout_height="0dp"
        android:layout_marginBottom="0dp"
        android:layout_marginEnd="0dp"
        android:layout_marginLeft="0dp"
        android:layout_marginStart="0dp"
        android:layout_marginTop="0dp"
        android:layout_weight="2"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
```

```
        app:layout_constraintHorizontal_bias="1.0"
        app:layout_constraintStart_toEndOf="@+id/list_selection_fragment"
        app:layout_constraintTop_toTopOf="parent" />

</android.support.constraint.ConstraintLayout>
```

`ConstraintLayout` should be familiar to you now. But what is going on with the Fragment and `FrameLayout`? When the larger layout is shown on a large screen, you want both the **ListSelectionFragment** and **ListDetailFragment** to appear.

The list selection fragment will be static and never hidden, so you dedicate an entire fragment tag to it. You even tell it which Fragment to use via the `android:name` attribute.

The FrameLayout is where the list detail fragment will sit. This is changeable because you want to show different lists depending on which list is selected in the selection fragment. Rather than update the entire Fragment, it's easier to load up a new one, that contains the list for the newly selected Fragment.

Open the original **content_main.xml** file and change the tag in between the `ConstraintLayout` tags so it only expects a single Fragment:

```
<fragment
    android:id="@+id/list_selection_fragment"
    android:name="com.raywenderlich.listmaker.ListSelectionFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_marginBottom="0dp"
    android:layout_marginLeft="0dp"
    android:layout_marginStart="0dp"
    android:layout_marginTop="8dp"
    android:layout_weight="1"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

This change makes it easier for you work out whether your app is running on a device with a large screen. You'll investigate this in detail later on.

You now need to change **MainActivity.kt** so it can handle both layouts. The first thing you need is a way to know if you're using the larger layout.

At the top of the file, add a Boolean that will track whether your larger layout is in use. You need a `ListDetailFragment` instance later on, so you'll create a property for it while you're here:

```
private var largeScreen = false
private var listFragment : ListDetailFragment? = null
```

In onCreate(), use the supportFragmentManager to find your ListSelectionFragment by its identifier, as well as the FrameLayout. Because your FrameLayout only exists in the larger layout, you can use a null check here to find out whether your larger layout is in use.

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_list)
  setSupportActionBar(toolbar)

  listSelectionFragment =
supportFragmentManager.findFragmentById(R.id.list_selection_fragment) as
ListSelectionFragment

  fragmentContainer = findViewById<FrameLayout>(R.id.fragment_container)

  largeScreen = fragmentContainer != null

  fab.setOnClickListener { view ->
    showCreateListDialog()
  }
}
```

Update showListDetail() so it uses the Boolean to work out whether it should show the Activity, or replace the ListDetailFragment shown by using the supportFragmentManager. If one is showing, then it will automatically show the other one instead:

```
private fun showListDetail(list: TaskList) {

  if (!largeScreen) {

    val listDetailIntent = Intent(this, ListDetailActivity::class.java)
    listDetailIntent.putExtra(INTENT_LIST_KEY, list)

    startActivityForResult(listDetailIntent, LIST_DETAIL_REQUEST_CODE)
  } else {
    title = list.name

    listFragment = ListDetailFragment.newInstance(list)

    supportFragmentManager.beginTransaction()
            .replace(R.id.fragment_container, listFragment,
getString(R.string.list_fragment_tag))
            .addToBackStack(null)
            .commit()

    fab.setOnClickListener { view ->
      showCreateTaskDialog()
    }
  }
}
```

Note that you're trying to get a string above to use with the supportFragmentManager.

This string is known as a **tag** and is used by the supportFragmentManager incase you wanted to reference it in the future. You need to add that string to your **strings.xml**.

Open up **res > values > strings.xml** and add the following string:

```
<string name="list_fragment_tag">List Fragment</string>
```

Note: If you get an error stating that list_fragment_tag *is unresolved*, this usually means that Android Studio hasn't recompiled the project's R file. Hit that build button in the top toolbar, or from the menu *Build*, select *Make Project*.

Back in **MainActivity.kt**. You must also change the behavior of the FloatingActionButton when adding tasks to a list. Since the RecyclerView was moved into the Fragment, you will see a compilation error at this point. Modify the positive button callback to pass in the newly created task through the Fragment by adding the following method to **MainActivity.kt**:

```kotlin
private fun showCreateTaskDialog() {
  val taskEditText = EditText(this)
  taskEditText.inputType = InputType.TYPE_CLASS_TEXT

  AlertDialog.Builder(this)
      .setTitle(R.string.task_to_add)
      .setView(taskEditText)
      .setPositiveButton(R.string.add_task, { dialog, _ ->
        val task = taskEditText.text.toString()
        listFragment?.addTask(task)
        dialog.dismiss()
      })
      .create()
      .show()
  }
```

Override onBackPressed() so your Activity knows how to deal with the back button being pressed.

```kotlin
override fun onBackPressed() {
  super.onBackPressed()

  title = resources.getString(R.string.app_name)

  // 1
  listFragment?.list?.let {
    listSelectionFragment?.listDataManager?.saveList(it)
  }

  // 2
  if (listFragment != null) {
    supportFragmentManager
        .beginTransaction()
```

```
          .remove(listFragment)
          .commit()
    listFragment = null
  }

  // 3
  fab.setOnClickListener { view ->
    showCreateListDialog()
  }
}
```

1. Since you aren't using two Activities, you cannot rely on `onActivityResult` to update your `ListDataManager` with any updates made to your list. Therefore, you need to tell the `ListDataManager` to save the list.

2. Remove the detail fragment from the layouts. Since a user can tap the back button as much as they wish, you'll have to make sure that the detail fragment is only removed once.

3. Reassign the FAB to create lists again.

With that done, you are ready to see your hard work in action! Run your app on a phone-sized device and start playing around with it, and you will see it working as expected:

Run your app on a tablet-sized device and you will immediately see the difference.

Your app has now displays two different screens, at the same time, making better use of the available space!

Where to go from here?

Fragments are a very difficult concept to grasp in Android. What you've encountered here is a brief dip into the benefits they can provide.

Any app that wants to succeed across multiple devices and multiple size classes needs to use Fragments to ensure it provides the best experience for users.

Chapter 12: Material Design

By Darryl Bayliss

When it comes to building apps, making them work is the easy part. The difficulty lies in making them work in a way that is stylish and appealing. This means taking color, animation and even the size of your widgets into account to ensure you convey the right message in your app. You'll often hear this referred to by designers as the **design language**.

This is such an important topic that Google created their own design language called **Material Design**. In this final chapter for Section II, you'll learn:

- What Material Design is

- What resources are available to learn about Material design

- How to update ListMaker so it adopts some Material Design principles

What is Material Design?

Material Design is a design language that aims to standardize how a user interacts with an app. This ranges from everything to button clicks, to widget presentation and positioning, even to animation within the app.

Before Material Design existed, there was no specific user interface an app was expected to adhere to. This was a problem for users because different apps worked in different ways, which meant users ended up with very different experiences from app to app.

All of that changed with the introduction of Material Design; Android developers finally had a set of concise UI and UX guidelines for their apps to follow. In fact, Google is so invested in Material Design it's dedicated an entire site to it at https://material.io.
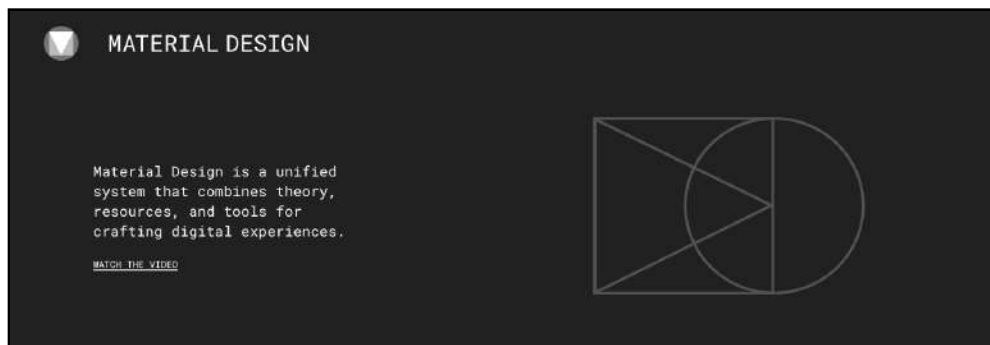
Head to the site and free to take a look around; there's plenty to look at. Once you're ready to proceed, click the **Design Guidelines** panel halfway down the page. This is where the Material Design guidelines are kept.
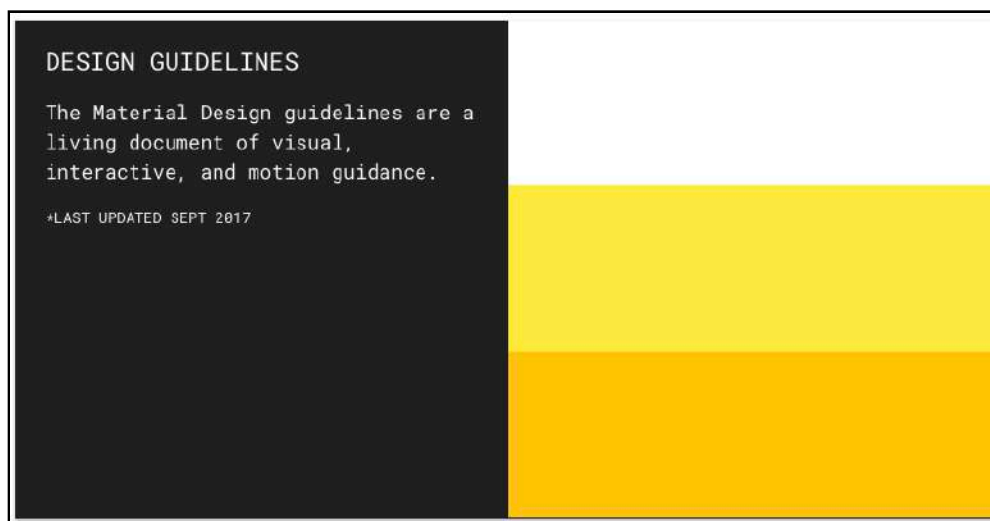
As Material Design is a living language, this page is regularly updated to represent the

latest changes. It's worth checking in regularly to see what's new.

Take a moment to look at what interests you. Once you're ready to move on, click **Style** ‣ **Color** on the left of the page. This takes you to the section of Material Design specific to color.
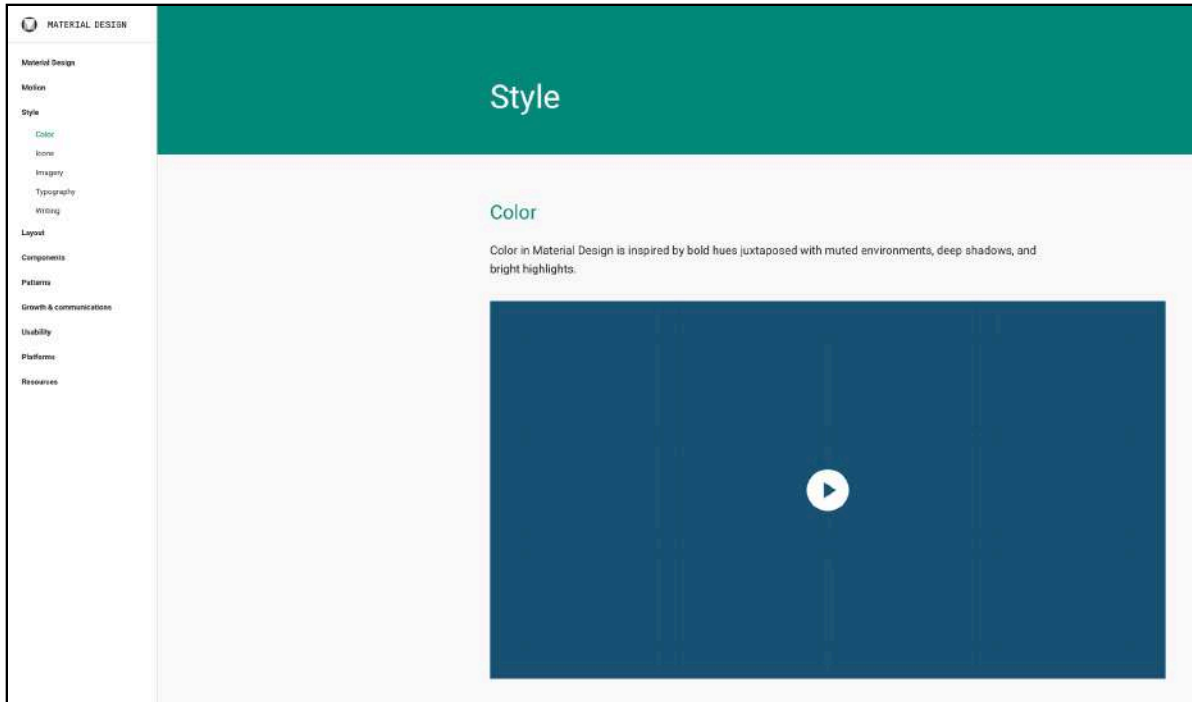
Primary and secondary colors

Color is an important tool in Material Design, as it helps draw your user's attention to areas of your app you want them to interact with.

While color is important, Material Design stresses that you shouldn't overdo it. Having too many colors in an app is distracting to a user and can confuse the purpose of widgets.

With this in mind, Material Design focuses on two color choices: **Primary** and **Secondary**.

The Primary Color is what you want to use most often across your app. Generally speaking, its should be the color of your brand. Places such as action bars and backgrounds are good spots to use primary color.

The Secondary Color is used to accent certain areas of your app, and should contrast with your Primary Color. This helps to draw attention to areas of your app that use

secondary color.

Areas of your app recommended for Secondary Color are Widgets like Buttons, Floating Action Buttons and progress bars — things you definitely want your user to notice!

It's time to bring some life to **ListMaker**. Click the **Color Tool** button in the color tool section at the top.
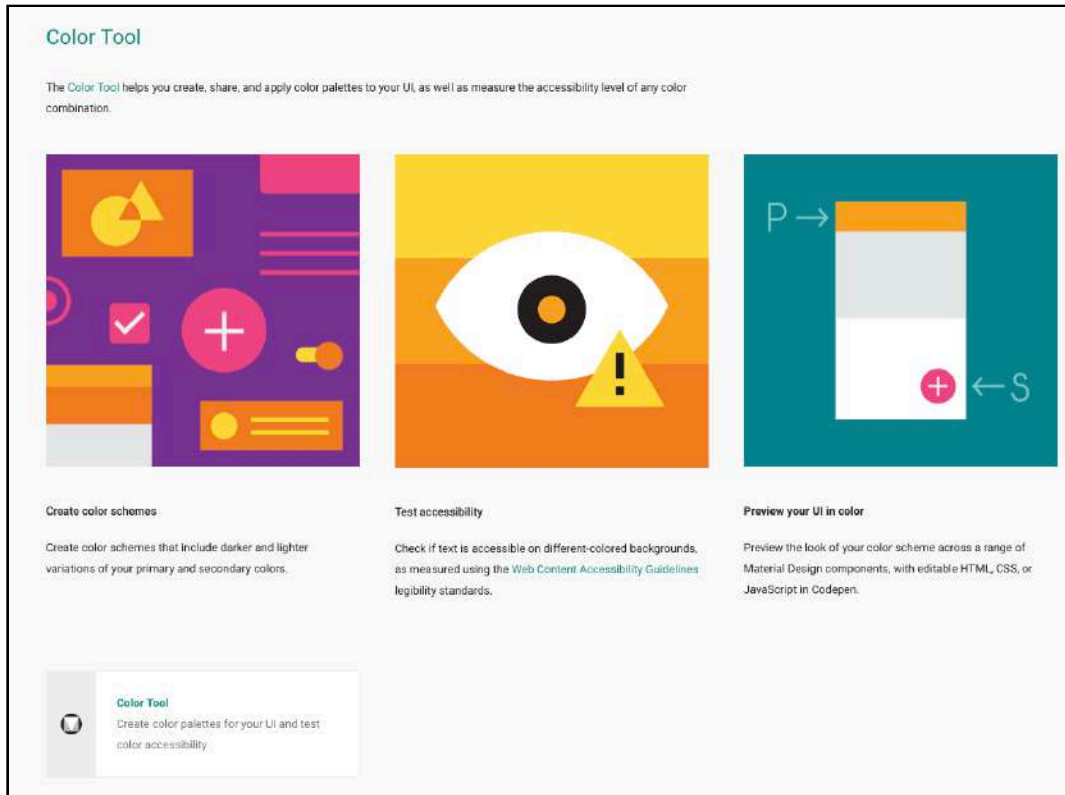
You'll be presented with a tool that generates a color scheme for your app.
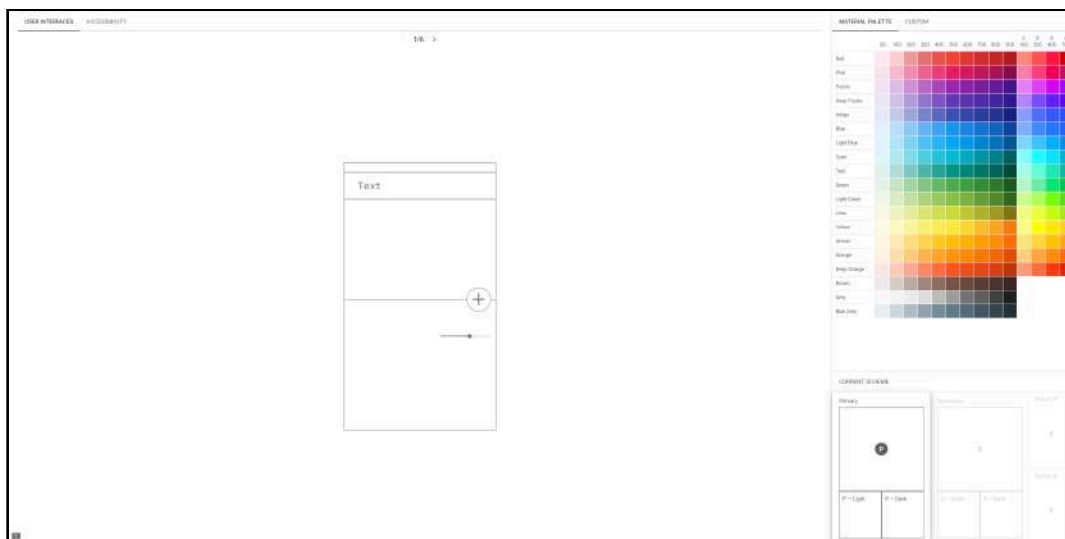
To begin, choose a Primary Color that you like, in the top right of the browser. Remember that this color is the most used color in your app, so choose wisely. We've chosen an indigo color, but you can choose something else that appeals to your sense of design.
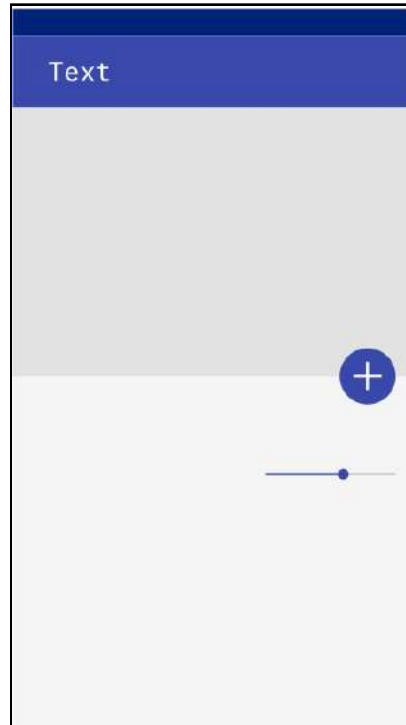
Next, in the bottom right, click the **Secondary** color scheme to inform the Color tool you want to pick a secondary color.

Pick another color you like to be your Secondary Color. Remember this should contrast with your Primary Color. We've chosen a dark red for our app.
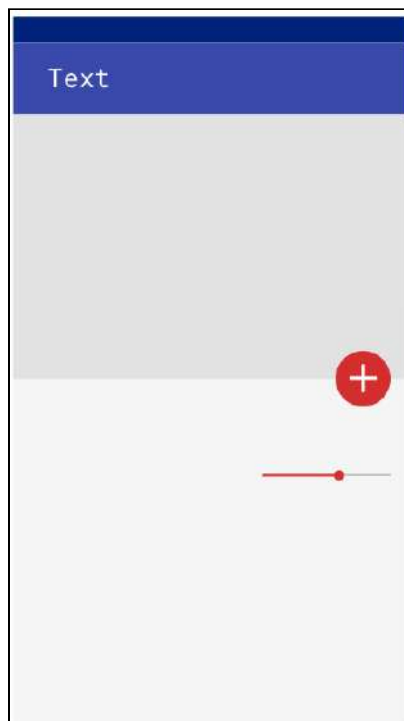
When you're happy with your selections, move your mouse cursor to the top right of the browser window and click **Export**.

A popup window appears over the export button, giving you the option to export your theme for various platforms. Click the **Android** button.
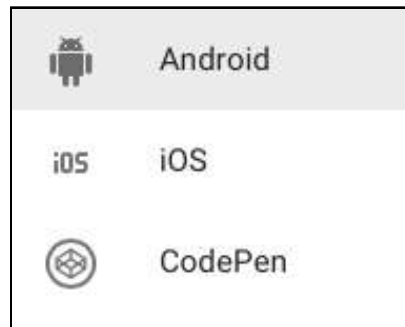
Check your download folder and you'll find **colors.xml** sitting there, ready to be imported into your project.

Open the ListMaker project in Android Studio and navigate to **colors.xml** file in **res ▸ values**. This file is where you'll declare all the colors you want to use in your app. You're going to replace the contents of this file with the new file you retrieved from the color tool.

Open up the **colors.xml** file you downloaded from the color tool, copy its contents and paste them into the **colors.xml** file in your Android Studio project.

Next, open **styles.xml**, which is found in the same directory as **colors.xml**.

```xml
<resources>

    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>

    <style name="AppTheme.NoActionBar">
        <item name="windowActionBar">false</item>
        <item name="windowNoTitle">true</item>
    </style>

    <style name="AppTheme.AppBarOverlay" parent="ThemeOverlay.AppCompat.Dark.ActionBar" />

    <style name="AppTheme.PopupOverlay" parent="ThemeOverlay.AppCompat.Light" />

</resources>
```

This file is responsible for declaring **Themes** for your app. A theme is a set of attributes that are grouped together. You can create many themes for your app to avoid declaring the same attributes over and over again in your layouts and widgets.

You might be seeing errors with this file, because the colors used in the **AppTheme** theme no longer exist. Within the AppTheme, adjust the `colorPrimary`, `colorPrimaryDark` and `colorAccent` entries like so:

```
<item name="colorPrimary">@color/primaryColor</item>
<item name="colorPrimaryDark">@color/primaryDarkColor</item>
<item name="colorAccent">@color/secondaryColor</item>
```

This tells your app which colors will be defined as the primary and secondary colors. Time for the big reveal! Run your app and check out the difference in color:

That Floating Action Button looks nice and bold. Tap it and notice the accent color change in the dialog:
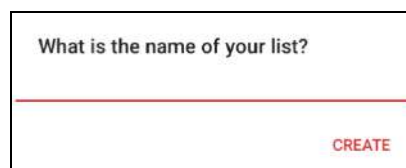
Very nice. You've just updated your app to use some Material Design colors of your own!

Card views

While reading the Material Design site, you might notice that it often emphasizes the use of **Cards**. Cards are designed as a gateway to more information.
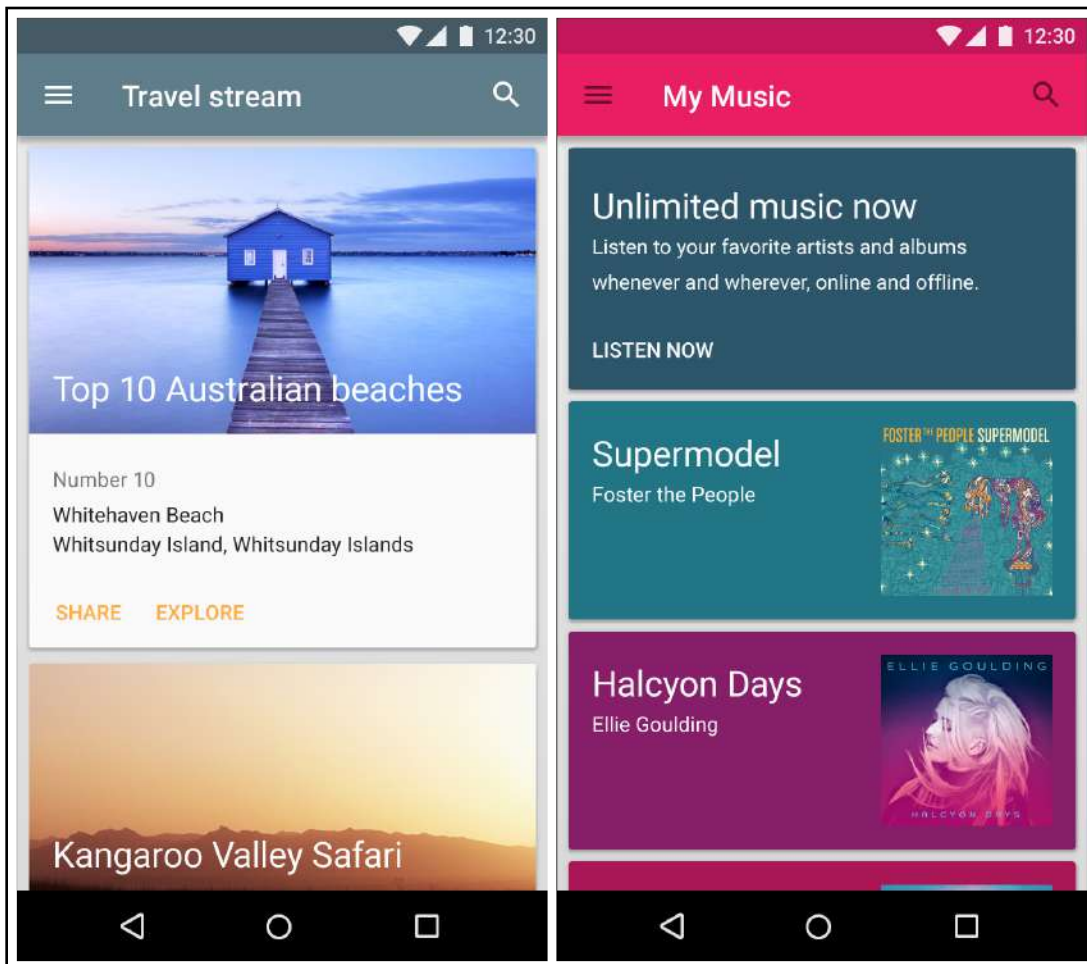
Example of apps using Cards

This sounds like a great way to visually inform users that a list contains a number of tasks in ListMaker.

> **Note**: You can read more about cards on the Material Design website at: https://material.io/guidelines/components/cards.html.

To use a Card in your app, you need to declare a new dependency in your app. Open up **build.gradle (Module: app)**, and to the dependencies block, add the following line:

```
implementation 'com.android.support:cardview-v7:26.1.0'
```

Click the **Sync Now** button that appeared when you changed the Gradle script. This
rebuilds your project, pulling in any new dependencies you added from the Internet to
use in your project.

Gradle files have changed since last project sync. A project sync may be necessary for the IDE to work properly.   Sync Now

With the Cards dependency added to your project, it's time to tell your list `ViewHolder`
to use cards. Open up the **list_selection_view_holder.xml** layout in **res ▸ layout**,
making sure you have the **Text** layout view open. Update the XML layout so it looks like
the following:

```xml
<android.support.v7.widget.CardView xmlns:android="http://
schemas.android.com/apk/res/android"
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="4dp"
    android:layout_marginLeft="4dp"
    android:layout_marginRight="4dp"
    card_view:cardCornerRadius="2dp">

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal">

        <TextView
            android:id="@+id/itemNumber"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="16dp" />

        <TextView
            android:id="@+id/itemString"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="16dp" />
    </LinearLayout>
</android.support.v7.widget.CardView>
```

The `LinearLayout` and its containing TextView widgets stay the same. The big change is
that these components are now wrapped up in a **CardView**. Cards behave similar to
other layout widgets, with a few additional properties. To access the properties, you first
need to assign a namespace to access them, this is done via `xmlns:card_view`.

You then use the namespace at the end of the open CardView tag, via
`cardCornerRadius`. This sets the rounding of each corner of the card. The Material
Design guidelines recommend a rounding of 2 density pixels (dp), which you've used
here.

It's also worth noting that the `CardView` has had its margins pushed 4dp from the left,

right and bottom. This is to ensure that it doesn't hit the edge of the screen or isn't clipped by a card beneath it, which would obscure the `CardView`.

Run your app and you'll see that your collection of lists are looking more appealing.
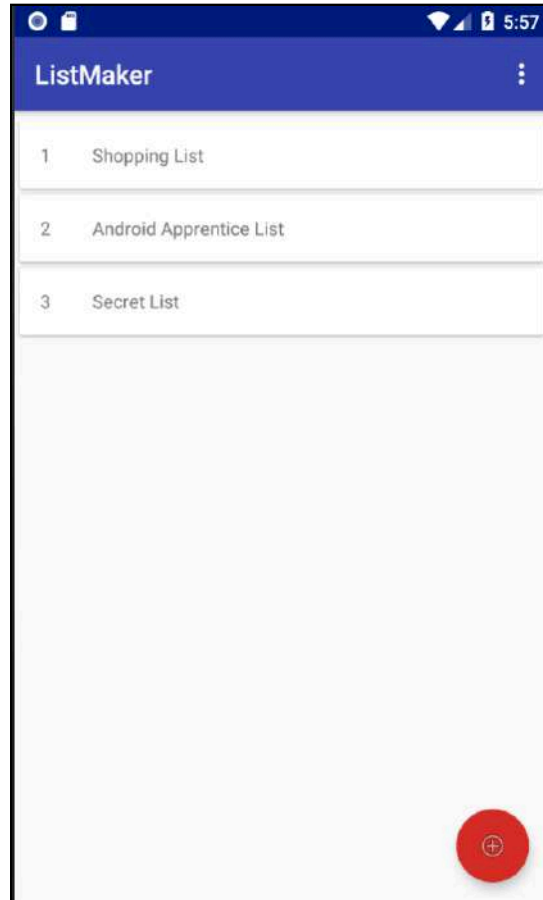
Where to go from here?

The visual flair you've added to ListMaker by implementing Material Design was well worth it! You've only had a peek into the benefits Material Design brings to an app. In future apps, it's worth reading up on the Material Design guidelines and finding ways to incorporate it into your app, ensuring you're providing your users with a great experience.

Keep checking https://material.io to make sure you stay up-to-date!

Section III: Creating Map-Based Apps

In this section, you'll build **PlaceBook**, an app that lets you bookmark your favorite places and save some notes about each place.

Chapter 13: Creating a Map-Based App

Chapter 14: User Location and Permissions

Chapter 15: Google Places

Chapter 16: Saving bookmarks with Room

Chapter 17: Detail Activity

Chapter 18: Navigation and Photos

Chapter 19: Finishing Touches

Chapter 13: Creating a Map-Based App

By Tom Blankenship

Have you ever been on a road trip and wanted to makes notes about places you've visited? Or needed to warn your future self about some heartburn-inducing greasy food from a roadside diner? Or have you ever wanted to keep reminders about the best menu items at your favorite local restaurants?

If you answered "yes" to any of those question, then you're in luck! You're about to build **PlaceBook**, an app that meets all of those needs by letting you bookmark and make notes using a map-based interface.

Even if you didn't answer "yes", that's the app you're going to be building, so roll up your sleeves and dive in!

Getting started

While building out PlaceBook, you'll use familiar techniques from the previous sections and learn about several new Android APIs. You'll use the **Google Maps API** to display a map, track the user's location, and add custom markers. You'll use the **Google Places API** to display place information and search for nearby places. You'll use the **Room Persistence Library** to store data. You'll also learn about **Implicit Intents** for sharing your data to other apps.

There's a lot of ground to cover, so let's get started! The final product will look like this:
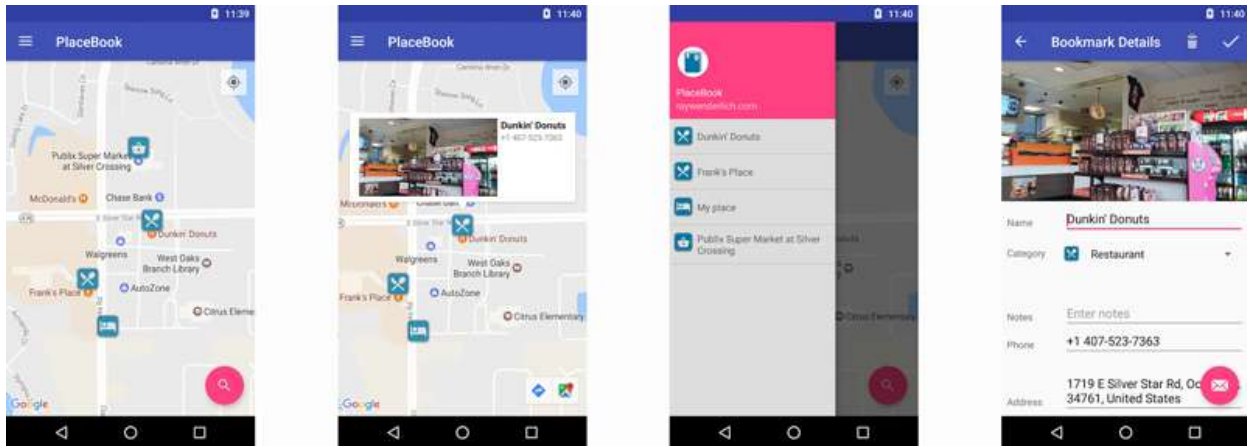
About PlaceBook

PlaceBook starts by displaying a **Google Map** centered around your current location. The map will display common places, and allow you to bookmark them. You can display details for bookmarked places and edit the place data and photo.

The navigation drawer on the left will display all of your bookmarks and tapping on one will zoom the map to that location. You can use the search icon to find nearby places and jump directly to them on the map.

Making a plan

With a large number of features to implement, it's easiest to think about them in bite-sized pieces, slowly building up to the finished product. The steps will include the following:

1. First, you'll create a basic map to display the user's current location. You'll get familiar with the Google Maps API and the Fused Location Provider.

2. You'll then allow the user to select Places on the map and display information about the place. You'll learn how to load detailed information about a Place using the Google Places API.

3. You'll add the basic bookmarking ability by using **Room** to store places in a local database and add map markers to show the user's bookmarked locations.

4. Next, you'll add a details screen to let the user edit their bookmark details, delete bookmarks, and replace the default bookmark photo with one from the camera or photo gallery.

5. You'll add a navigation drawer to let the user jump directly to any saved bookmark.

6. You'll then use the Google Places autocomplete service to let the user search for nearby locations.

7. You'll add the ability to long tap any location on the map to add a bookmark that doesn't already have an existing place on the map.

8. Finally, you'll add some finishing touches to make the app look better with a custom color theme and icons.

Location service components

The Android SDK provides three main components related to location and mapping:

- **Framework Location APIs**: Known collectively as the **location framework**, this framework has been around the longest and is the traditional means for getting the user's current location. However, it's also the most difficult one to use.

- **Google Maps API**: The Google Maps API makes it easy to display interactive maps within your app. It provides a lot of functionality out of the box, including everything needed to display detailed map data and respond to user gestures. You'll cover this API in detail in the next chapter.

- **Google Play Services location APIs**: The Google Play services location APIs are built on top of the core location framework and alleviate much of the pain involved with tracking user location. You'll be using the **FusedLocationProviderApi** component of this API in the book.

Map wizard walk-through

To save time, you'll use the Maps Activity project template to generate an app with a single activity that displays a map that looks like the following:

To begin, launch Android studio and select **Start a new Android Studio Project**.
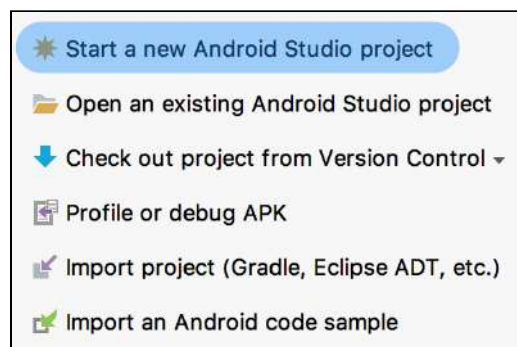
Fill out the **Create Android Project** dialog like this:

- Application name: PlaceBook

- Company domain: raywenderlich.com

- Project location: select a directory for the project files

- Include C++ support: unchecked

- Include Kotlin support: checked
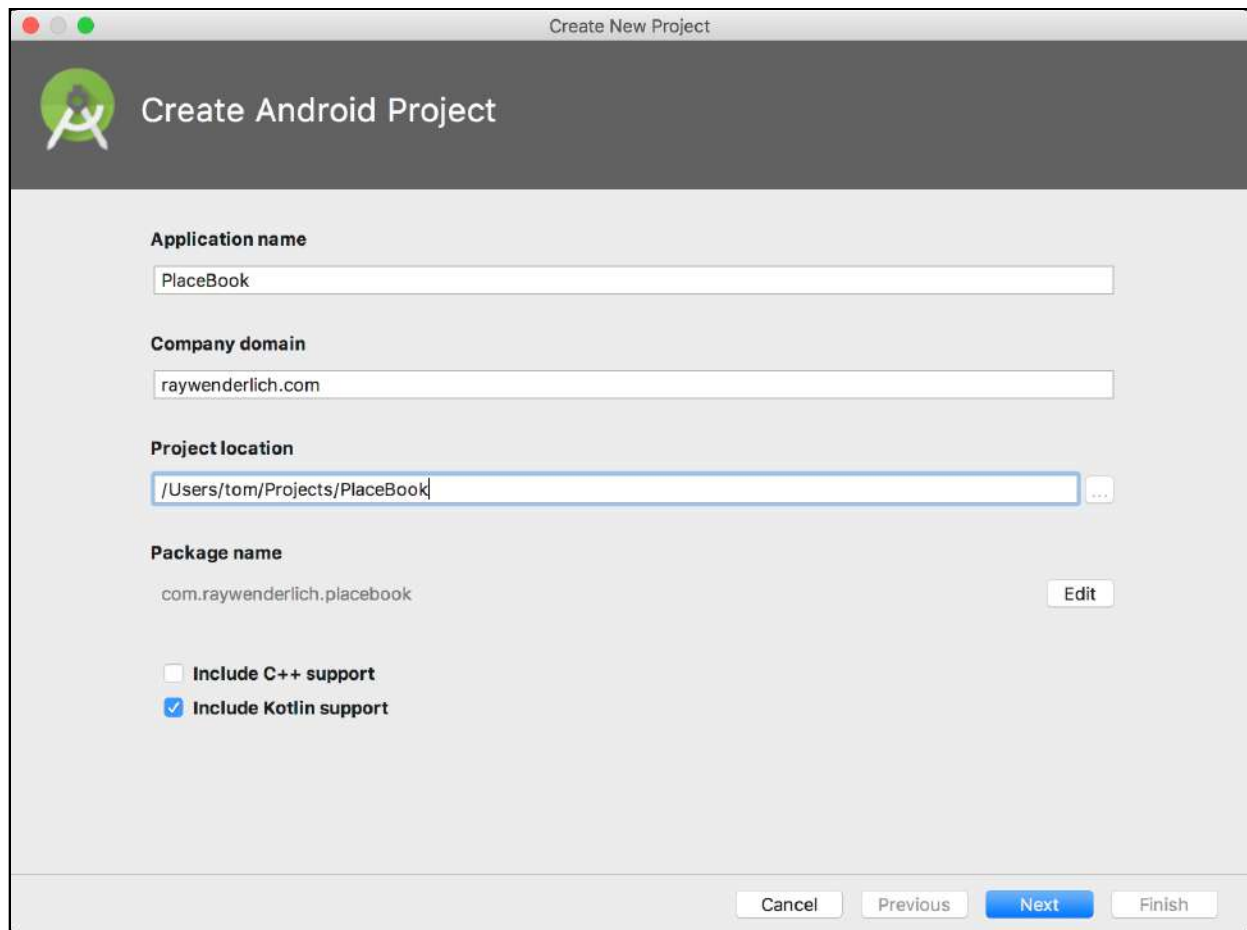
Click **Next**.

Fill out the **Target Android Devices** dialog like this:

- Phone and Tablet: checked

- Minimum SDK: API 19

- Leave everything else unchecked.

As with all other apps in the book, PlaceBook will run on devices back to API level 19.
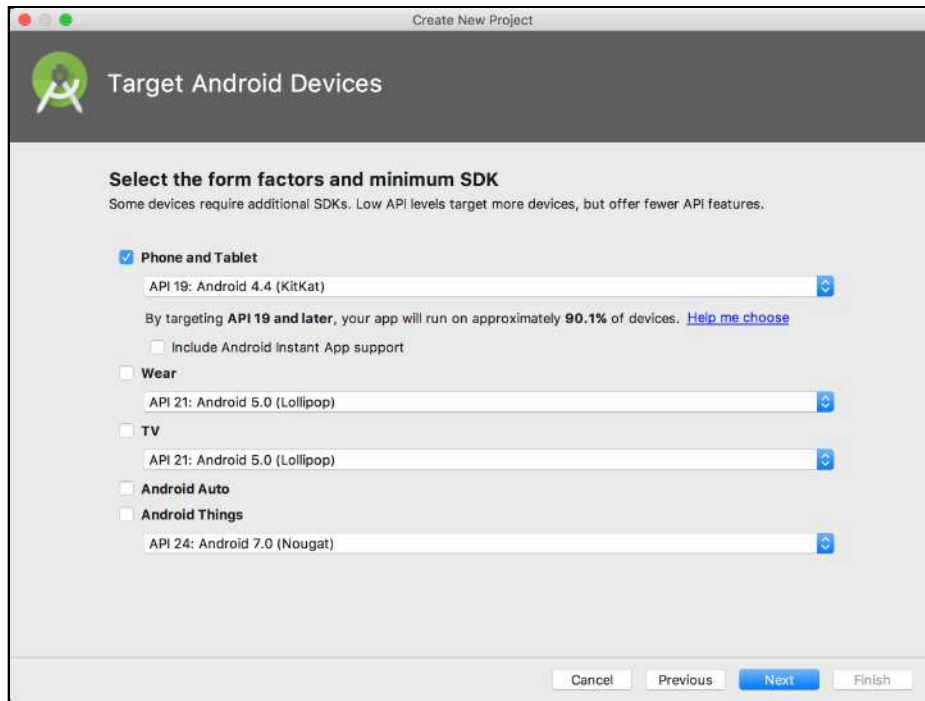
Click **Next**.

Select the **Google Maps Activity** option on the **Add an Activity to Mobile** dialog and select **Next**.
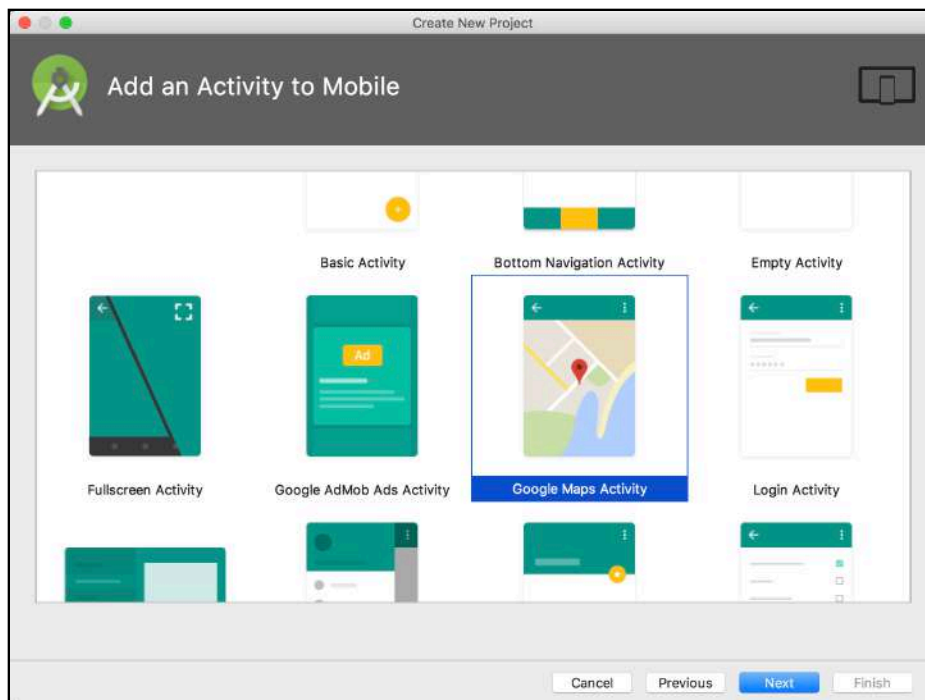
In the **Configure Activity** dialog, update the title to "PlaceBook", leave the other options at the defaults, and select **Finish**.
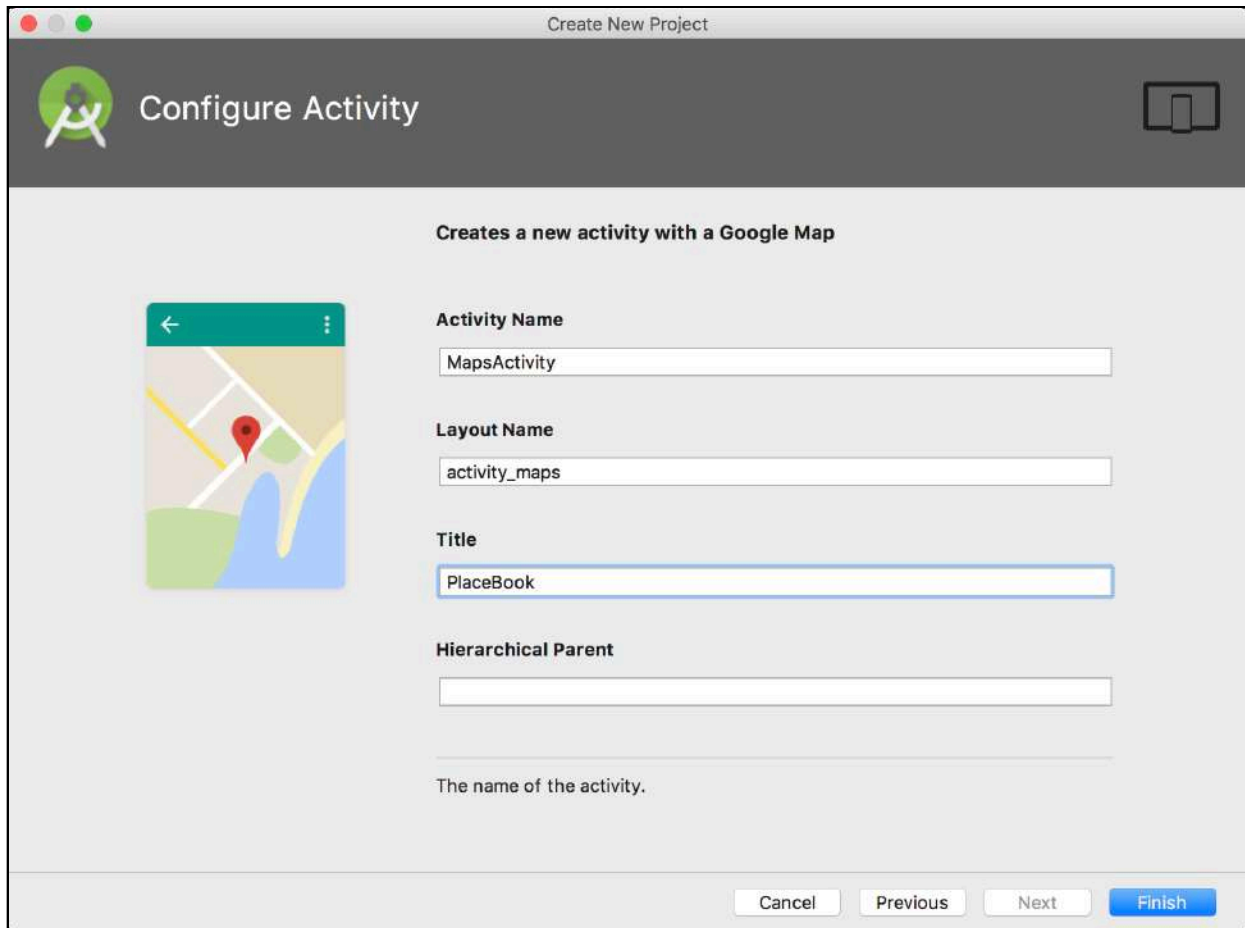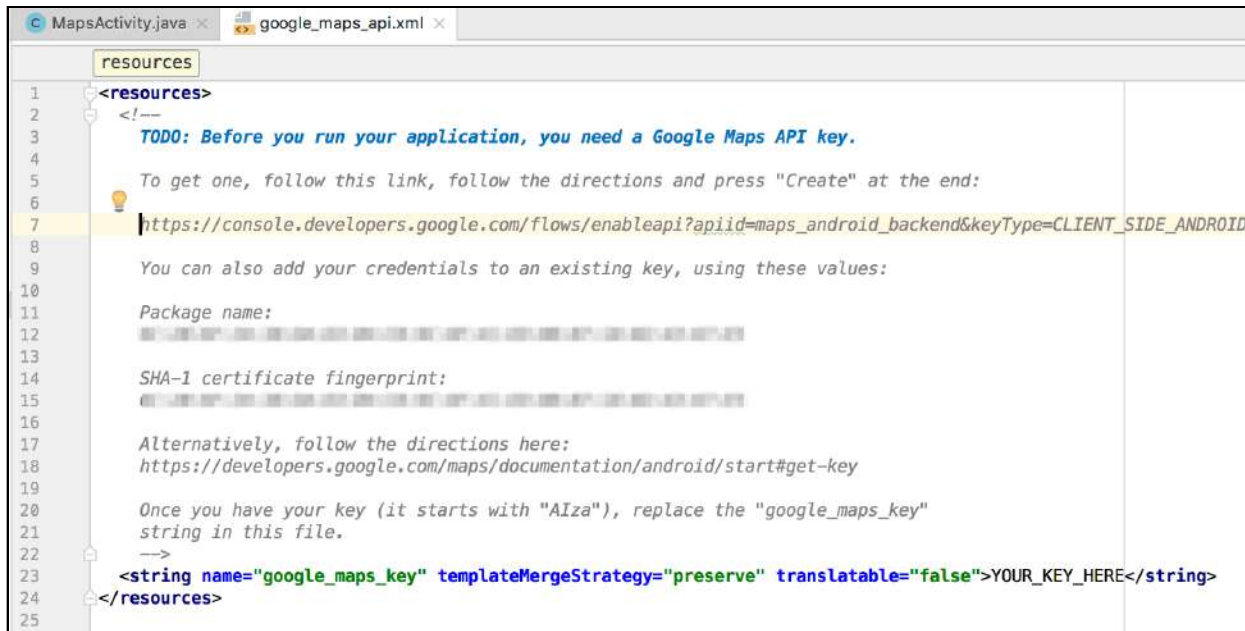
Android Studio will now automagically generate your new project and perform an initial build. If all goes as planned, you will be left viewing the **google_maps_api.xml** file.

Google Maps API key

Before your app will work, you have to generate an API key using the Google Developer console. You'll need a Google account to sign in to the developer console. Don't worry, use of the developer console is free!

The Google Maps API communicates with the Google Map servers and will only work if a valid key is provided by the app. Android Studio generates the **google_maps_api.xml** file to make your life easier. It provides important information that will help you create the Google Maps API key.

The easiest way to create your own API key is to use the link at the top of **google_maps_api.xml**, shown here and highlighted in yellow:

Take note of the **Package Name** and **SHA-1 Fingerprint** values. These are the two requirements for generating a key. The link is just an easy way to pass those values to the key generation page in the Google Developer Console.

Package Name is straightforward: It's the package name you used when creating the project. The SHA-1 Fingerprint may look a little odd if you aren't familiar with SHA-1. SHA-1 is a method for generated secure hashes. Just like a real fingerprint uniquely identifies you, each SHA-1 fingerprint uniquely identifies a set of bytes.

The fingerprint in **google_maps_api.xml** is a SHA-1 hash of the certificate from your debug keystore file. A keystore file contains everything needed to digitally sign an Android application (APK) file. During development, your apps are signed with a debug keystore file. When delivering apps to the Play Store, you sign with a release keystore file.

The debug keystore file is automatically generated when you first install Android Studio and is shared among all of your projects. Using a release keystore will be covered in detail in Chapter 30, "Preparing for Release".

If you've worked with Google Maps before, you may have already generated a Google Maps API key. You can add the Package Name and SHA-1 Fingerprint to an existing key instead of generating a new one.

There are actually two versions of **google_maps_api.xml** in your project. One version is used only when building the debug version, and the other only for the release version.

If you are using the **Android** view in the **Project View**, you'll only see one version of the file in the **app/res/values** folder, but you'll see **(debug)** after the filename.

To see both versions, switch over to the Project Files view by selecting the **Project Files** tab in the **Project View** window. Open up the **app/app/src/debug/res/values and app/app/src/release/res/values** folders and you'll notice that there is a **google_maps_api.xml** file in each one. By placing files in these build-specific folders, Android Studio will apply them separately to debug or release builds as appropriate.
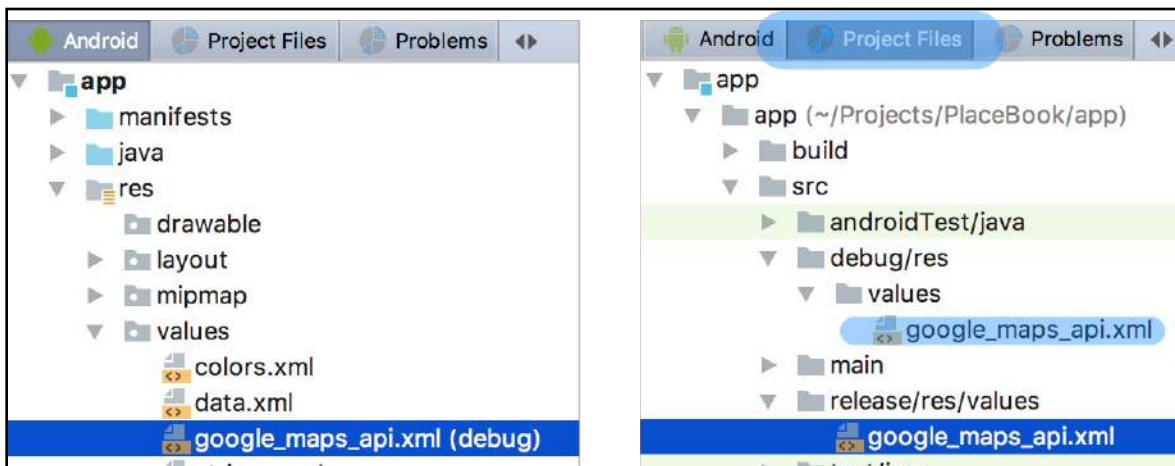
Follow the link provided in the **google_maps_api.xml** file and you should see the following page after signing in to your Google account:
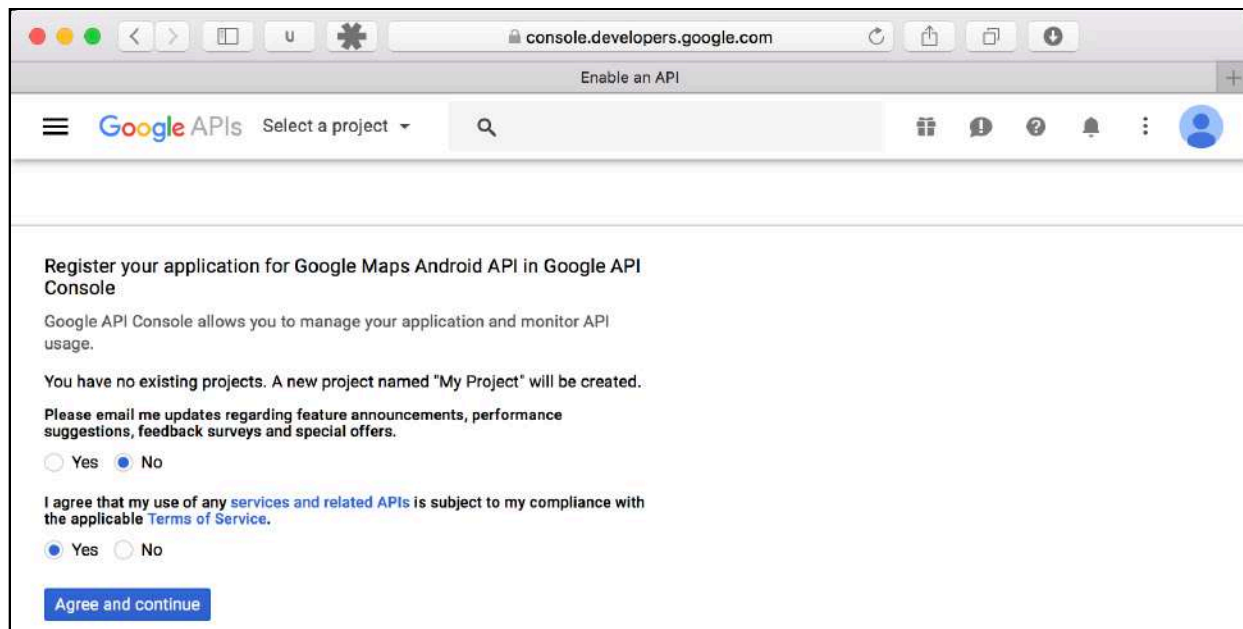
Click **Agree and continue**, and you'll come to a page displaying **The API is enabled**.
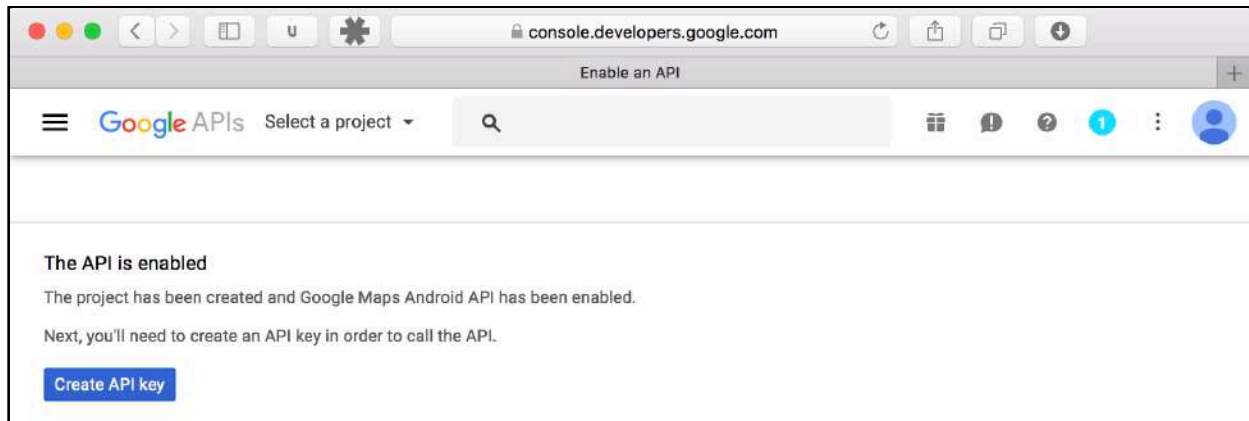
This created a project behind the scenes in your Google Developer console and enabled the Google Maps API for you. In a later chapter, you'll learn how to manually enable APIs. For now just remember which Google account you created this project with so you can later edit it further.

Click **Create API key** button and you'll see the **API key created** dialog containing your shiny new key:
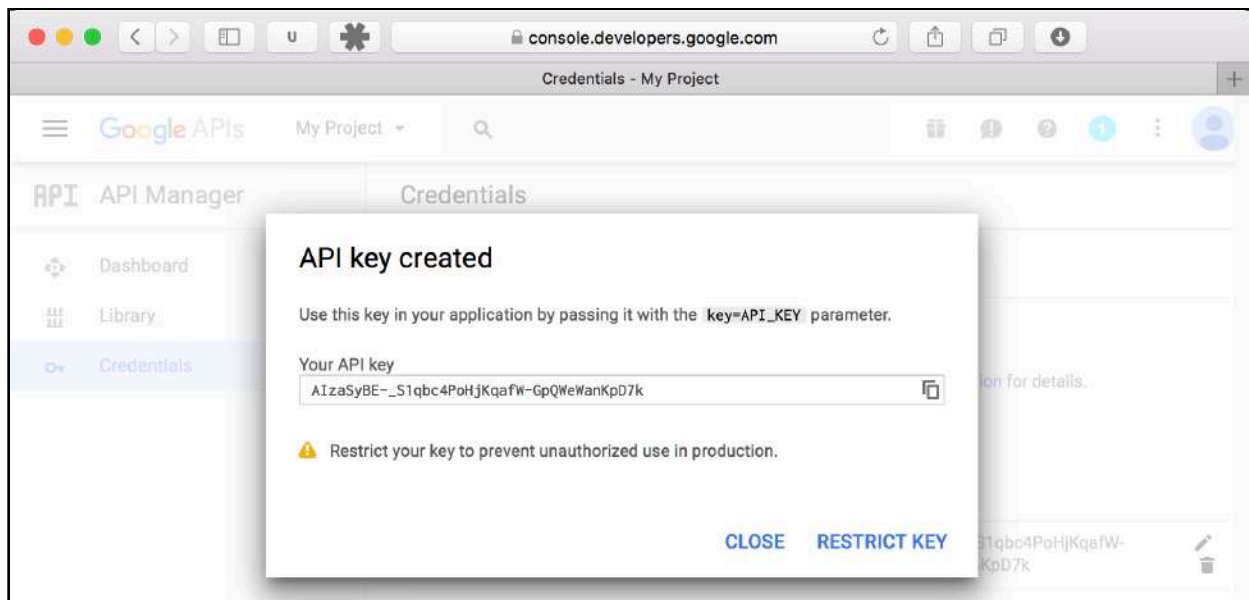
> **Note**: Don't worry about the **RESTRICT KEY** option. The key will already be restricted to your developer certificate SHA-1 fingerprint and package name. When you release your app to the public, you can place further restrictions around this to prevent unauthorized use.

Copy the key from this dialog and paste it into **google_maps_api.xml** where it says
YOUR_KEY_HERE. The resulting file should look something like this, but with your key
instead:

```xml
<string name="google_maps_key"
        templateMergeStrategy="preserve"
        translatable="false">
        AIza5sD-_G2dq7PjafW-Ad4pKpU5a</string>
```

Getting the keystore fingerprint

Although Android Studio has conveniently placed your debug keystore fingerprint in
the XML file, it's helpful to know how to get the fingerprint yourself if you ever need to
regenerate it. The following instructions will work for debug builds; getting the SHA1
key for release builds will be covered in Section VI, "Submitting Your App".

First locate your keystore file.

- On macOS, it will be found in **~/.android/**.

- On Windows, you'll find it in **C:\Users\your_user_name\.android**.

On macOS, run the following command:

```
keytool -list -v -keystore ~/.android/debug.keystore -alias
androiddebugkey -storepass android -keypass android
```

On Windows, run the following command:

```
keytool -list -v -keystore "%USERPROFILE%\.android\debug.keystore" -alias
androiddebugkey -storepass android -keypass android.
```

This should produce output similar to the following:

```
Alias name: androiddebugkey
Creation date: Jan 01, 2013
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:  18:5E:95:D0:A6:86:89:BC:A8:70:BA:34:FF:6A:AC:A4
     SHA1: A5:1F:AC:74:D3:21:E1:43:07:71:9B:62:90:AF:A1:66:6E:44:5D:46
     Signature algorithm name: SHA1withRSA
     Version: 3
```

The SHA1 key you see should match what is already in the XML file.

Maps and the emulator

That's all you should need if you're installing on a real device, but if you're using an emulator, then things can get a little more complicated.

A basic requirement of the Google Maps API is that your device must have the Google APIs installed. Not all emulators include this by default. If you don't have one already, follow the following steps to create an emulator with API Level 19 or newer that includes the Google APIs.

Select **Tools ▸ Android ▸ SDK Manager**. Under the SDK Platforms tab, select the **Show Package Details** option.

Select a version from Android with API level 19 or newer. Make sure **Android SDK Platform** and **Google APIs Intel x86 Atom_64 System Image** are selected. If **Google APIs** is an option, it should be selected also. The following shows Android 8.0 (Oreo) with the necessary items selected.

Click **OK** to install the platform files.
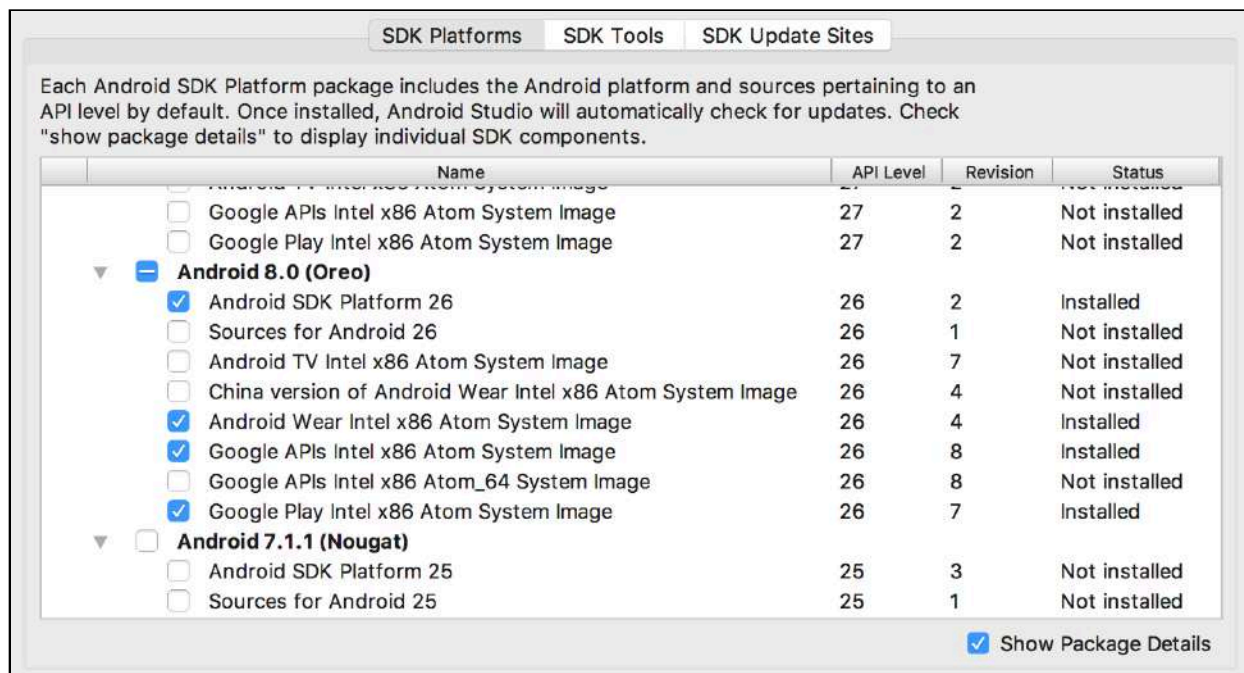
Once the installation has finished, select **Tools** ‣ **Android** ‣ **AVD Manager** and then click the **Create Virtual Device** button.
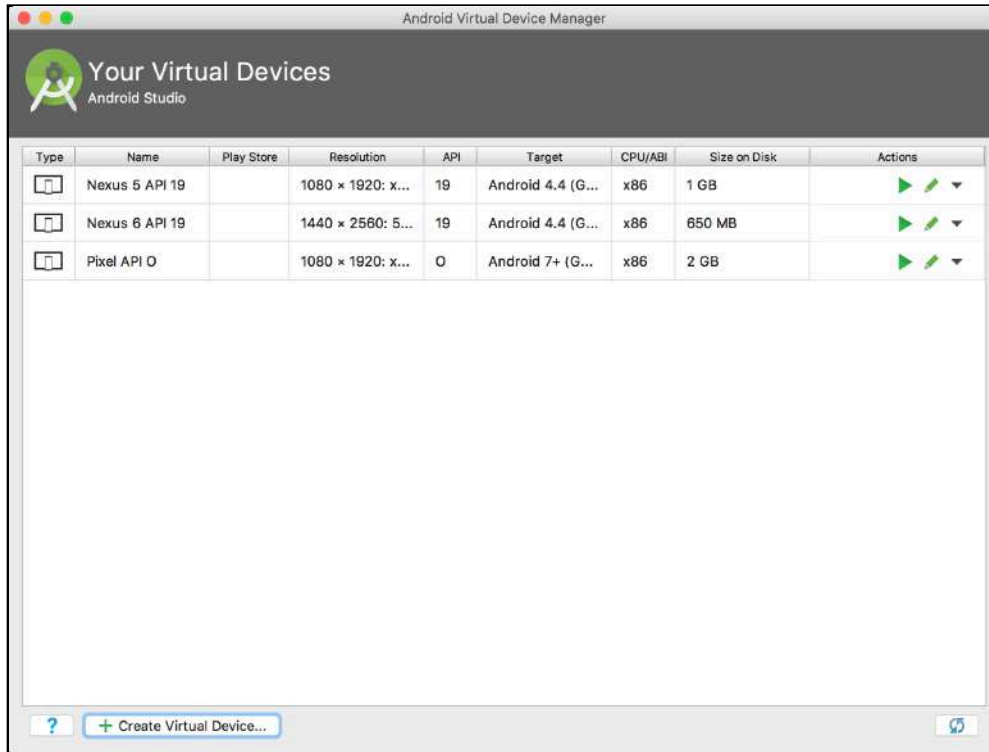
Select your preferred device and click **Next**. For demonstration purposes, shown next is an emulator set up for a Pixel device.
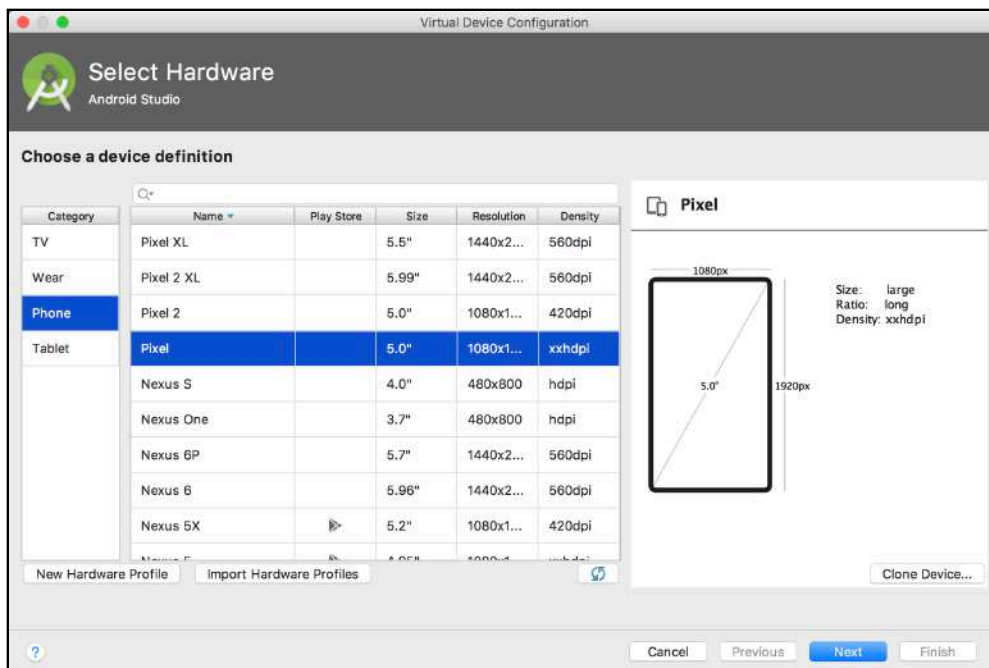
The **Recommended** tab should display a choice that matches the SDK platform files you downloaded in the previous step. Make sure the option selected is one with the Google APIs option as shown here. If you don't see one on the Recommended tab, then try the **x86 images** tab.

Click **Next**.
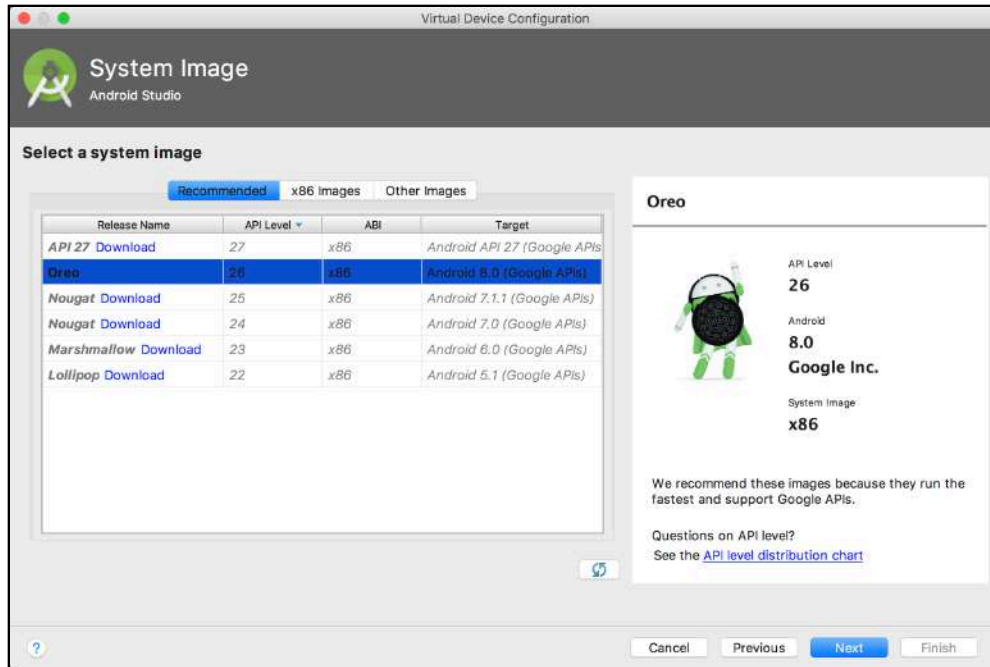
Leave the default settings on the configuration screen and click **Finish**.
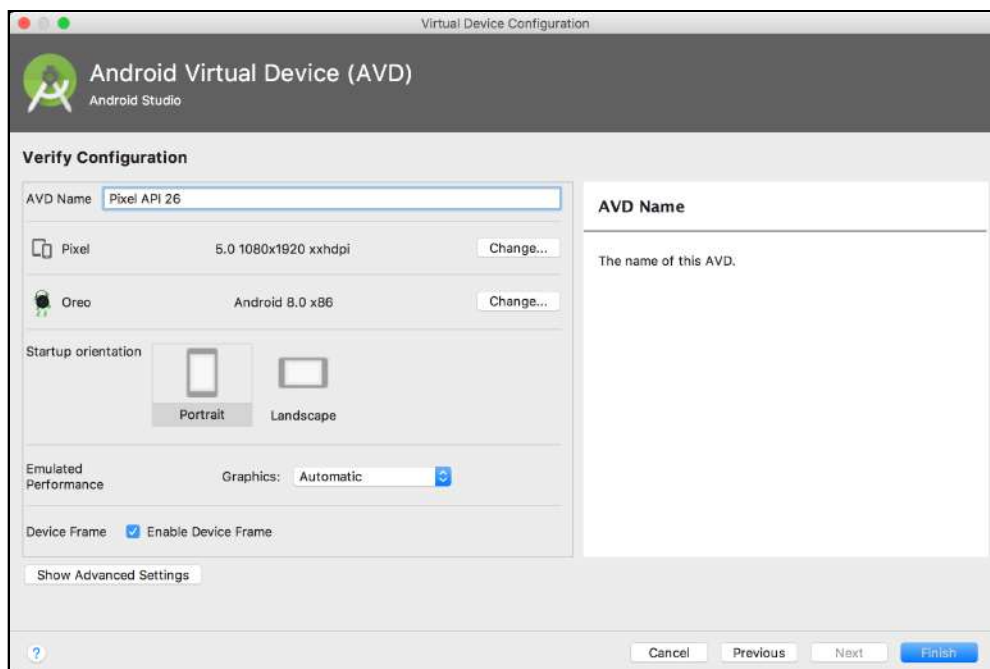
You should now see the new virtual device shown along with any others you may have created before. Make sure to use this virtual device when launching the app.
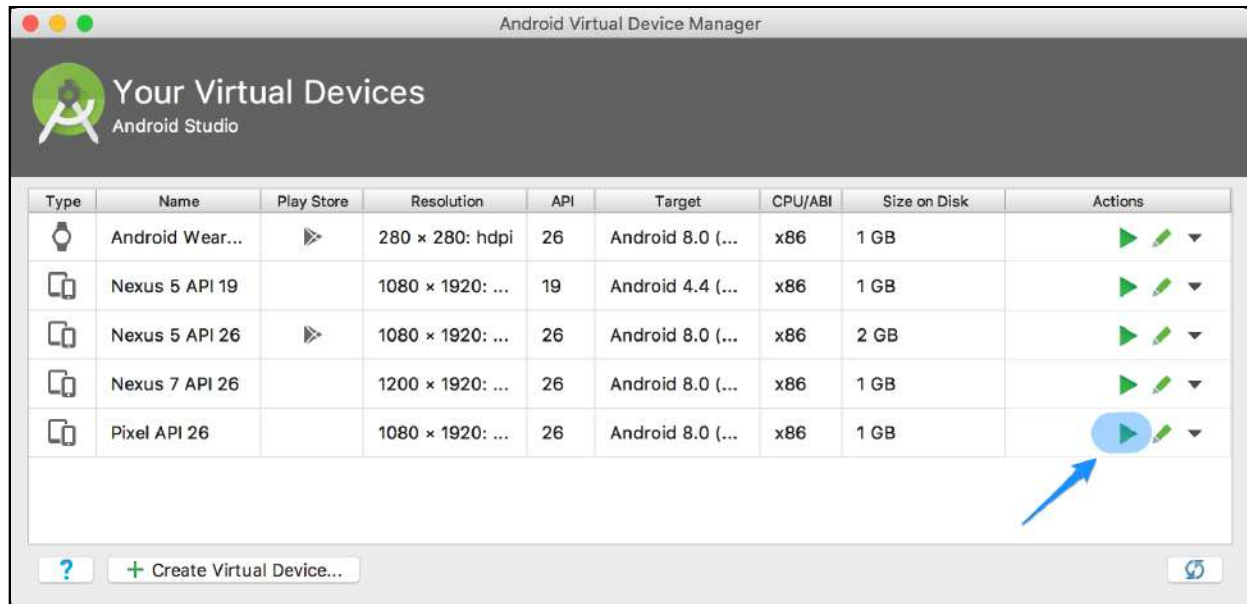
Running the app

Launch the app from Android Studio.

If your key is valid, you should see a map on the screen. If you see a blank screen, check Logcat to see if there are any error messages.

If you see an error message in Logcat that looks like the one shown here, then double check that you have pasted the correct key in **google_maps_api.xml**:

```
Google Maps Android API: Authorization failure.  Please see https://
developers.google.com/maps/documentation/android-api/start for how to
correctly set up the map.
Google Maps Android API: In the Google Developer Console (https://
console.developers.google.com)
Ensure that the "Google Maps Android API v2" is enabled.
Ensure that the following Android Key exists:
API Key: YOUR_KEY_HERE
    Android Application (<cert_fingerprint>;<package_name>): 6A:27:6F:
34:38:DA:D3:04:C8:9C:8F:
41:ED:BB:B7:18:02:77:67:D2;com.raywenderlich.placebook
```

Look at the key shown after **API Key:** in Logcat and make sure it matches the one you received when you created your key earlier.

Once you have the correct key, you should see a map with a marker placed over Sydney, Australia:

Congratulations! You're off to a great start with your maps app. Pan and zoom around a bit. There's not much more you can do at this point. That will change soon enough!

Before moving on, take a moment to review the files Android Studio created for you.

Project dependencies

There are two dependencies required before you can use maps in your app. To find the first one, open **build.gradle** from your application module folder. In the dependencies section, you should see the following line.

```
implementation 'com.google.android.gms:play-services-maps:11.8.0'
```

This instructs the Gradle build system to include the Maps API in your build and is required to use maps.

You may be wondering, "How do I know which version of the library to include?"

Good question! There are at least three ways you can find the latest version:

1. Go to https://developers.google.com/android/guides/setup. Scroll down to see the list of APIS. This list is dynamically generated and reflects the most recent version of each API.

Table 1. Individual APIs and corresponding `build.gradle` descriptions.

API	Description in `build.gradle`
Google+	com.google.android.gms:play-services-plus:11.8.0
Google Account Login	com.google.android.gms:play-services-auth:11.8.0
Google Actions, Base Client Library	com.google.android.gms:play-services-base:11.8.0
Google Sign In	com.google.android.gms:play-services-identity:11.8.0
Google Analytics	com.google.android.gms:play-services-analytics:11.8.0
Google Awareness	com.google.android.gms:play-services-awareness:11.8.0

2. Go to https://developers.google.com/android/guides/releases. Note the latest release of Google Play services.

December 18, 2017 - Version 11.8.0

Highlights from the Google Play services 11.8.0 release:

- **Auth**
  - The `WorkAccountApi` class is now deprecated
    instead.
- **Cast**
  - Made the following changes to the `UIMediaCo`

3. Select **File ‣ Project Structure**. Select **app** under Modules, then select the **Dependencies** tab, click the + button, and select **1 Library Dependency**.

Ads
Authentication
Notifications
—— Modules ——
app

m com.android.support:appcompat-v7:26.1.0
m com.google.android.gms:play-services-maps:11.8.0
m junit:junit:4.12
m com.android.support.test:runner:1.0.1
m com.android.support.test.espresso:espresso-core:3.0.1

+ — ▲ ▼
m 1 Library dependency
2 Jar dependency
3 Module dependency
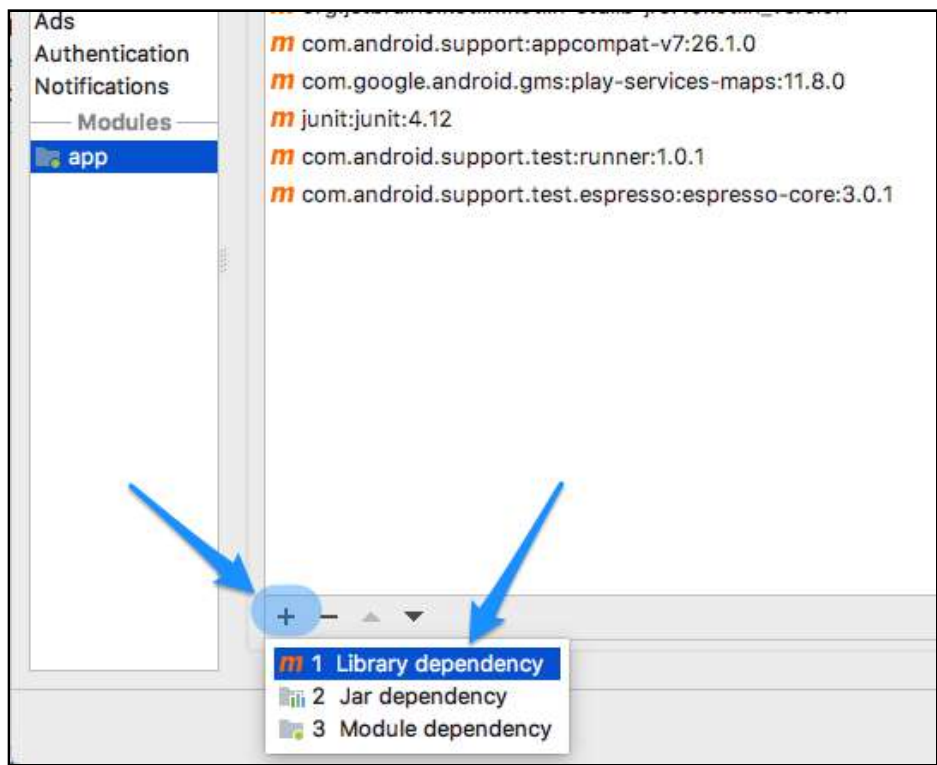
Type **play-services-maps** and press **Enter**.
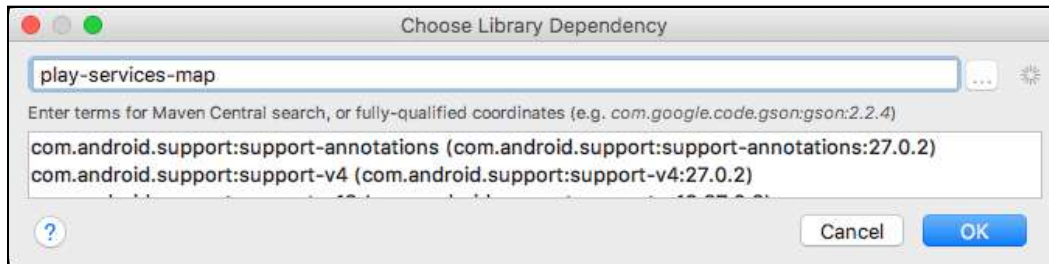
You should be shown the latest available version:
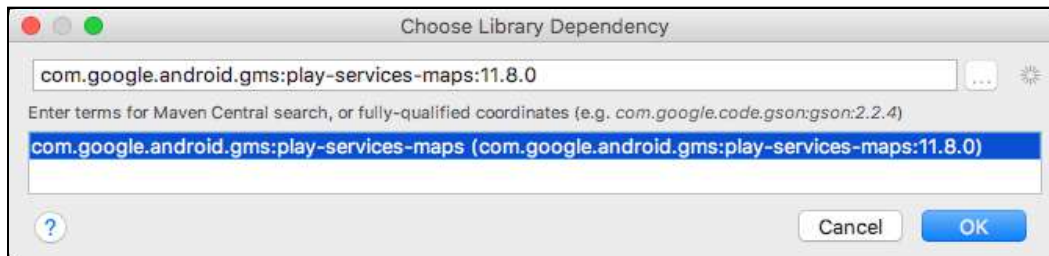
Note: In Google Play services versions prior to 6.5, all of the Play services were included in one package named **play-services**. This would often lead to problems with creating APK files that exceeded the 65 KB method limit.

Now, you can choose just the subset of play services required for your app, such as the Google Maps API.

The manifest

First, from **app/manifests**, open up **AndroidManifest.xml**. It should look like this, with your actual API key displayed in place of `@string/google_maps_key`:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.raywenderlich.placebook"
          xmlns:android="http://schemas.android.com/apk/res/android">
  <!--
        The ACCESS_COARSE/FINE_LOCATION permissions are not
        required to use Google Maps Android API v2, but you
        must specify either coarse or fine location permissions
        for the 'MyLocation' functionality.
    -->
  <uses-permission
      android:name="android.permission.ACCESS_FINE_LOCATION"/>

  <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">
        <!--
                 The API key for Google Maps-based APIs is defined
                 as a string resource. (See the file
                 "res/values/google_maps_api.xml"). Note that the
                 API key is linked to the encryption key used to
                 sign the APK.
            -->
        <meta-data
            android:name="com.google.android.geo.API_KEY"
            android:value="@string/google_maps_key"/>

        <activity
            android:name=".MapsActivity"
            android:label="@string/title_activity_maps">
          <intent-filter>
            <action android:name="android.intent.action.MAIN"/>

            <category
                 android:name="android.intent.category.LAUNCHER"/>
          </intent-filter>
        </activity>
      </application>

    </manifest>
```

This is a fairly standard manifest file, and most of it should look familiar from previous sections. Look at the `uses-permission` element. As the comment in the file says, the `ACCESS_FINE_LOCATION` permission is not required to show the map. You *could* remove this line and your app would continue to run just fine, but you're going to need this later. In the next chapter, you'll cover permissions in detail and discover why this permission is needed when obtaining the user's location.

The `meta-data` tag under the Application section is where Android Studio looks for your API key when signing the APK. From the raw source shown above, you can see the key is pulling from the string resource you defined in **google_maps_api.xml**. When viewing the file in Android Studio, it will show you the actual key.

The activity and layout

Open **MapsActivity.kt**. This is the startup activity created from the Maps template. Note that it inherits from the **AppCompatActivity** class and the **OnMapReadyCallback** interface.

```
  class MapsActivity : AppCompatActivity(), OnMapReadyCallback {
```

Map display options

There are two ways to display a map in your app:

1. **As a fragment using the SupportMapFragment class**. SupportMapFragment is a subclass of Fragment and is the typical choice unless you need very fine-grained control of the map. You can also use `MapFragment`, but using `SupportMapFragment` provides the best support for backwards compatibility.

   Remember how you used fragments to host your main UI in the CheckList app? The MapsActivity template does the same thing by hosting the `SupportMapFragment` within your main activity.

   `SupportMapFragment` acts as a reusable component that you can easily plug into any activity. It handles all aspects of displaying the map and gives you access to the **GoogleMap** object.

2. **As a view using the MapView class**. MapView is a subclass of View and can be used in two modes: **Fully Interactive Mode** or **Lite Mode**. You can place `MapView` directly inside your own fragment or activity. When using this in fully interactive mode, you are responsible for forwarding lifecycle methods to the MapView. In lite mode, forwarding the lifecycle events is optional.

The template uses the MapFragment option. Take a look at `onCreate` in `MapsActivity`:

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_maps)
  // Obtain the SupportMapFragment and get notified when the map is ready
to be used.
  val mapFragment = supportFragmentManager
      .findFragmentById(R.id.map) as SupportMapFragment
  mapFragment.getMapAsync(this)
}
```

It loads the **activity_maps.xml** layout, then finds the map fragment from the layout and uses it to initialize the map using `getMapAsync` method.

activity_maps.xml contains nothing but a container for the SupportMapFragment mentioned earlier.

```
<fragment
    android:id="@+id/map"
    android:name=
        "com.google.android.gms.maps.SupportMapFragment"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:map="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.raywenderlich.placebook.MapsActivity"/>
```

Asynchronous map setup

When you call `getMapAsync`, the SupportMapFragment object handles all of the work of setting up the map and creating a **GoogleMap** object. The GoogleMap object is what you will use to control and query the map.

If you are familiar with the concept of asynchronous methods, you may have guessed from the name that `getMapAsync` is asynchronous. Unlike a normal or synchronous method, which does its work then returns to the caller, an asynchronous method starts up a different thread to do its work and doesn't return immediately to the caller. The code that calls the asynchronous method goes on its merry way while the real work is done behind the scenes.

While `getMapAsync` is doing its background work, you should not try to interact with the map. So how will you know when the map is ready? That's where `OnMapReady` comes to the rescue!

Take a look at `OnMapReady`:

```kotlin
override fun onMapReady(googleMap: GoogleMap) {
  mMap = googleMap

  // Add a marker in Sydney and move the camera
  val sydney = LatLng(-34.0, 151.0)
  mMap.addMarker(MarkerOptions().position(sydney).title("Marker in
Sydney"))
  mMap.moveCamera(CameraUpdateFactory.newLatLng(sydney))
}
```

The `override` keyword on `onMapReady` lets you know that this is overriding a method from the base class or an interface. In this case, `onMapReady` is part of the `OnMapReadyCallback` interface included in the class declaration.

`OnMapReady` will be called by the SupportMapFragment object when the map is ready to go. You're passed in a `GoogleMap` object that is then used to interact with the map.

> **Note:** It's possible that the device running your app won't have the Google Play services installed. If this is the case, the SupportMapFragment object will prompt the user to install the Google Play services. `getMapAsync` will not call `onMapReady` until the services are installed.

The `GoogleMap` object is stored away in the `mMap` local variable, and then a couple of methods are used to add a marker and zoom the map to it. Don't worry about how the methods work for now; you'll cover those in detail in upcoming chapters.

> **Note**: You might be wondering why the `GoogleMap` object is being held in a variable named mMap. It may seem like a typo; however, Android Studio generates code using what's known as **Hungarian Notation**. It was developed during a time where advanced development environments like Android Studio didn't exist. The notation was used as a way for developers to easily identify if a variable was a class property, or a local variable, or a static variable.
>
> Now, development environments use colors to help you identify variables and their scopes. You can read all about it at https://en.wikipedia.org/wiki/Hungarian_notation. For the purposes of this book, we'll just use sensible naming, so go ahead and rename `mMap` to just `map`.

The difficulty of determining locations

Determining a user's location is a rather involved process under the hood. There are multiple sources of location data to handle, and they all affect your device's idea of where it is in the world.

Some of the challenges in locating the user's location are:

- **Dealing with multiple methods for determining location.** Your mobile device has several ways to determine your location, and each has its own benefits and disadvantages. Your phone uses the GPS chip, WiFi location and cell towers to zero in on your location. You have to decide which one to use in order to balance desired accuracy with power consumption.

- **Tracking change in user location.** As the user moves around you have to know when to update the location to reflect the current position.

- **Handling different accuracy levels.** Each location source offers different levels of accuracy and can vary at any time. In some cases an older location has better accuracy than the most recent location.

Android uses **Location Providers** to give access to the different location sources mentioned above. When you need the user's location, you decide which set of location providers to use and instruct them to start listening for location updates through the location manager.

A typical flow to get the user's location would look like this:

1. At a point after the application has started, begin listening for updates from your chosen location providers.

2. Implement logic to filter out updates and select the most appropriate ones. Remember that newer locations are not always the best.

3. Stop listening when you are done to preserve power.

4. Make use of the best location in your app logic.

For instance, the following graphic illustrates the location signals that your application might receive as it goes along. Note the graph shows a time sequence of location events.
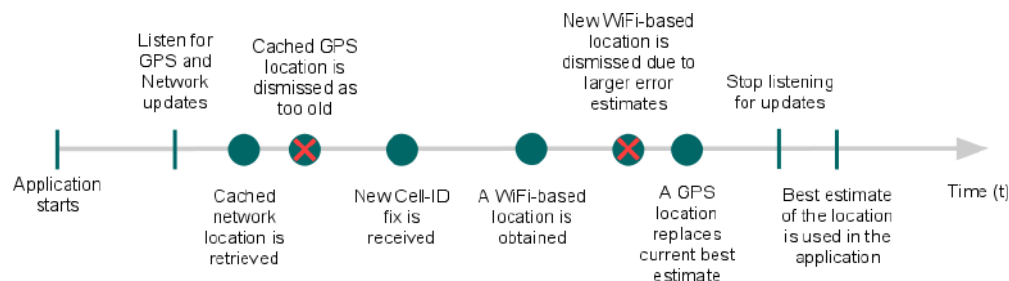
Location events over time

There are many decisions to make in order to determine how to best calculate the user's location:

- **Determine exactly when to start listening for updates.** You may want to start listening before the location is needed, so the user doesn't perceive a delay.

- **Determine the filter criteria for weeding out locations based on their accuracy and time received.** Do you want the quickest locations? The most accurate? Or some combination of the two?

- **Determine how long to listen to balance power efficiency.** On a mobile device, battery is a precious resource. Keeping the location provider running will drain this resource faster than just about anything else on the device.

As you can see, there are a lot of moving parts, and there's a lot of code required to provide a seamless experience to the user. Thankfully, the location APIs are there to do the heavy lifting for you!

That's it for this chapter! In the next chapter you'll get your first look at customizing the map behavior with location tracking and markers.

Chapter 14: User Location & Permissions

By Tom Blankenship

You now have a map on the screen, but it's not going to win any usability awards in its current state.

For starters, the map always starts off centered over Sydney, Australia. Unless that's where the user happens to be located, they'll have to pan and zoom around to find their current location. The other issue is there's no way to track the user's location as they move.

In this chapter, you'll address some of these problems by adding the following features to the app:

- Automatically center the map on the user's location at startup.

- Allow the user to recenter the map to their current location at any time.

Getting started

If you were following along with your own app, open it and keep going with it for this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the PlaceBook app under the **starter** folder. If you do use the starter app, don't forget to add your `google_maps_key` in **goole_maps_api.xml**. Check out Chapter 13 for more details about the Google Maps key. The first time you open the project, Android Studio takes a few minutes to set up your environment and update dependencies.

Let's get going. Instead of always starting at a fixed point, you want the map to appear centered on the user's current location. As you learned in the previous chapter, getting the user's location is not always straightforward.

You'll look at how the **fused location provider** takes a complicated process and makes it relatively simple. The previous chapter gave you a brief introduction to the fused location provider. This chapter will take a more in depth look at how it works.

Fused location provider

The job of the fused location provider is to take all the different inputs provided by the hardware and fuse them into location data that reflects the user's accuracy requests.

OK, that was a mouthful. Let's break down how it works in practice.

There are two primary ways to interact with the fused location provider:

1. Ask directly for the last known device location.

2. Request location updates based on hints about accuracy and power consumption.

Asking for the last known device location is a simple call to `FusedLocationProviderClient.getLastLocation()`. This returns a Task that can then be used to get the last known location of the device. This may return `null` if the device has not yet retrieved a location.

In the second scenario, requesting location updates based on hints, you ask for periodic location updates by calling `FusedLocationProviderClient.requestLocationUpdates()` and indicating your priorities with `LocationRequest`.

The fused location provider will use the most appropriate sensors on the device to match your priorities while preserving as much battery power as possible.

You can request location updates in two ways:

1. **Using a LocationListener callback method.** This method works best when your app is running in the foreground and actively displaying the user's location. Whenever there is relevant location data available, this makes an asynchronous call to a method you've defined yourself.

2. **Using a PendingIntent**. This is useful when you want to be notified of location events, even if your app is not currently running.

Adding location services

The fused location provider is part of the location services library within Google Play Services. Before using it, you'll need to add a new dependency.

Open **build.gradle (Module:app)** and add the following line to the dependencies section, taking care to use the same version as the existing `play-services-maps` dependency in the prior chapter.

```
implementation 'com.google.android.gms:play-services-location:11.8.0'
```

This adds the location APIs to the app.

Note: The Google Play services APIs provide a wealth of useful features. You'll explore more of them in later sections of the book, but if you want a sense of the depth of capabilities, check out the list of services at https://developers.google.com/android/.

Ad-Hoc Gradle properties

Before moving on, this is a good time to practice the DRY principle in our Gradle dependency management. Your **app/build.gradle** dependencies section now has two entries for `play-services` that both use the **11.8.0** library version. You'll fix that by adding some ad-hoc properties using Gradle's **ExtraPropertiesExtension**.

As your Gradle files grow with more dependencies they can be easier to manage if you define the library versions in a single location. The place to define global Gradle ad-hoc properties is in the project Gradle file.

Open your **project build.gradle** and remove the following line:

```
ext.kotlin_version = '1.2.21'
```

Note: Your version might be different depending on when you're following this book. Simply remove whatever `ext.kotlin` version your file has.

Update the first part of the `buildscript` section to match this:

```
buildscript {
  ext {
    kotlin_version = '1.2.21'
    play_services_version = '11.8.0'
  }
```

You now have two properties defined within the buildscript domain that can be accessed from any .gradle file in the project.

Open **app/build.gradle** and update the play services dependencies to take advantage of the new `play_services_version` extension property.

```
implementation "com.google.android.gms:play-services-maps:" +
    "$play_services_version"
implementation "com.google.android.gms:play-services-location:" +
    "$play_services_version"
```

> **Note:** The single quotes must be changed to double quotes when using extension properties. The string doesn't need to be split up as shown in the example, that's only done for readability in the book.

Creating the location services client

In order to use the fused location API, you must create a Fused Location Provider Client using the `FusedLocationProviderClient` class.

In **MapsActivity.kt**, add a new private member below the `map` member:

```
private lateinit var fusedLocationClient: FusedLocationProviderClient
```

Add the following method to `MapsActivity` under the `onMapReady` method:

```
private fun setupLocationClient() {
   fusedLocationClient =
       LocationServices.getFusedLocationProviderClient(this)
}
```

Finally, add a call to `setupLocationClient()` at the bottom of `onCreate()`.

```
setupLocationClient()
```

Querying current location

Next, you'll start by trying to query the user's current location, then place a marker and center the map on the location. Location detection requires the user's permission before it will work in your app.

Before moving on to the details of location permissions, a quick overview of how permissions work on Android is in order.

Permissions overview

Each app running on your Android device lives in its own little world. This is known as process sandboxing. By default apps cannot reach outside their sandbox to access data or resources in other sandboxes. This is done to protect the user's privacy as well as system stability.

If your app needs to reach outside its sandbox and access protected features, it must add a `<uses-permission>` tag to the apps manifest file. Android divides permissions into two main categories; **Normal** and **Dangerous**.

- **Normal permissions**: Permissions in this category are considered less harmful and will be granted automatically if they are listed in the manifest. Examples of normal permissions include **BLUETOOTH**, **ACCESS_NETWORK_STATE**, **INTERNET**, and **SET_ALARM**.

- **Dangerous permissions**: Permissions in this category could affect user's privacy or system stability. For these permissions, the system will explicitly ask the user to allow the permissions. Examples of dangerous permissions include **READ_CALENDAR**, **READ_CONTACTS**, **CALL_PHONE**, and **SEND_SMS**.

  Android will handle the dangerous requests differently depending on the OS version. If running Android 6.0 or higher and the app's **targetSdkVersion** is 23 or higher, you must request the user approval at run-time. On this version, the user can revoke individual permissions at any time, so the app must check for permissions every time it uses a protected feature. Even though you'll request dangerous permissions at run-time, they still must be specified in the manifest.

  If running Android 5.1.1 or lower, or the app's **targetSdkVersion** is 22 or lower, the user will be asked to approve the permissions when the app is first installed. If an app update adds new permissions , then the user will be asked to approve the new permissions when the app is updated. On this version, the user can only remove permissions by uninstalling the app.

In addition to the primary categories, the dangerous permissions are separated into groups. Android will not display the specific permission when asking the user for permission, it will only show the group that the permissions belong to.

For example, the **SEND_SMS** and **RECEIVE_SMS** permissions are part of the **SMS** group. If your app requests **SEND_SMS** and **RECEIVE_SMS** permissions, only a single **SMS** permission will be requested by the system.

> **Note**: It's also possible for an app to define its own permissions. This allows an app to share resources or capabilities with other apps.
>
> You can learn more about this feature at https://developer.android.com/guide/topics/permissions/defining.html.

The first run-time permission you'll use is **ACCESS_FINE_LOCATION** from the **LOCATION** group. It's already specified with the **<uses-permission>** tag in the manifest file. Now, you'll check for it at run-time before any code uses the location features.

Permission accuracy options

Your app can choose between two levels of location accuracy:

1. **ACCESS_FINE_LOCATION**: Used when you want the most accurate location data possible. This uses all location sources, including the GPS chip, and will use more battery.

2. **ACCESS_COARSE_LOCATION**: The less "refined" location permission. If you don't need a location more accurate than a city block, then choose this option. This only uses the Wi-Fi and cell towers to provide location data.

You should only choose one of these options.

In PlaceBook, you want to get the most accurate location readings, so you'll use `ACCESS_FINE_LOCATION`.

Adding run-time permissions

Open up **MapsActivity.kt** and add the following method:

```
private fun requestLocationPermissions() {
  ActivityCompat.requestPermissions(this,
      arrayOf(Manifest.permission.ACCESS_FINE_LOCATION),
      REQUEST_LOCATION)
}
```

Ignore the unresolved reference for `REQUEST_LOCATION`, you'll define it next.

This method uses `requestPermissions()` to prompt the user to grant or deny the `ACCESS_FINE_LOCATION` permission. Notice that this is the same permission as in **AndroidManifest.xml**.

You pass the current activity as the context, then an array of requested permissions, and finally a `requestCode` to identify this specific request.

Add the following to the `MapsActivity` class:

```
companion object {
  private const val REQUEST_LOCATION = 1
  private const val TAG = "MapsActivity"
}
```

`REQUEST_LOCATION` is a request code passed to `requestPermissions()`. It will be used to identify this specific permission request when the result is returned by Android.

`TAG` will be passed into the `Log.e` method in the next code block. `Log.e()` is used to print information to the Logcat window to help see with debugging.

With that in place, you're ready to create a method to get the user's current location.

Add the following new method:

```
private fun getCurrentLocation() {
  // 1
  if (ActivityCompat.checkSelfPermission(this,
        Manifest.permission.ACCESS_FINE_LOCATION) !=
    PackageManager.PERMISSION_GRANTED) {
    // 2
    requestLocationPermissions()
  } else {
    // 3
    fusedLocationClient.lastLocation.addOnCompleteListener {
      if (it.result != null) {
        // 4
        val latLng = LatLng(it.result.latitude, it.result.longitude)
        // 5
        map.addMarker(MarkerOptions().position(latLng)
            .title("You are here!"))
        // 6
        val update = CameraUpdateFactory.newLatLngZoom(latLng, 16.0f)
        // 7
        map.moveCamera(update)
      } else {
        // 8
        Log.e(TAG, "No location found")
      }
    }
  }
}
```

`getCurrentLocation()` gets the user's current location and moves the map so it centers on the location.

To understand this, take things one step at a time:

1. First, you check if the `ACCESS_FINE_LOCATION` permission has been granted before requesting a location.

2. If permission have not been granted then `requestLocationPermissions()` is called.

3. This may look a little odd. Why is `addOnCompleteListener` called on the `lastLocation` property? The reason is that the `lastLocation` property is actually a **Task** that will run in the background to fetch the location. You request to be notified when the location is ready by adding an `OnCompleteListener` to the `lastLocation` Task.

   When the Task is completed, it calls the default `onComplete()` method with a `Task<TResult>` object. `it.result` represents a **Location** object containing the last known location. `it.result` can be `null` if there is no location data available. The reason for this will be explained soon.

4. If `it.result` is not `null`, you create a `LatLng` object from `it.result`. `LatLng` is just a simple object for storing the latitude and longitude coordinate for a single map location. You'll see this often when working with location services.

5. You use `addMarker()` on `map` to create a marker at the location. `addMarker()` tells the map to add and display the marker. There are many options when adding markers to a map. In this case, you're using the default marker style with a simple title that will display if tapped on. You'll learn more about markers in future chapters.

6. You use `CameraUpdateFactory.newLatLngZoom()` to create a `CameraUpdate` object. `CameraUpdate` objects are used to specify how the map camera is updated. Let's cover this part in more detail.

When working with Google Maps, you can change the view of the map by adjusting parameters on a virtual map camera. You can think of the map view as a flat plane with the virtual camera looking straight down on it. The main camera properties you can adjust are:

- **Target**: This is the location the camera is looking at. The map is always centered on this location.

- **Bearing**: This is the direction that a vertical line on the map will point. This starts at 0 degrees north and increases in a clockwise direction. For example, if you wanted the top of the map to be east, you would set the bearing to 90 degrees.

- **Tilt**: Maps can be shown at an angle to give a perspective view. The tilt is the angle in degrees from the camera nadir line (the line pointing directly down from the camera).

- **Zoom**: You set the scale of the map using this parameter. Larger values zoom you closer to the map and display more detail. A zoom value of 0 will show the full Earth on a 256dp-wide screen. A zoom level of 15 is typical for a street level view.

CameraUpdateFactory provides several convenience methods for creating CameraUpdate objects. You use newLatLngZoom() to specify updates to the camera target and zoom.

> **Note:** See https://developers.google.com/android/reference/com/google/android/gms/maps/CameraUpdateFactory for additional options for CameraUpdateFactory.

7. You call moveCamera() on map to update the camera with the CameraUpdate object.

8. If result is null you log an error message.

With getCurrentLocation() implemented, it can be called once the map is ready.

Replace onMapReady() with the following code.

```kotlin
override fun onMapReady(googleMap: GoogleMap) {
  map = googleMap
  getCurrentLocation()
}
```

Here you initialize map when the map is ready to be displayed and then call the getCurrentLocation() method you just implemented.

Finally, define the callback method to handle the user's response to the permission request. When requestLocationPermissions() is called, the system will be display a permission dialog to the user. It will then call onRequestPermissionsResult() with the results.

```kotlin
override fun onRequestPermissionsResult(
    requestCode: Int,
    permissions: Array<String>,
    grantResults: IntArray) {
  if (requestCode == REQUEST_LOCATION) {
    if (grantResults.size == 1 && grantResults[0] ==
        PackageManager.PERMISSION_GRANTED) {
      getCurrentLocation()
    } else {
      Log.e(TAG, "Location permission denied")
    }
  }
}
```

First, you check to make sure this result matches the REQUEST_LOCATION request code. Next, you check to see if the first item in the grantResults array contains the PERMISSION_GRANTED value. If so, the use granted permission and you call getCurrentLocation() again. If grantResults doesn't indicate permission was granted, then you print an error message to the Logcat window using Log.e().

Testing permissions

Run the app on a hardware device or emulator running Android 6.0 or newer, and you should see the following prompt:
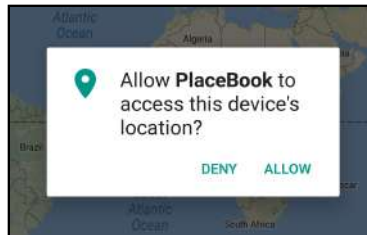
Click **DENY** and the **Location permission denied** message will appear in Logcat.

Rotate the device and the prompt will display again with one small change, offering the user a chance to tell the system "Don't ask again."
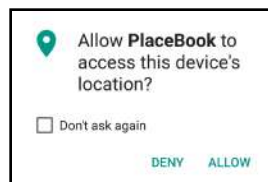
If you choose "Don't ask again", the dialog will never be displayed again within the app. The only way to then grant permissions is to manually turn them on in device settings by tapping on **Apps>PlaceBook>Permissions**.

> **Note:** Google recommends that you display a more detailed reason for asking for a permission if the user denies it multiple times. There is a built in method — ActivityCompat.shouldShowRequestPermissionRationale — you can use to determine if it's time to show a detailed reason. See https://developer.android.com/training/permissions/requesting.html#perm-request for more information.

Now click **Allow** on the permission dialog. At this point, you may expect that the app would return your current location and then zoom the map to your current location.

If you're running on a hardware device, that's most likely true and you'll be looking at a screen similar to the following, although centered at your current location.

If running on the emulator the map will likely not move, and you can see the **No location found** message printed in the Logcat window:

```
06-21 14:14:05.571 4230-4230/com.raywenderlich.placebook E/MapsActivity: No location found
```

It's because the emulator hasn't simulated a user location. An emulator doesn't have access to GPS hardware, so you need another way to supply GPS locations.

> **Note:** If you see the **No location found** message on a hardware device, then check that location services are turned on in the device settings.

Faking locations in the emulator

The problem is that the fused location provider doesn't have any location data to pull from. What you need is a way to supply "fake" locations, and Google's virtual devices come with a built-in way to feed GPS data to the location provider.

Launch your emulator and click the three dots (**…**) at the bottom of the floating toolbar to bring up the extended controls, and then click the **Location** tab on the left.
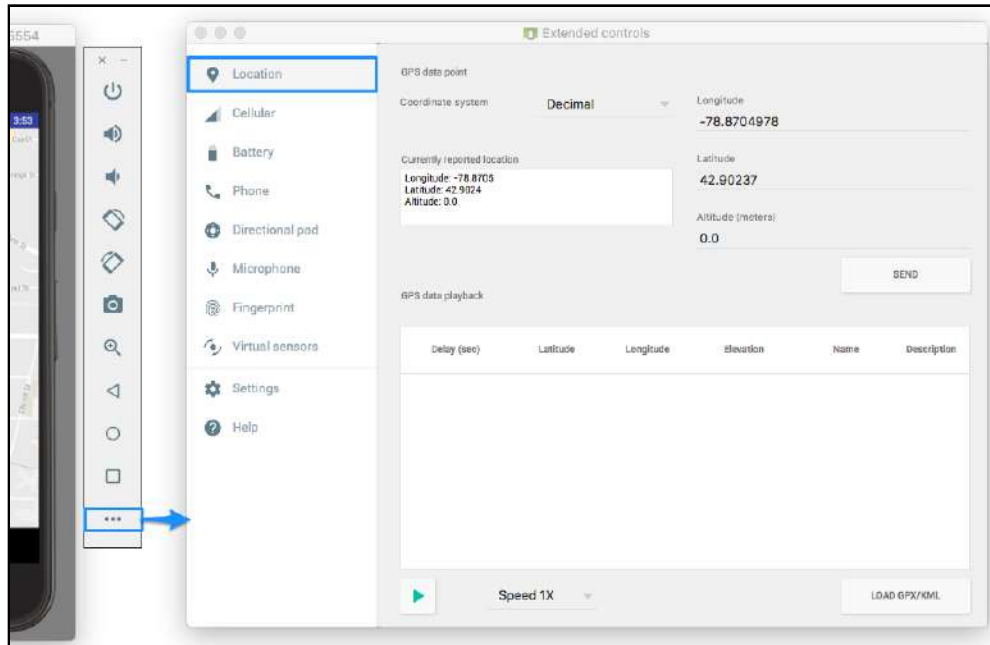
Enter the following coordinates and click **SEND**.

• Longitude: -78.8704978

• Latitude: 42.90237
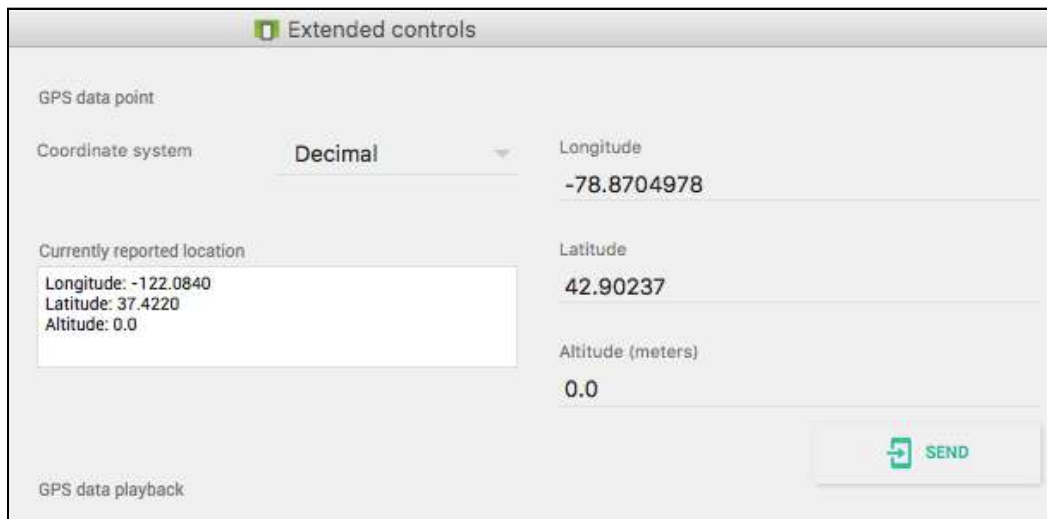
Close the app and run again, but the map still won't display that location. What's going on?

There's one final item to address. The fused location provider needs at least one app to actively request a location before it will return valid data from getLastLocation().

On a real device, there are usually plenty of other processes requesting locations and feeding the fusion location provider with data. That's not the case on the emulator.

One way to wake up the fusion API is to run the **Google Maps** app. Once you run Google Maps, click on the **My Location** icon (the target) and approve any prompts to turn on location services.

Once you see that Google Maps zooms you to the entered location, close and launch the PlaceBook app again. This time it should zoom to the location you entered.

If it doesn't work the first time, try and try again. Sometimes the emulator can be a little finicky, but eventually it should zoom to the entered location in Buffalo, New York.

In upcoming chapters, you'll update the app so it works in the emulator without being triggered by Google Maps.

Tracking the user's location

It's great that you have a way to display the user's location when the app first launches — but what happens when the user moves to a new location? No problem. Simply relaunch the app and it will update to your new location.

That's not the most intuitive way to update the map. You can do better!

You need a way to keep track of the user's location as they move around. This can be done by directly asking the fused location provider for periodic location updates. This is where the `FusedLocationClient.requestLocationUpdates()` comes into play. `FusedLocationClient.requestLocationUpdates()` asks the fused location provider to start sending the app location updates.

Calling requestLocationUpdates()

In order to request updates from the location client, you need a `LocationRequest` object to describe the level of accuracy you would like to achieve. Add the following new property at the top of `MapsActivity`:

```
private var locationRequest: LocationRequest? = null
```

Now, go to `getCurrentLocation()` and add the following before the call to `fusedLocationClient.lastLocation.addOnCompleteListener`:

```
if (locationRequest == null) {
  locationRequest = LocationRequest.create()
  locationRequest?.let { locationRequest ->
    // 1
    locationRequest.priority =
        LocationRequest.PRIORITY_HIGH_ACCURACY
    // 2
    locationRequest.interval = 5000
    // 3
    locationRequest.fastestInterval = 1000
    // 4
    val locationCallback = object : LocationCallback() {
      override fun onLocationResult(locationResult: LocationResult?) {
        getCurrentLocation()
      }
    }
    // 5
    fusedLocationClient.requestLocationUpdates(locationRequest,
        locationCallback, null)
  }
}
```

You first check to see if `locationRequest` has already been created. If not, you create a new one, and then if the creation succeeds you set the following properties:

1. **`priority`**: This provides a general guide to how accurate the locations should be. The following options are allowed:

   PRIORITY_BALANCED_POWER_ACCURACY: Use this setting if you only need accuracy to the city block level, which is around 40-100 meters. This will use very little power and only poll for location updates every 20 seconds or so. The system is likely to only use Wi-Fi or cell tower to determine your location.

   PRIORITY_HIGH_ACCURACY: Use this setting if you need the most accuracy possible, normally within 10 meters. This uses the most battery power and typically polls for locations about every 5 seconds.

   PRIORITY_LOW_POWER: Use this setting if you only need accuracy at the city level within 10 kilometers. This will use a minimal amount of battery power.

   PRIORITY_NO_POWER: You normally only use this setting if your app can live with or without location data. It will not actively request any location from the system, but will return a location if another app is requesting location data.

   Here you are set `priority` to `LocationRequest.PRIORITY_HIGH_ACCURACY` so it will return the most accurate location possible. In the emulator, anything less than `PRIORITY_HIGH_ACCURACY` may not trigger any updates to occur.

2. **`interval`**: This lets you specify the desired interval in milliseconds to return updates. This is simply a hint to the system, and if other apps have requested faster updates your app will get the updates at that rate as well.

   Here you set the requested update interval to 5 seconds by setting `interval` to 5000.

3. **`fastestInterval`**: This sets the shortest interval in milliseconds that your app is capable of handling. Since other apps can affect the update interval, this sets a hard limit on how often you'll receive updates. Here you set the shortest interval to 1 second with `locationRequest.fastestInterval = 1000`.

> **Note:** Keep in mind that the `LocationRequest` settings are more like guidelines than they are rules. The fused location provider will try to meet the requested options, but there are no guarantees.

4. The fused location provider will call LocationCallBack.onLocationResult when it has a new location ready. You define a `LocationCallBack` object with the `onLocationResult` method. You use this opportunity to update the map to center on the new location. Although `onLocationResult()` receives a list of locations that you could use to center the map, you just call the existing `getCurrentLocation()` to grab the latest location and center the map.

5. Finally, you call `fusedLocationClient.requestLocationUpdates()`, passing in the `LocationRequest` object, and the `LocationCallback` object.

After calling `requestLocationUpdates()`, your app can go about its business and wait for the `onLocationChanged()` to be called by the location services.

Add the following line in `getCurrentLocation()` before the call to `map.addMarker`:

```
map.clear()
```

Since `getCurrentLocation()` will be called each time the location changes, you need to call `clear()` on the `GoogleMap` object to remove the previous marker.

Testing location updates

Run the app again on the emulator, and it should center the map over the location you entered before. To verify that the location updates are working, try dragging the map away from the current location. Click the **SEND** button on the GPS Location controls for the emulator and you should see the map jump back to the entered location. Try entering some other coordinates and clicking the **SEND** button each time.

If you run this on a hardware device, you should notice the map jumping to your current location as you move around. If you drag the map to a new location it will jump back to your current location within a few seconds.

My location

Showing a marker at your current location works for demonstration purposes, but it's not the typical way to show the user's location. In addition, you don't really want the map to continually track the user's location. The user should be able to freely pan around the map and recenter at will.

You'll fix these two issues by making the following changes:

1. Display a blue dot at the user's location and have it move to keep up with the user.

2. Add a control that allows the user to recenter the map.

3. Disable the continuous map centering.

Believe it or not, changes 1 and 2 can be accomplished with one line of code with the magic of the `GoogleMap.isMyLocationEnabled` property.

Using GoogleMap.isMyLocationEnabled

The GoogleMap object already has the ability to do exactly what you need without any additional coding. The feature is called **MyLocation**; you enable it by setting the `isMyLocationEnabled` to `true`.

Add the following line to `getCurrentLocation()` before the call to `fusedLocationClient.lastLocation`:

```
map.isMyLocationEnabled = true
```

Setting `isMyLocationEnabled` adds a new layer to the map with several useful features:

1. It displays the trusty blue dot that always keeps up with the user's current location. Note that it does this without having to request location updates from the location services.

2. It displays a target icon that will recenter the map on the user's location if they tap on it.

3. It will add controls to let the user choose whether the map should rotate with the user's current bearing.

As a bonus, turning on `isMyLocationEnabled` handles all of the logic to request location updates, and you can remove the code for location updates that was added earlier.

Remove the following items:

1. Remove the following line from the top of `MapsActivity`:

```
private var locationRequest: LocationRequest? = null
```

2. Remove the following block of code from `getCurrentLocation()`:

```
if (locationRequest == null) {
  locationRequest = LocationRequest.create()
  locationRequest?.let { locationRequest ->
    // 1
    locationRequest.priority =
        LocationRequest.PRIORITY_HIGH_ACCURACY
    // 2
```

```
        locationRequest.interval = 5000
        // 3
        locationRequest.fastestInterval = 1000
        // 4
        val locationCallback = object : LocationCallback() {
          override fun onLocationResult(locationResult: LocationResult?) {
            getCurrentLocation()
          }
        }
        // 5
        fusedLocationClient.requestLocationUpdates(locationRequest,
            locationCallback, null)
    }
  }
```

3. Remove the following lines from `getCurrentLocation()`:

```
map.clear()
map.addMarker(MarkerOptions().position(latLng)
    .title("You are here!"))
```

Run the app and check out the great new functionality you added with minimal effort.
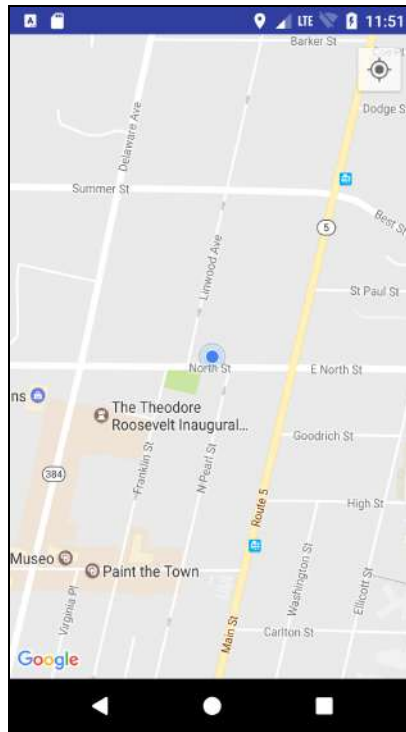
Click the **SEND** button on the GPS Location controls, and you should see the blue dot appear at the current location. Pan the map around and then click the **My Location** icon to recenter back to the blue dot.

Where to go from here?

Congratulations — you've completed everything needed for the basic map controls! In the next chapter, you'll start working with Google Places.

Chapter 15: Google Places

By Tom Blankenship

Before you can achieve your ultimate goal of allowing users to bookmark places, you need to let them identify existing places on the map.

In this chapter, you'll learn how to identify when a user taps on a place and use the **Google Places API** to retrieve detailed information about the place.

Getting started

If you were following along with your own app, open it and keep going with it for this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the PlaceBook app under the **starter** folder. If you do use the starter app, don't forget to add your `google_maps_key` in **goole_maps_api.xml**. Check out Chapter 13 for more details about the Google Maps key. The first time you open the project, Android Studio takes a few minutes to set up your environment and update dependencies.

If you're following along with your own app, you'll need to copy the following resource from the starter project into yours:

- src/main/res/drawable/default_photo.png

Make sure to copy the files from all of the drawable folders (hdpi,mdpi,xhdpi,xxhdpi).

Before using the Google places API, you'll need to take care of a bit of housekeeping first, by enabling the Places API in the developer console and adding the Places API dependency.

> **Note:** The Google screens in our book might be slightly different than what you see on the Google developer portal since Google changes these often.

Enable the places API

The Google Maps API was enabled on your Google developer account when you created the initial Google Maps key. But you'll need to turn the Google Places API on manually.

Log into your Google developer account at https://console.developers.google.com

Ensure the project containing the Maps API key you created previously is selected. Switch to the Library tab on the left.
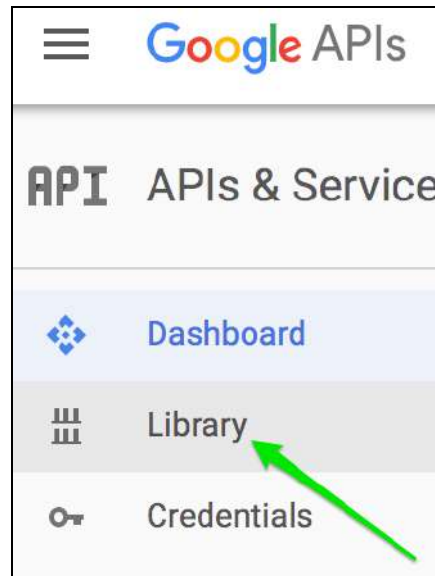
Click on **Google Places API for Android**.
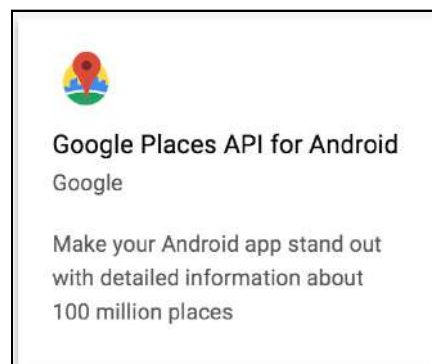
You should see the following screen with an **Enable** link at the top:
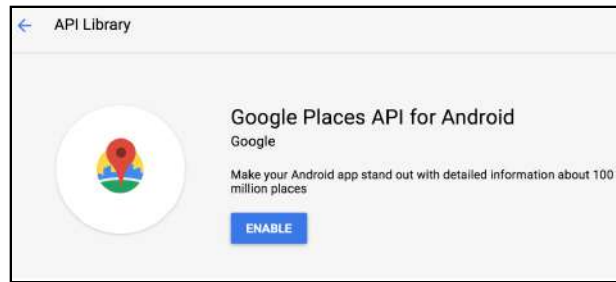
Click on **ENABLE** and wait while Google enables the API. Once the API is enabled the screen will change to an overview of the API.
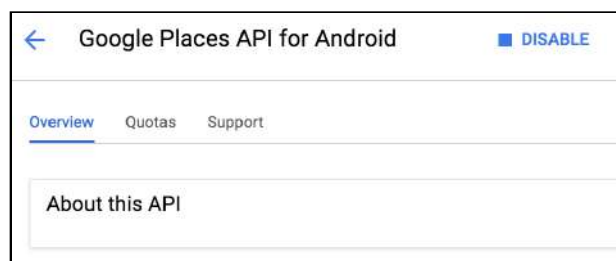
Click on the back arrow next to Google Places API for Android (and a few more times) until you get back to the main Dashboard. Both the Google Maps and Google Places API are listed.
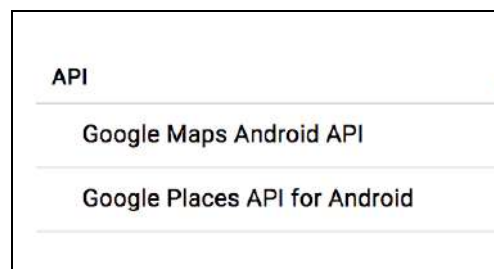
Places API overview

The Google Places API provides a wealth of capabilities all related to — wait for it — working with places on a map! A place is anything that can be identified on a map, such as a household, a business or a public park. Google places gives you access to over 100 million places stored in the main Google Maps database.

Although referred to as a single API, you generally interact with the Places API through a number of sub-APIs. The PlaceBook app uses the following two sub-APIs:

1. **Place Autocomplete API.** This lets you search places by name or address and returns results as the user types. These results can be filtered by location.

2. **Geodata API.** This lets you load up detailed information about a single place. You can access items such as address, geographic location, phone numbers, reviews and photos.

> **Note:** Google Places for Android enforces a default limit of 1,000 requests per 24 hour period. There will be a further API checkpoint when the app reaches 150,000 requests. To prevent your app from failing when it exceeds these limits, follow the instructions in the usage limits guide at https://developers.google.com/places/android-api/usage.

Add the Places API dependency

Just like the Places API, you'll have to add the Play services dependency for Google Places yourself.

Open **app/build.gradle** and add the `play-services-places` library to the dependencies section as follows:

```
implementation "com.google.android.gms:play-services-places:
$play_services_version"
```

This instructs the Gradle build system to include the Places API in your build.

Selecting points of interest

You may have noticed icons with place names scattered throughout the map. These are called points of interest, or **POI**s, and they will let the user look up details about each place. You'll begin by making the POIs a little more interesting by allowing the user to interact with them.

The **Google Map** object has convenient, built-in capabilities to let you know when the user has tapped on a POI. You simply need to set up a POI click listener and wait for the user to tap away.

Like all other interactions with the map object, you'll wait to set up the listener until
`OnMapReady()` is called. Open **MapsActivity.kt** and add the following to the end of
`onMapReady()`:

```
map.setOnPoiClickListener {
  Toast.makeText(this, it.name, Toast.LENGTH_LONG).show()
}
```

Here, you call `setOnPoiClickListener()` on `map` and provide it a lambda that
implements the single `onPoiClick()` method of the `PoiClickListener` interface.

The `map` object will call your lambda anytime it detects that the user has tapped on a
POI. The lamba is passed in a single parameter of type `PointOfInterest` that you access
through the implicit `it` variable.

`PointOfInterest` contains only three properties:

1. **latLng**: The geographic location of the selected POI represented by a latitude and
   longitude in decimal degrees.

2. **name**: The name of the POI. This will normally match what is shown on the map.

3. **placeId**: A string that uniquely identifies the POI. You can use the placeId to
   retrieve a `Place` object from the places API.

Run the app and tap on a few places. You'll see toasts pop up with the name of each
place you tap:
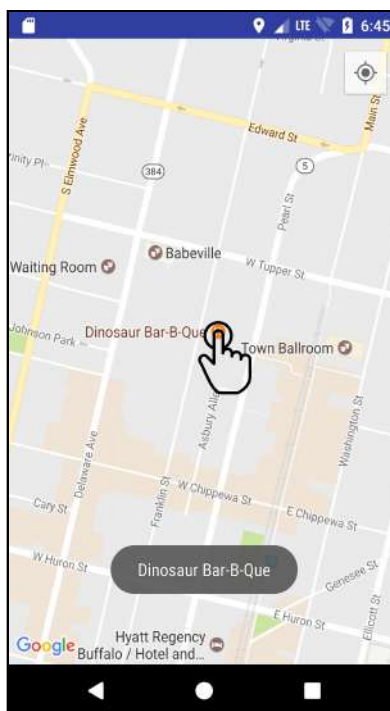
Load place details

Now that you have the `placeId` when a user taps a POI, you can use it to look up more details about the place. The goal is to provide the user with a quick popup info window, from which they can decide if they want to bookmark the place.

To retrieve the details for places, you'll use your first Places API: `GeoDataApi`.

Before using the GeoDataApi, you'll need to create a Google API client. This client is your gateway to all of the available APIs provided by Google Play services.

To create the Google API client, you'll use the familiar builder pattern. There are three primary methods you'll use with the GoogleApiClient builder:

1. **enableAutoManage**. This instructs the Google API client to auto manage the connections to the Google services and prompt the user if there is a problem such as the Google Play service requiring an update. It is rare that you would want to use the GoogleApiClient without making this call.

2. **addApi**. You call this once for each required play services API.

3. **addConnectionCallbacks**. This sets your activity up to receive callbacks when the client has connected to the play services.

In **MapsActivity.kt**, add a new private member below the `map` member:

```
private lateinit var googleApiClient: GoogleApiClient
```

Add the following method to `MapsActivity` under the `onMapReady` method:

```
private fun setupGoogleClient() {
  googleApiClient = GoogleApiClient.Builder(this)
      .enableAutoManage(this, this)
      .addApi(Places.GEO_DATA_API)
      .build()
}
```

This creates the `GoogleApiClient` with a builder, telling it to auto manage connections and add the `Places.GEO_DATA_API`. You must add at least one Api with the builder, and the `Places.GEO_DATA_API` will be used soon. The `addConnectionCallbacks` call is not required at this time.

You'll notice that you are passing in the `this` variable to both parameters on `enableAutoManage()`. The first one is the `FragmentActivity` to be managed. The second parameter is an object that implements the `OnConnectionFailedListener` interface. This second parameter will be flagged as an error because you haven't implemented the interface yet. Let's take care of that now.

Place the cursor over the problematic `this` and press **Option+Return** if you are using a Mac, or **Alt+Enter** on Windows. This will display options to auto-correct the problem. Click the one that says **"Let 'MapsActivity' implement the interface 'GoogleApiClient.OnConnectionFailedListener'"**.

This will display a second dialog with an option to add `onConnectionFailed()`. Highlight the single method and click **OK**.

This will add `GoogleApiClient.OnConnectionFailedListener` to your class declaration and add `onConnectionFailed()` to your file:

```
override fun onConnectionFailed(p0: ConnectionResult) {
  TODO("not implemented") //To change body of created
  // functions use File | Settings | File Templates.
}
```

The error about the `this` variable should go away.

Rename the `p0` argument to `connectionResult` and remove the `TODO` item. Now `onConnectionFailed()` looks like this:

```
override fun onConnectionFailed(connectionResult:
                                ConnectionResult) {
}
```

If `onConnectionFailed()` is called, it means that there was a serious error connecting to the Play service. At this point you can choose to display an error message, silently fail, or try to resolve the error. For PlaceBook, you'll silently fail with a log message.

> **Note:** Instead of failing silently, there's certainly more that you can be done to give the user more information. See https://developers.google.com/android/guides/api-client#handle_connection_failures for more details.

Add the following line to `onConnectionFailed()`:

```
Log.e(TAG, "Google play connection failed: " +
    connectionResult.errorMessage)
```

Add the following method to **MapsActivity.kt**:

```kotlin
private fun displayPoi(pointOfInterest: PointOfInterest) {
  // 1
  Places.GeoDataApi.getPlaceById(googleApiClient,
      pointOfInterest.placeId)
      // 2
      .setResultCallback { places ->
        // 3
        if (places.status.isSuccess && places.count > 0) {
          // 4
          val place = places.get(0)
          // 5
          Toast.makeText(this,
              "${place.name} ${place.phoneNumber}",
              Toast.LENGTH_LONG).show()
        } else {
          Log.e(TAG,
              "Error with getPlaceById ${places.status.statusMessage}")
        }
        // 6
        places.release()
      }
}
```

Let's break this down:

1. First, you make a call to `getPlaceById()`, passing in `googleApiClient` and the unique place identifier represented by the POI `placeId` property. This returns a `PendingResult` object.

2. You then call `setResultCallback()` on `PendingResult` and pass in a lambda that receives the `places` `PlaceBuffer` object.

3. Next, you check to make sure the result was successful and contains at least one place. Note that `getPlaceId()` can receive multiple place ids, and may return multiple places.

4. Following that, pick the first from the `places` buffer.

5. Then display the place name and phone number.

6. Next, release the `places` buffer so it doesn't cause a memory leak.

> **Note:** Some `PlaceBuffer` properties are always available, but many of them may return null or negative values if they are not available. Properties that are always available include `id`, `name`, `latLng` and `placeTypes`. Properties that may *not* always be available are `phone`, `address`, `priceLevel`, `rating` and `websiteUri`.

Now to update `setOnPoiClickListener()` to call this new method. In `onMapReady()`, replace the call to `map.setOnPoiClickListener()` with the following:

```
map.setOnPoiClickListener {
   displayPoi(it)
}
```

This calls `displayPoi()` when a place on the map is tapped.

Pending results

Let's go over the `PendingResult` object in more detail, since it's commonly used throughout the Google Places APIs.

`PendingResult` represents a pending result from a call made to a Google Play service API method. Many of the Google Play API methods make network calls and can take a long time to return. For this reason, Google Play services offloads these tasks to the background and gives you a `PendingResult` that can be used in two different ways:

1. **Blocking mode**: In this mode, you call `await()` on the `PendingResult` and your code stops and waits for the results to come back. This is useful if you want your logic to proceed synchronously in-line. This mode should not be used on the UI thread because it can block user interactions or cause your app to appear frozen.

2. **Asynchronous mode**: In this mode, you call `setResultCallBack()`, passing in a callback method. `setResultCallBack()` returns immediately and your code continues on without waiting for the results. When the results are ready, the `PendingResult` object calls your method on the UI thread.

In both cases, you end up with an object that implements the `Result` interface and represents the final results. The `Result` interface has a `status` property that can be used to query the status of the result. You should always check `status.isSuccess()` before working with a `Result`.

Google places buffers

The call to `getPlaceById()` returns a result of type `PlaceBuffer`. `PlaceBuffer` is one of many buffer types used by the Google Places API, and you'll encounter more of them throughout the rest of this chapter. Buffers provide collections of objects that are managed by the Google places APIs.

You should be aware of two key points when working with buffers:

1. Calling `release()` on a buffer is critical to prevent memory leaks.

2. If you need to use an object from the buffer even after it has been released, you should call `freeze()` first. You can use `isDataValid()` on a buffer object to check if the object has been released already.

Run the app and tap on a few more places. This time, you'll see the place name and its phone number, if one is available.

> **Note:** If you don't see the toast pop up, check the Logcat for error messages. If you see **PLACES_API_ACCESS_NOT_CONFIGURED**, then check that you have enabled the Google Places API in the developer console.

You have a lot of details about the place, but wouldn't it be nice to also have a photo?

Getting a photo is not as simple as getting the basic place details, but armed with your newfound knowledge of result callbacks and buffers, you're up to the task!

You'll use the same result callback pattern to get a photo for the selected place with a separate call to `getPlacePhotos`.

Loading a place photo

You'll use the following steps to get a photo for a selected place:

1. Call `GeoDataApi.getPlacePhotos()` to get back the `PlacePhotoMetadataResult`.

2. Get the first `PlacePhotoMetadata` object from the resulting `PlacePhotoMetadataBuffer` of `PlacePhotoMetadataResult`.

3. Get a scaled down version of the first photo by calling `getScaledPhoto()` on `PlacePhotoMetadata`.

Putting together the entire sequence to get the place details and place photo using the asynchronous setResultCallback pattern can lead to some deeply nested code. You'll place each main step in its own method to keep things nice and clean. You could start by refactoring `displayPoi()` to kick off the first step and this (but don't actually do this yet):

```kotlin
private fun displayPoi(pointOfInterest: PointOfInterest) {
  displayPoiGetPlaceStep(pointOfInterest)
}
```

```
private fun displayPoiGetPlaceStep(pointOfInterest: PointOfInterest) {
  Places.GeoDataApi
      .getPlaceById(googleApiClient, pointOfInterest.placeId)
      .setResultCallback { places ->

        if (places.status.isSuccess && places.count > 0) {
          val place = places.get(0)
          Toast.makeText(this,
              "${place.name} ${place.phoneNumber}",
              Toast.LENGTH_LONG).show()
        } else {
          Log.e(TAG, places.status.statusMessage)
        }
        places.release()
      }
}
```

Refactoring in Android Studio

All you really would have done here was take the code inside of `displayPoi()` and move it into a new method that takes a single argument. You would then have added a call to the new method inside `displayPoi()`. This is common refactoring step that Android Studio can automate for you.

Instead of manually cutting and pasting or typing in the method call, try this:

1. Select all of the code inside `displayPoi()`.

2. Press **Cmd+Option+M** on Mac or **Ctrl+Alt+M** on Windows to initiate the **Extract Function** command.

3. Type in the name of the new method: `displayPoiGetPlaceStep`. Look at the preview window and notice that Android Studio is smart enough to add the `pointOfInterest` parameter that it knows you will need in the new method.

4. Click **OK**.

Voilà! The method is created and the call is added to `displayPoi()`.
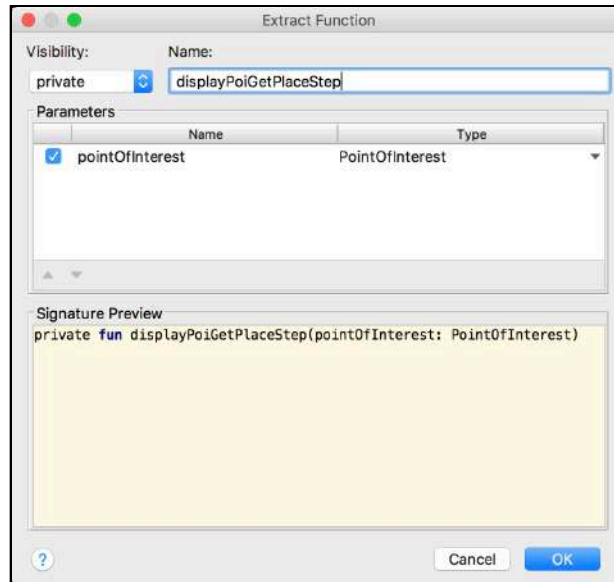
Getting the metadata

Add the following new method to **MapsActivity**:

```
private fun displayPoiGetPhotoMetaDataStep(place: Place) {
  Places.GeoDataApi.getPlacePhotos(googleApiClient, place.id)
      .setResultCallback { placePhotoMetadataResult ->

        if (placePhotoMetadataResult.status.isSuccess) {
          val photoMetadataBuffer =
              placePhotoMetadataResult.photoMetadata

          if (photoMetadataBuffer.count > 0) {
            val photo = photoMetadataBuffer.get(0).freeze()
            // next step here
          }
          photoMetadataBuffer.release()
        }
      }
}
```

Here you call `getPlacePhotos()`, passing in `googleApiClient` and the place ID. You call `setResultCallback()` on the `PendingResult`, passing it a lambda. The lambda takes in a `PlacePhotoMetadataResult` and checks that the call was successful. You then grab the `photoMetadataBuffer` from the results and make sure that it contains at least one item. You get the first and only `PhotoMetaData` item from the buffer and assign it to `photo`.

Replace the Toast call in `displayPoiGetPlaceStep()` with the following:

```
    displayPoiGetPhotoMetaDataStep(place)
```

Remember the call to `release()` that you added to make sure memory wasn't leaked for the `places` buffer? Well, that will cause an issue now that you are passing along the `place` object from the buffer and using it with another asynchronous call.

To understand why, take a look at the following diagram:
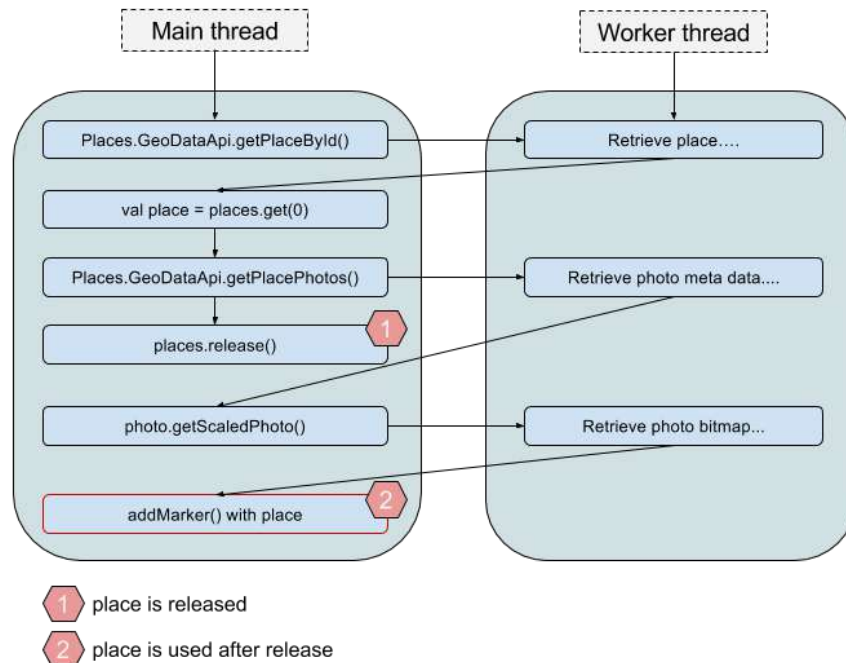
Notice that the call to `places.release()` happens before the `place` object is referenced when adding the marker. To fix this, you'll use the previously mentioned `freeze()` method on the `place` object. In `displayPoiGetPlaceStep()`, update the line that calls `places.get(0)` like so:

```
val place = places.get(0).freeze()
```

Now you have guaranteed that `place` will be retained in memory even after the `places` buffer has been released. Next, you'll add a step to retrieve the scaled down photo from `PhotoMetaData`.

Add the following new method to **MapsActivity**:

```
private fun displayPoiGetPhotoStep(place: Place, photo:
PlacePhotoMetadata) {
  photo.getScaledPhoto(googleApiClient,
      resources.getDimensionPixelSize(R.dimen.default_image_width),
      resources.getDimensionPixelSize(R.dimen.default_image_height))
      .setResultCallback { placePhotoResult ->

    if (placePhotoResult.status.isSuccess) {
```

```
      val image = placePhotoResult.bitmap
      // next step here
    } else {
      // next step here
    }
  }
}
```

Before explaining this method, fix the unresolved references for
`R.dimen.default_image_width` and `R.dimen.default_image_height` that are displayed
in red. To resolve the errors, follow these steps:

1. Place the cursor on `default_image_width` and press **Alt + Return**, then select
   Create a dimen value resource 'default_image_width'.

2. In the dialog that appears, set **Resource Value** to **480px** and leave the other values
   at their defaults. Click **OK**.
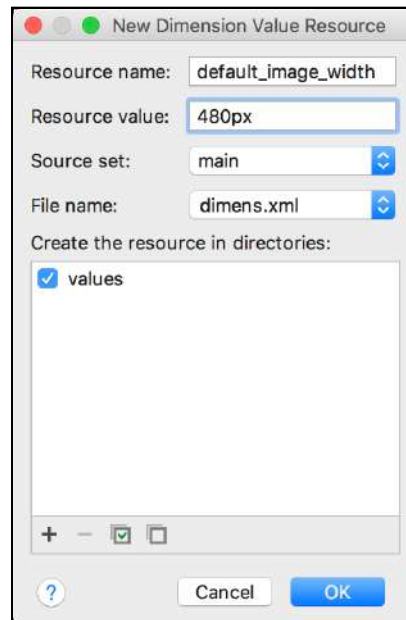
3. Place the cursor on `default_image_height` and type **Alt + Return**, then select
   Create a dimen value resource 'default_image_height'.

4. In the dialog that appears, set **Resource Value** to **270px** and leave the other values
   at their defaults. Click **OK**.
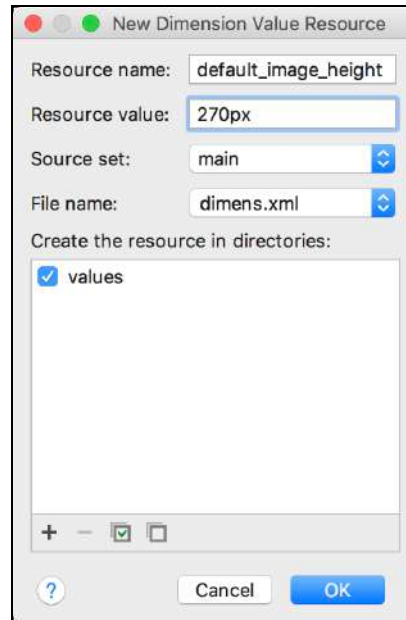
This will create two values in **res/values/dimens.xml** for the default image width and height. You'll see these values pop up again as you build out the app.

In `displayPoiGetPhotoStep()`, you use `getScaledPhoto()` on `photo` to get a scaled down version of the original photo. `getScaledPhoto()` takes in a width and height and will scale the image proportionally to fit the smaller of the two dimensions.

`PlacePhotoMetata` also contains a `getPhoto` method that will retrieve the full size image.

There are two benefits to using `getScaledPhoto()` instead of `getPhoto()`.

1. **Memory savings**: In general, you never want to load photos into memory that are larger than required. By using the `getScaledPhoto()` you limit the possibility of memory issues that can happen on lower end devices.

2. **Bandwidth savings**: `getScaledPhoto()` does the scaling on the server-side and only sends down the final version.

Check the `status` of the result for success and then assign the photo to `image`.

> **Note:** You could have used two hard coded numbers for the width and height parameters to `getScaledPhoto()`, but these are considered "magic" numbers in the code and should always be avoided.
>
> Placing them in **dimen.xml** is two steps closer to coding nirvana: you gain reuse and follow the DRY principle with a single location for updating the values, as well as built-in documentation for the types of values that the numbers represent.

Replace the commented line `// Next step here` in `displayPoiGetPhotoMetaDataStep()` to call your new step:

```
displayPoiGetPhotoStep(place, photo)
```

Add a place marker

Finally, add a step to display a marker with the place details and photo.

Add the following new method to **MapsActivity**:

```
private fun displayPoiDisplayStep(place: Place, photo: Bitmap?) {
  val iconPhoto = if (photo == null) {
    BitmapDescriptorFactory
        .defaultMarker()
  } else {
    BitmapDescriptorFactory.fromBitmap(photo)
  }

  map.addMarker(MarkerOptions()
      .position(place.latLng)
      .icon(iconPhoto)
      .title(place.name as String?)
      .snippet(place.phoneNumber as String?)
  )
}
```

If `photo` is `null` you create `iconPhoto` as a default marker bitmap. If it's not `null`, you create `iconPhoto` from the `photo`. Next, add a marker to the map by creating a new `MarkerOptions` object and setting the properties to the place details and the `iconPhoto`.

Using markers will be covered in more detail soon, but for now it's enough to know that `addMarker()` places a persistent marker on the map represented by an icon. The default marker icon is a red balloon pin, but can be replaced with any bitmap image. Markers will respond to user taps and display an info window with more details.

Replace the first commented line `// Next step here` in `displayPoiGetPhotoMetaDataStep()` to call your new step:

```
displayPoiDisplayStep(place, image)
```

Here you pass along the `place` object and the bitmap `image`.

Replace the second commented line `// Next step here` in `displayPoiGetPhotoMetaDataStep()` to call your new step:

```
displayPoiDisplayStep(place, null)
```

Here you include the `place` object and null, indicating that no image is available.

Run the app and tap on some places. You should see place photos appear on the map. Tap on a photo to display an info window with the place name and phone number.
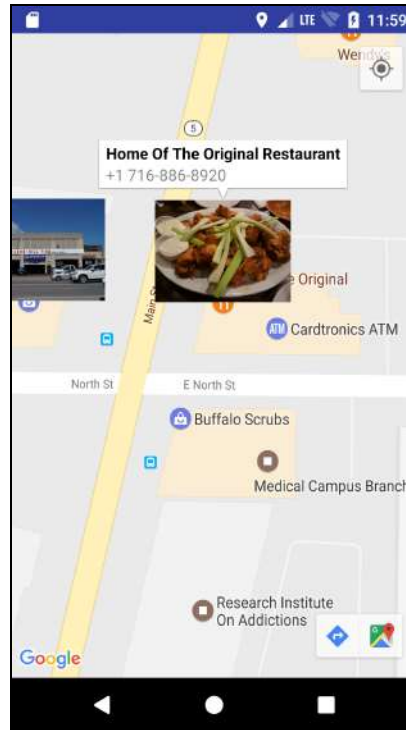
Custom info window

Now you're making some progress! The user is able to tap places to view a photo and details, but having large photos all over the map is a little unwieldy. A better experience would be to display a standard marker next to each place and only show the photo and details in a popup info window.

By default, tapping on a marker displays a standard info window. This window looks like the following:
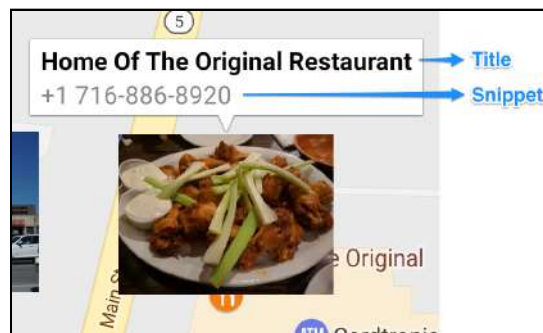
The standard info window will display the `title` and `snippet` as defined on the marker.

If you want to display additional information, a custom info window is in order.

InfoWindowAdapter class

To create a custom info window, you simply create a class that conforms to the **InfoWindowAdapter** interface and then call `map.setInfoWindowAdapter()` with an instance of the class.

There are two methods to implement in your InfoWindowAdapter class:

1. `getInfoWindow()`: This one allows you to return a custom view for the full info window.

2. `getInfoContents()`: This allows you to return a custom view for the interior contents of the info window only without changing the default outer window and background.

In your case, only the info window contents will be replaced. Before creating a custom info window, you need to create a layout file for the contents. The layout will look like this:
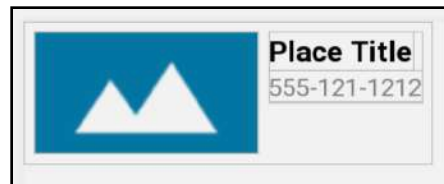
Create a new layout resource file named `res/layout/content_bookmark_info.xml` with the following contents:

```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:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:padding="5dp">

  <ImageView
      android:id="@+id/photo"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_marginEnd="5dp"
      android:layout_marginRight="5dp"
      android:adjustViewBounds="true"
      android:maxWidth="200dp"
      android:scaleType="fitStart"
```

```xml
        android:src="@drawable/default_photo"/>

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="vertical">

      <TextView
          android:id="@+id/title"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:ellipsize="end"
          android:textColor="#ff000000"
          android:textSize="14sp"
          android:textStyle="bold"
          tools:text="Place Title"/>

      <TextView
          android:id="@+id/phone"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:ellipsize="end"
          android:maxLines="1"
          android:textColor="#ff7f7f7f"
          android:textSize="12sp"
          tools:text="555-121-1212"/>

    </LinearLayout>
  </LinearLayout>
```

You use a horizontal `LinearLayout` to wrap the place image and another vertical `LinearLayout` for the place details. You'll load this layout from your `InfoWindowAdapter` class and populate the `ImageView` and both `TextViews`.

Create a new package named `adapter` and then a new Kotlin class named `BookmarkInfoWindowAdapter.kt` within the `adapter` package.

`BookmarkInfoWindowAdapter` is your custom `InfoWindowAdapter`.

Replace the contents of `BookmarkInfoWindowAdapter.kt` with the following:

```kotlin
// 1
class BookmarkInfoWindowAdapter(context: Activity) :
    GoogleMap.InfoWindowAdapter {

  // 2
  private val contents: View
```

```
    // 3
    init {
      contents = context.layoutInflater.inflate(
          R.layout.content_bookmark_info, null)
    }

    // 4
    override fun getInfoWindow(marker: Marker): View? {
      // This function is required, but can return null if
      // not replacing the entire info window
      return null
    }

    // 5
    override fun getInfoContents(marker: Marker): View? {
      val titleView = contents.findViewById<TextView>(R.id.title)
      titleView.text = marker.title ?: ""

      val phoneView = contents.findViewById<TextView>(R.id.phone)
      phoneView.text = marker.snippet ?: ""

      return contents
    }
  }
```

1. You declare `BookmarkInfoWindowAdapter` to take a single parameter representing the hosting activity. The class implements the `GoogleMap.InfoWindowAdapter` interface.

2. You declare the property `contents` to hold the contents view.

3. When the `GoogleMap` instantiates the adapter, you inflate **content_bookmark_info.xml** and save it to `contents`.

4. You override `getInfoContents()` and return `null` to indicate that you won't be replacing the entire info window.

5. You override `getInfoWindow()` and fill in the `titleView` and `phoneView` widgets on the layout.

Once this object is assigned, the map will call `getInfoWindow()` whenever it needs to display an info window for a particular marker.

Note that you're not providing an image for the `ImageView` at this point. The only information you are given in `getInfoWindow()` is the associated `Marker`, and it doesn't store the photo. This will be fixed soon, but for now you'll continue to hook up the window adapter.

Assigning the InfoWindowAdapter

In **MapsActivity.kt**, add the following line to `onMapReady()` after `map` is assigned:

```
map.setInfoWindowAdapter(BookmarkInfoWindowAdapter(this))
```

Here you assign your custom `InfoWindowAdapter` to `map`.

You no longer need to set the photo as the marker icon. In `displayPoiDisplayStep()`, remove the lines that create the `iconPhoto` variable. Then remove `setIcon()` from `MarkerOptions`. The entire body of `displayPoiDisplayStep()` should look like this:

```
map.addMarker(MarkerOptions()
    .position(place.latLng)
    .title(place.name as String?)
    .snippet(place.phoneNumber as String?)
)
```

Run the app and tap on any place. A default red balloon marker will be added. Tap on the marker to display the info window.
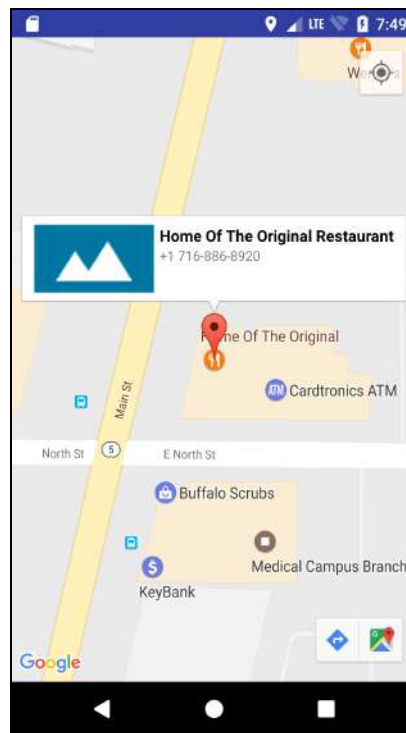
Marker tags

You'll finish off the `BookmarkInfoWindowAdapter` by adding the place image.

So how do you associate the image with the marker? There's several ways to tackle this problem, but they all involve using the tag property of the `Marker` object.

`Marker` provides the tag property as a means to associate the marker with data you are managing in the app. This could be a simple index into a list or dictionary, a full complex object, or in this case, a `Bitmap` object.

In `displayPoiDisplayStep()`, replace the call to `addMarker()` with this:

```
val marker = map.addMarker(MarkerOptions()
    .position(place.latLng)
    .title(place.name as String?)
    .snippet(place.phoneNumber as String?)

)
marker?.tag = photo
```

Here, `addMarker()` returns a `Marker` object and you assign it to `marker`. You then assign `photo` to the tag property.

Next, add the following lines to `getInfoContents` in **BookmarkInfoWindowAdapter.kt** before the `return contents` line:

```
val imageView = contents.findViewById<ImageView>(R.id.photo)
imageView.setImageBitmap(marker.tag as Bitmap?)
```

Since you assigned the place's image bitmap with the marker's tag property, when the map draws the info window contents it can set the `ImageView` to display the photo.

Run the app and tap on a place and the marker. This time the place photo will display in the info window.
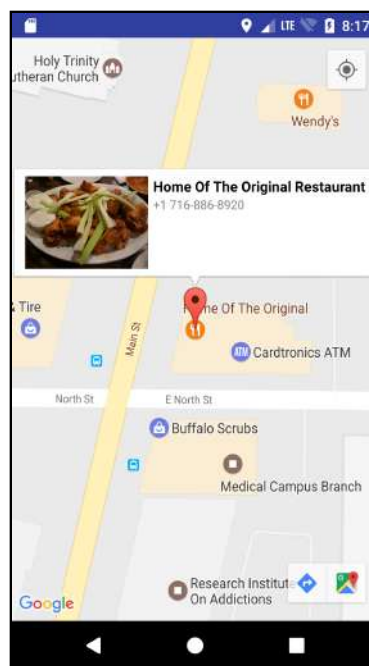
Where to go from here?

Give yourself a pat on the back for making it this far! You have everything you need to move on to the bookmarking feature.

In the next chapter, you'll learn how to save places to a local database and let the user edit place details.

Chapter 16: Saving Bookmarks with Room

By Tom Blankenship

Now that the user can tap on places to get an info window pop-up, it's time to give them a way to bookmark and edit a place.

In this chapter, you'll:

1. Learn about the **Room Persistence Library** and how it fits into the overall **Android Component Architecture** system.

2. Create a **Room** database to manage bookmarks.

3. Store bookmarks when the user taps on a map info window.

4. Learn about **LiveData** and use it to automatically update the view.

Getting started

If you were following along with your own app, open it, and keep going with it for this chapter. If not, don't worry! Locate the **projects** folder for this chapter, and open the PlaceBook app in the **starter** folder. If you use the starter app, don't forget to add your `google_maps_key` in **goole_maps_api.xml**. Check out Chapter 13 for more details about the Google Maps key. The first time you open the project, Android Studio takes a few minutes to set up your environment and update dependencies.

In the ListMaker app you used **Shared Preferences** to permanently store data. While Shared Preferences is a great way to manage simple key-value pairs, it's not designed to store large amounts of structured data.

For PlaceBook, you'll use the Room Persistence Library to store the bookmarks in a structured database. Room is built on top of **SQLite** and provides several advantages over Shared Preferences.

1. Works directly with **Plain Java Objects** (POJOs) with minimal effort.

2. Provides advanced search and sorting through SQL queries.

3. Manages relationships between different data types.

4. Efficiently stores large amounts of data.

Room overview

Before diving into the code, it's important to understand the three basic components of Room.

1. **Database**: This is the main interface to the underlying SQLite database. This component maintains one or more **Data Access Objects** (DAOs) and is annotated with the list of all **Entities** used by the database. A database class inherits from `RoomDatabase` and uses the `@Database` annotation.

2. **Entity**: This represents a single data type stored in the database. Room will create a table in the database for each entity, and the rows of the table represent individual entity items.

    Entities are defined as POJO classes using the `@Entity` annotation. All properties on the entity class are automatically defined as fields in the database unless you use the `@Ignore` annotation. At least one entity property should be designated as the primary key using the `@PrimaryKey` annotation.

3. **DAO**: Data Access Objects are the hero of Room. This is where you define the interface for accessing the database. DAOs should be the only part of your app that talks directly to the database. The database class must contain at least one abstract method that returns a DOA annotated interface.

The following diagram illustrates how these three components fit into the PlaceBook app.
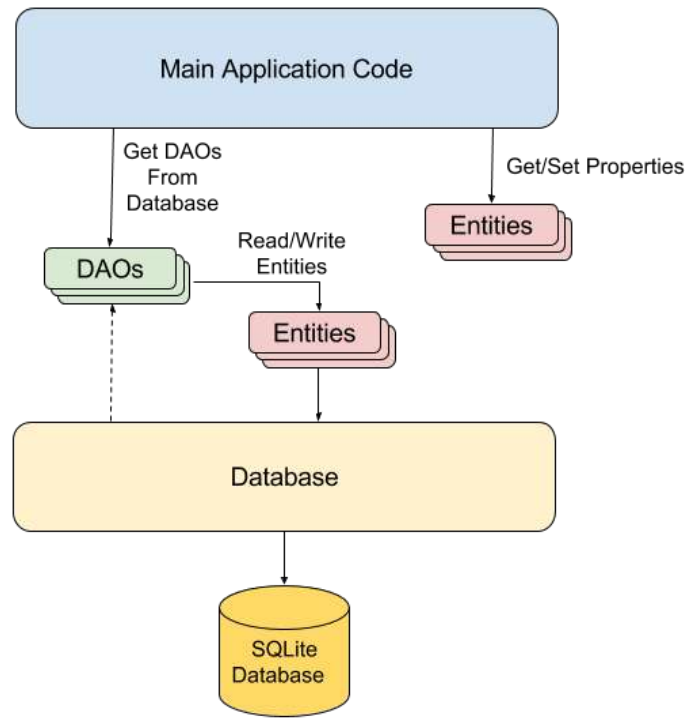
You'll learn more about how these three components work together as you proceed through this chapter.

Room and Android Architecture Components

Room is part of a larger set of libraries known as the **Android Architecture Components**. The other components are:

1. **Lifecycle management** - Provides several classes to help build lifecycle-aware objects.

2. **LiveData** - Holds data that can be observed for changes and respects lifecycles.

3. **ViewModel** - Manages view related data without being tied to configuration changes. This is the bridge between UI views and the rest of the app.

Don't worry about the details of these components right now, they will be covered in more detail as you build out the app.

PlaceBook architecture

Before creating your first Room classes, let's organize the application to achieve a clean overall architecture. The application can be separated into distinct areas of responsibility along these lines:

1. Data access and persistence (Room)

2. Data model (Model)

3. Data abstraction (Repository)

4. Business/Domain logic (ViewModel)

5. User interface (Activity/Fragment)

One key goal will be to ensure that communication only flows in one direction between these layers. This will result in a loosely coupled architecture that is easy to modify without causing side effects.

The overall architecture will look like this:
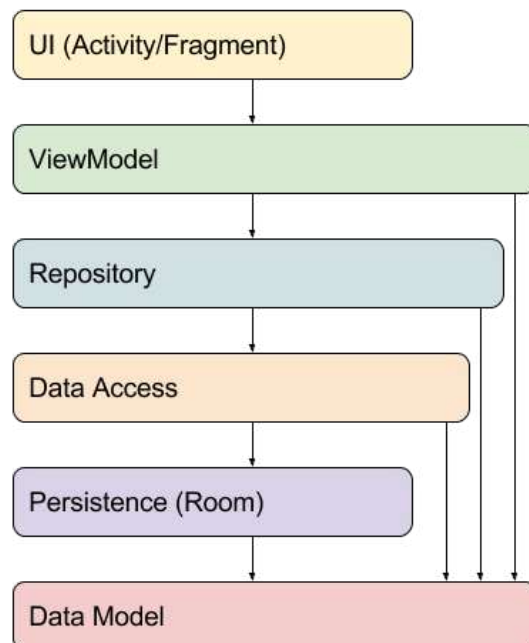
The arrows represent lines of communication and visibility. Notice that the UI layer is completely independent of all other layers except for the ViewModel. The ViewModel layer knows nothing about the UI layer.

As the rest of the app is built out, we'll be uncompromising about sticking with the communication flow shown in the above diagram. It will sometimes take a little more work to adhere strictly to this pattern, but the payoff for larger apps is worth the effort. Even for a small app such as PlaceBook, you can immediately recognize some benefits:

1. The way you store data in Room can be completely replaced with minimal impact. The only layers affected would be **Persistence** layer itself and its immediate parent, the **Data Access** layer.

2. The UI layer can be fully replaced without any other layer being any the wiser.

3. You can easily test all of the layers without any active UI running.

Development approach

Think about the architecture as a multi-layered cake. Have you ever seen somebody eat a cake one layer at a time? That would be a little odd! Likewise, you're not going to build out the app one layer at a time. You're going to take one slice at a time. Each slice may cut through all of the layers as you slowly build out the final product.

The PlaceBook Architecture Cake
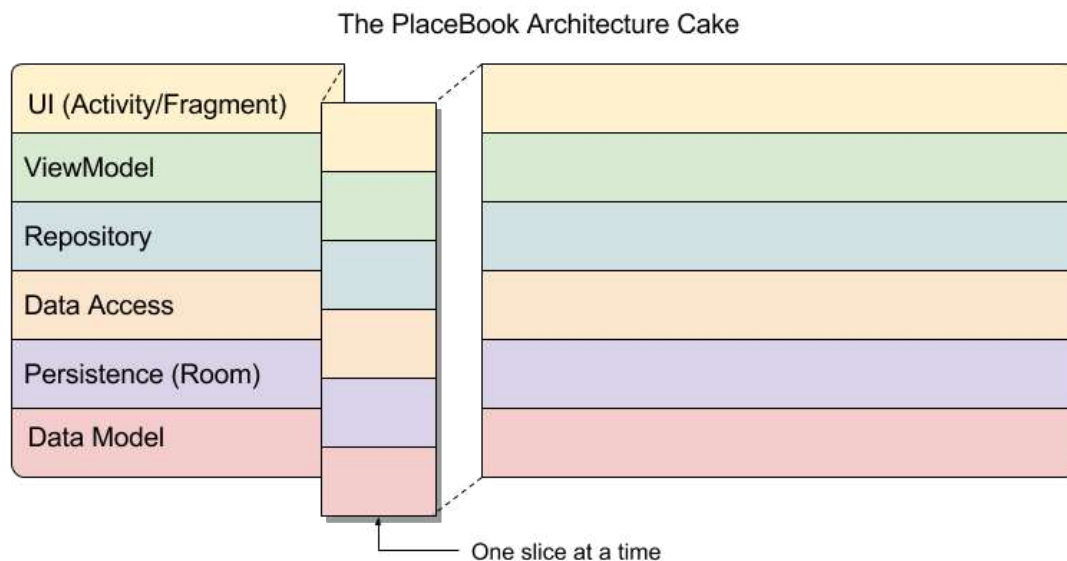
Create the following packages to help organize the project to match the architecture:

1. **db** - Data access and persistence. You'll keep the **Room Database** and **DAO** objects here.

2. **model** - Model objects. This will include all **Room Entities** as POJOs.

3. **repository** - Data abstraction. This provides a layer of abstraction for all data access.

4. **ui** - User interface. All views and view control logic belong here.

5. **viewmodel** - Business/Domain logic. Contains ViewModel classes that drive your user interface and app logic.

Drag `MapsActivity` from the root package to the ui package.

Accept the default settings from the dialog and click **Refactor**.
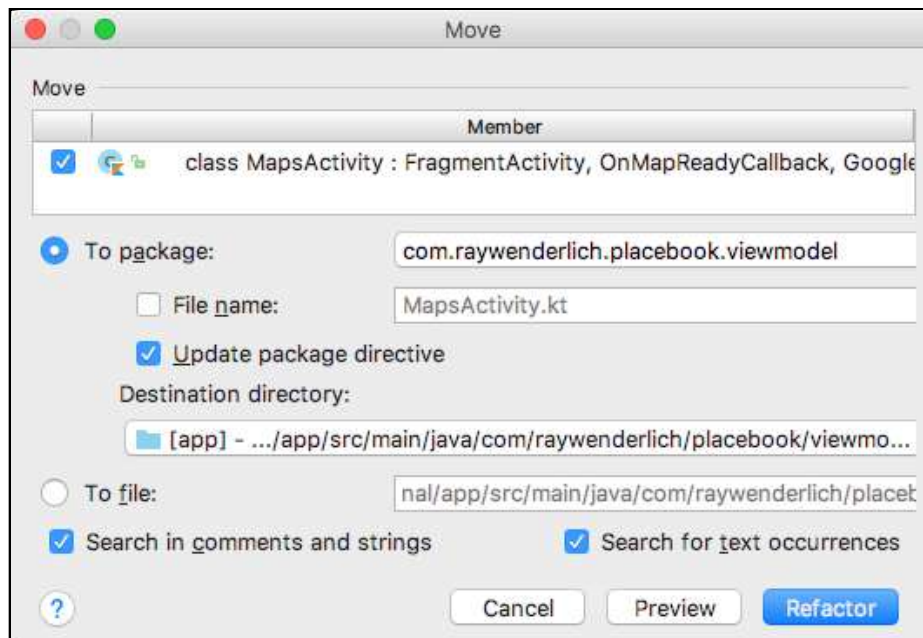
Your project tree-view should look like this:
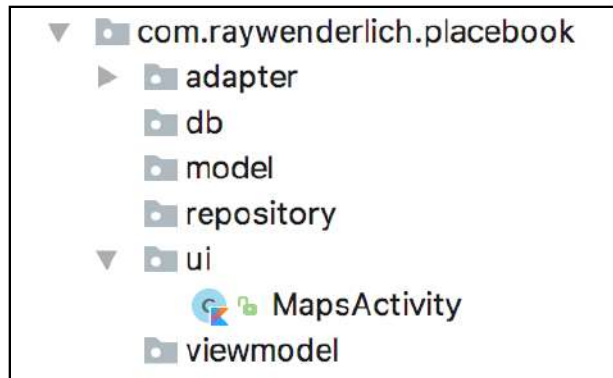
Adding the architecture components

The Architecture Components are provided as separate libraries from Google's Maven repository. The gradle file is already set up to use this repository, but you'll need to import the individual libraries.

First, define gradle extension properties for the library versions.

Open the project **build.gradle (Project: PlaceBook)** and add the following lines to the `ext` section:

```
architecture_version = '1.1.0'
room_version = '1.0.0'
```

Now you'll bring in the individual components.

Open the app **build.gradle (Module: app)** and add the following lines in the `dependencies` section.

```
// 1
implementation "android.arch.lifecycle:extensions:" +
    "$architecture_version"
// 2
implementation "android.arch.persistence.room:runtime:" +
    "$room_version"
// 3
annotationProcessor "android.arch.persistence.room:compiler:" +
    "$room_version"
// 4
annotationProcessor "android.arch.lifecycle:compiler:" +
    "$architecture_version"
// 5
kapt "android.arch.persistence.room:compiler:" +
    "$room_version"
```

Let's go through the above dependencies.

1. Adds the main LifeCycle class along with extensions such as LiveData.

2. Adds the Room library.

3. Adds the annotation processor for the Room library.

4. Adds the annotation processor for the lifecycle classes.

5. Adds the Kotlin annotation processor for the Room library.

Room classes

Now you're ready to add the basic classes required by Room. This will include the Entities, DAOs, and the Database. Behind the scenes, Room will take your class structure and do all of the hard work to create a SQLite database with tables and column definitions.

The database will be named `PlaceBookDatabase` and the model class will be named `Bookmark`. The following diagram will help visualize the process that Room uses to convert your classes into the underlying database:
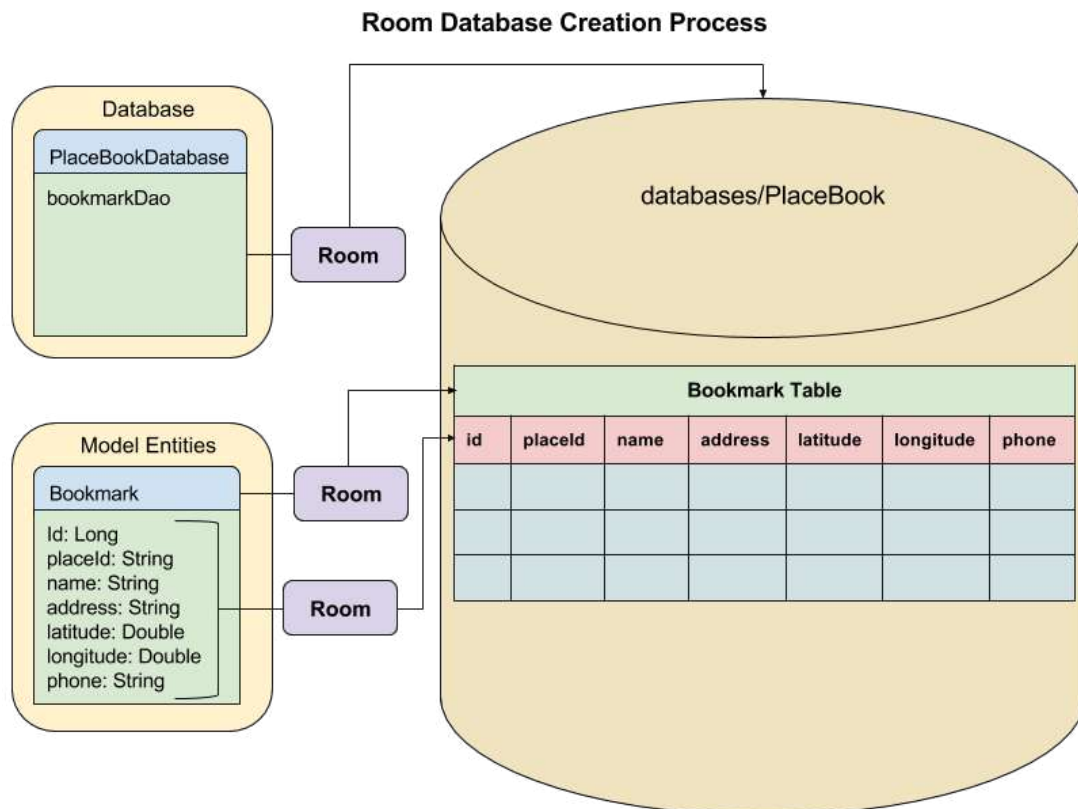
Entities

The PlaceBook application only requires a single entity type to store Bookmarks.

Create a new Kotlin file named **Bookmark.kt** in the **model** package and replace the contents with the following:

```
// 1
@Entity
// 2
data class Bookmark(
    // 3
    @PrimaryKey(autoGenerate = true) var id: Long? = null,
    // 4
    var placeId: String? = null,
    var name: String = "",
    var address: String = "",
    var latitude: Double = 0.0,
    var longitude: Double = 0.0,
    var phone: String = ""
)
```

1. The `@Entity` annotation tells Room that this is a database entity class.

    > **Note:** There are several attributes you can apply to the Entity annotation.
    >
    > `foreignKeys()` - List of ForeignKey constraints.
    >
    > `indices()` - List of indices to include on the table.
    >
    > `primaryKeys()` - List of primary key column names. Not required if using the `PrimaryKey` annotation.
    >
    > `tableName()` - Table name to use in the database. Defaults to class name.

2. The `Bookmark` class primary constructor is defined using arguments for all properties with default values defined. By defining default values you have the flexibility to construct a Bookmark with a partial list of properties.

    > **Note:** Room will look for arguments on the constructor and class properties when defining fields for the table.

3. The id property is defined using the `@PrimaryKey` annotation. There must be at least one of these per Entity class. The `autoGenerate` attribute tells Room to automatically generate incrementing numbers for this field. You can debate with your friendly neighborhood database administrator about the use of natural vs. surrogate primary keys, but nobody was ever fired for using a surrogate key!

4.   The rest of the fields are defined with default values.

When creating the new class **Bookmark.kt**, you might need to import these if Android Studio did not automatically add them for you:

```
import android.arch.persistence.room.Entity
import android.arch.persistence.room.PrimaryKey
```

DAOs

Next, you'll define the data access object that reads and writes from the database.

Create a new Kotlin file named **BookmarkDao.kt** in the **db** package and replace the contents with the following:

```
// 1
@Dao
interface BookmarkDao {

  // 2
  @Query("SELECT * FROM Bookmark")
  fun loadAll(): LiveData<List<Bookmark>>

  // 3
  @Query("SELECT * FROM Bookmark WHERE id = :arg0")
  fun loadBookmark(bookmarkId: Long): Bookmark

  @Query("SELECT * FROM Bookmark WHERE id = :arg0")
  fun loadLiveBookmark(bookmarkId: Long): LiveData<Bookmark>

  // 4
  @Insert(onConflict = IGNORE)
  fun insertBookmark(bookmark: Bookmark): Long

  // 5
  @Update(onConflict = REPLACE)
  fun updateBookmark(bookmark: Bookmark)

  // 6
  @Delete
  fun deleteBookmark(bookmark: Bookmark)
}
```

Note: When you add this code, you may get an error about the references to IGNORE and REPLACE. Place the cursor on IGNORE and press **Option+Return** on a Mac or **Ctrl+Enter** on Windows and select the android.arch.persistence.room.OnConflictStrategy.IGNORE option.

> Place the cursor on `REPLACE` and press **Option+Return** on a Mac or **Ctrl+Enter** on Windows and select the
> `android.arch.persistence.room.OnConflictStrategy.REPLACE` option.

`BookmarkDao` defines what would traditionally be known as the **CRUD** database operations. The CRUD operations consists of:

- *C* - Create. Create new objects in the database.

- *R* - Read. Read objects from the database.

- *U* - Update. Update objects in the database.

- *D* - Delete. Delete objects in the database.

All access to the Bookmark data will be through this class. You can name the methods anything you like, but the real power is in the annotations. The `@Query`, `@Insert`, `@Update`, and `@Delete` annotations provide Room with valuable information. Room uses this to generate the code that automatically converts your data entities to rows in the database and vice versa.

There are several new concepts introduced with this class; let's explore them in detail.

1. The `@Dao` annotation tells Room that this is a **Data Access Object**. DAO classes must be either interfaces or abstract classes. Room will create the concrete class at runtime based on the method definitions you define.

2. `loadAll()` uses the `@Query` annotation to define a SQL statement to read all bookmarks from the database and return them as a `List` of `Bookmarks`.

> **Note:** SQL stands for Structured Query Language and is a well-known method for working with relational databases such as SQLite. You won't need to know much SQL to build out PlaceBook. If you want to learn more about SQL and specifically the syntax used for SQLite, take a look at https://sqlite.org/lang.html.

You're wrapping the returned `List` with `LiveData`, which provides a couple of advantages:

`LiveData` objects can be observed by another object. `LiveData` will notify any observers when the data changes. This provides a great way to keep user interface elements up to date when items change in the database.

LiveData objects do their work in a background thread. By default, Room won't allow you to make calls to DAO methods on the main thread. By returning LiveData objects, your method becomes an asynchronous query, and there is no restriction to calling it from the main thread.

3. Here the @Query annotation is used to define loadBookmark(). This method loads a single Bookmark based on the bookmark id. Room binds the arguments from your method to the :arg# strings in the SQL statement. You replace # with the index number of the argument. This method returns a single Bookmark object.

    You also define an asynchronous version named loadLiveBookmark that returns a LiveData wrapper around a single Bookmark.

    > **Note:** A bug in the current Kotlin implementation requires the use of the :arg# syntax. Once this bug is fixed, you'll be able to use the actual names of the method arguments.

4. The @Insert annotation is used to define insertBookmark(). This saves a single Bookmark to the database and returns the new primary key id associated with the new bookmark. The onConflict attribute of the @Insert annotation defines what happens if there is an existing record with the same primary key. This is not a concern for PlaceBook, as you're using an autogenerated primary key.

    > **Note:** To learn more about conflict options, please see this page: https:// sqlite.org/lang_conflict.html.

5. The @Update annotation is used to define updateBookmark(). This updates a single Bookmark in the database using the passed in bookmark argument. The onConflict attribute of the @Update annotation is set to REPLACE so that the existing bookmark in the database will be replaced with the new bookmark data.

6. Finally, the @Delete annotation is used to define deleteBookmark(). This deletes an existing bookmark based on the passed in Bookmark.

Database

The last piece needed to complete the Room classes is the Database.

Create a new Kotlin file named **PlaceBookDatabase.kt** in the **db** package and replace the contents with the following:

```kotlin
// 1
@Database(entities = arrayOf(Bookmark::class), version = 1)
abstract class PlaceBookDatabase : RoomDatabase() {
  // 2
  abstract fun bookmarkDao(): BookmarkDao
  // 3
  companion object {
    // 4
    private var instance: PlaceBookDatabase? = null
    // 5
    fun getInstance(context: Context): PlaceBookDatabase {
      if (instance == null) {
        // 6
        instance = Room.databaseBuilder(
            context.applicationContext,
            PlaceBookDatabase::class.java,
            "PlaceBook").build()
      }
      // 7
      return instance as PlaceBookDatabase
    }
  }
}
```

1. The `@Database` annotation is used to identify a Database class to Room. `entities` is a required attribute on the `@Database` annotation and defines an array of all entities used by the database.

    Your database class must be abstract and inherit from `RoomDatabase`.

2. The abstract method `bookmarkDao` is defined to return a DAO interface. Note that there can be as many DAOs as you would like, but PlaceBook only needs one.

    This is all that's required for the `Database` class. The rest of the code is added so that the `Database` interface object can be used as a singleton. This is recommended by Google because spinning up new `Database` objects can be an expensive operation.

3. Define a `companion object` on `PlaceBookDatabase`.

4. Define the one and only `instance` variable on the companion object.

5. Define `getInstance()` to take in a `Context` and return the single `PlaceBookDatabase` instance.

6. If this is the first time `getInstance` is being called, create the single
    `PlaceBookDatabase` instance. `Room.databaseBuilder()` is used to create a Room
    Database based on the abstract `PlaceBookDatabase` class.

7. Return the `PlaceBookDatabase` instance.

> **Note:** Now that you have the database defined, you can test out a great feature of
> Room. It verifies the SQL in your @Query annotations at compile time. If you have
> an error in the SQL syntax, such as referring to a non-existent table name, it will
> give you an error. It will also warn if the return type on your method doesn't
> match the return type of your SQL statement.
>
> Test this out by changing `Bookmark` to `Bookmarks` in one of the @Query strings in
> **Bookmark.kt**, and then rebuild the project. This will result in a compile error that
> says "Error:There is a problem with the query: [SQLITE_ERROR] SQL error or
> missing database (no such table: Bookmarks)". If you've ever worked with Android
> SQLite databases before Room was available, you'll realize what a big help this is.
> Room provides a safety net to prevent common typos in your SQL statements.

Creating the Repository

Your basic Room classes are ready to go, but let's add one more layer of abstraction
between Room and the rest of the application code. By doing this, you make it easy to
change out how and where the app data is stored. This abstraction layer will be
provided using a **Repository** pattern. The repository is a generic store of data that can
manage multiple data sources but expose one unified interface to the rest of the
application.

Although the repository in PlaceBook will have a single data source, the `BookmarkDao`
class, the power is that it could utilize multiple data sources or swap out a data source
completely without affecting other parts of the application. The app you'll build in
Section 4 will make full use of the Repository pattern.

Create a Kotlin file named **BookmarkRepo.kt** in the **repository** package and replace
the contents with the following:

```kotlin
// 1
class BookmarkRepo(private val context: Context) {
  // 2
  private var db: PlaceBookDatabase =
      PlaceBookDatabase.getInstance(context)
  private var bookmarkDao: BookmarkDao = db.bookmarkDao()
  // 3
```

```kotlin
  fun addBookmark(bookmark: Bookmark): Long? {
    val newId = bookmarkDao.insertBookmark(bookmark)
    bookmark.id = newId
    return newId
  }
  // 4
  fun createBookmark(): Bookmark {
    return Bookmark()
  }
  // 5
  val allBookmarks: LiveData<List<Bookmark>>
    get() {
      return bookmarkDao.loadAll()
    }
}
```

1. Define the `BookmarkRepo` class with a constructor that defines a single property named `context`.

2. Two properties are defined that `BookmarkRepo` will use for its data source. The first is the `PlaceBookDatabase` singleton instance, and the second is the `DAO` object from `PlaceBookDatabase`. Note that the `bookmarkDao` property must follow `db` as it depends on `db` being created first.

3. Create `addBookmark()` to allow a single `Bookmark` to be added to the repo. This method returns the **unique id** of the newly saved `Bookmark` or null if the `Bookmark` could not be saved. This method uses `insertBookmark()` on `bookmarkDao` to add the `Bookmark` to the database. It then assigns the `newId` to the Bookmark and returns the `newId` to the caller.

4. Add `createBookmark()` as a helper method to return a freshly initialized `Bookmark` object. In this case you return a simple `Bookmark` object. Having your application code get all new objects from the repository gives the repository an opportunity to apply special initialization code if necessary.

5. Create the `allBookmarks` property that returns a `LiveData` list of all `Bookmarks` in the Repository. You call `loadAll()` on the `bookmarkDao` and return the results to the caller.

The ViewModel

The **ViewModel** layer serves as the intermediary between your application views and the data provided by the **BookmarkRepo**. The ViewModel will drive the UI based on the repository data and will update the repository data based on user interactions.

You'll typically have one ViewModel for each view (Activity or Fragment) in your application. The naming convention used for ViewModel classes is to simply append ViewModel to the view class prefix.

Create a Kotlin file named **MapsViewModel.kt** in the **viewmodel** package to go along with the **MapsActivity**. Replace the contents with the following:

```kotlin
// 1
class MapsViewModel(application: Application) :
    AndroidViewModel(application) {

  private val TAG = "MapsViewModel"
  // 2
  private var bookmarkRepo: BookmarkRepo = BookmarkRepo(
      getApplication())
  // 3
  fun addBookmarkFromPlace(place: Place, image: Bitmap) {
    // 4
    val bookmark = bookmarkRepo.createBookmark()
    bookmark.placeId = place.id
    bookmark.name = place.name.toString()
    bookmark.longitude = place.latLng.longitude
    bookmark.latitude = place.latLng.latitude
    bookmark.phone = place.phoneNumber.toString()
    bookmark.address = place.address.toString()
    // 5
    val newId = bookmarkRepo.addBookmark(bookmark)

    Log.i(TAG, "New bookmark $newId added to the database.")
  }
}
```

1. When creating a `ViewModel` it should inherit from `ViewModel` or `AndroidViewModel`. Inheriting from `AndroidViewModel` allows you to include the Application context which is needed when creating the `BookmarkRepo`.

2. Create the `BookmarkRepo` object.

3. Declare the method `addBookmarkFromPlace` that takes in a Google `Place` and a `Bitmap` image.

4. Use `BookmarkRepo.createBookmark()` to create an empty `Bookmark` object and then fill it in using the Place data.

5. Finally, save the `Bookmark` to the repository and print out an info message to verify that the bookmark was added.

Adding bookmarks

You have everything in place for adding bookmarks to the database, now you just need to detect when the user taps on a place info window.

In **MapsActivity.kt**, add the following property at the top of the class:

```
private lateinit var mapsViewModel: MapsViewModel
```

You're declaring a private member to hold the **MapsViewModel**. This will be initialized when the map is ready.

Add a new private method named `setupViewModel`:

```
private fun setupViewModel() {
  mapsViewModel =
      ViewModelProviders.of(this).get(MapsViewModel::class.java)
}
```

You may be wondering about the odd syntax for creating the `MapsViewModel`. A big benefit of using the ViewModel class is that it is aware of lifecycles. In this case, `ViewModelProviders` will create a new `mapsViewModel` only the first time the Activity is created. If a configuration change happens, such as a screen rotation, `ViewModelProviders` will return the previously created `MapsViewModel`.

Add a call to `setupViewModel()` to the end of `onMapReady()`:

```
setupViewModel()
```

`onMapReady()` will continue to grow as you add new capabilities to `MapsActivity`. This is a good time to do some quick cleanup before moving on.

Create a new method named `setupMapListeners` and move the calls to `map.setInfoWindowAdapter` and `map.setOnPoiClickListener` into this new method:

```
private fun setupMapListeners() {
  map.setInfoWindowAdapter(BookmarkInfoWindowAdapter(this))
  map.setOnPoiClickListener {
    displayPoi(it)
  }
}
```

Add a call to `setupMapListeners()` before the call to `setupViewModel()` in `onMapReady()`.

The new version of `onMapReady()` should now match this:

```
override fun onMapReady(googleMap: GoogleMap) {
  map = googleMap
```

```
    setupMapListeners()
    setupViewModel()
    getCurrentLocation()
}
```

The next step is to respond to the user tapping on an info window and then call
`MapsViewModel.addBookmarkFromPlace()` with the `Place` and `Bitmap` objects.

Houston, we have a problem!

As the code is now, when you add a marker, you're setting the marker tag to the place
image only. You don't have access to the original `Place` object. What's needed is a way
to set both the full `Place` object and the `Bitmap` image as the `Marker` tag. This can be
solved by creating a private class to hold both pieces of information.

Add the following internal class to the bottom of the `MapsActivity` class:

```
class PlaceInfo(val place: Place? = null,
    val image: Bitmap? = null)
```

This defines a class with two properties to hold a `Place` and a `Bitmap`.

In `displayPoiDisplayStep()`, replace the line that sets the `marker.tag` with this line:

```
marker?.tag = PlaceInfo(place, photo)
```

Now the marker tag holds the full `place` object and the associated bitmap `photo`.

In **BookmarkInfoWindowAdapter.kt**, update the `setImageBitmap` call in
`getInfoContents()` to this:

```
imageView.setImageBitmap((marker.tag as
    MapsActivity.PlaceInfo).image)
```

You're casting the `marker.tag` to a `PlaceInfo` object and then accessing the image
property to set it as the `imageView` bitmap. Now you'll handle the action when the user
taps the info window for a place. Add the following method to **MapsActivity.kt**:

```
private fun handleInfoWindowClick(marker: Marker) {
  val placeInfo = (marker.tag as PlaceInfo)
  if (placeInfo.place != null && placeInfo.image != null) {
    mapsViewModel.addBookmarkFromPlace(placeInfo.place,
        placeInfo.image)
  }
  marker.remove()
}
```

This method will handle taps on a `place` info window. You get the `placeInfo` from the
`marker.tag`, verify that the data is not `null`, and then call `mapsViewModel.`

addBookmarkFromPlace() to add the `place` to the repository. Finally, you remove the marker from the map.

Add the following line to the end of setupMapListeners():

```
map.setOnInfoWindowClickListener {
  handleInfoWindowClick(it)
}
```

Here you set up a listener to call handleInfoWindowClick() whenever the user taps an info window.

Now, whenever the user taps a place info window, it will call handleInfoWindowClick() which will in turn call mapsViewModel.addBookmarkFromPlace() and add a bookmark to the database.

Build and run the app.

Tap on a place so it shows a marker. Tap on the marker, and then tap on the info window.
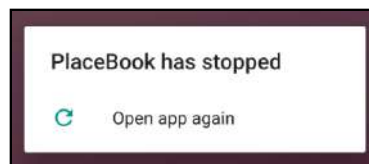
Ok, that didn't turn out exactly as planned! It was supposed to trigger a call to addBookmarkFromPlace() and add the bookmark to the database. Check the Logcat window and see if you can identify the problem.

You should have seen the info message "New bookmark 1 added to the database.", but instead the following exception was printed:

```
java.lang.IllegalStateException: Cannot access database on the main
thread since it may potentially lock the UI for a long period of time.
```

This exception is being thrown on the call to addBookmarkFromPlace() and as the message explains, it's because the database cannot be accessed on the main thread. There are several ways to fix this problem, and the easiest would be to configure Room to allow database access on the main thread. This would only be a stop-gap measure though. The proper solution is make sure that addBookmarkFromPlace() runs in a background thread.

One way to attack the problem is to use **AsyncTask**, but a simpler method is to use Kotlin **Coroutines**.

Coroutines

Coroutines make asynchronous programming easier by hiding many of the underlying complications. This frees you to think about your code in a more traditional sequential fashion that is easier to comprehend. You'll learn more about Coroutines in future chapters, but for now you only need to know about the **launch** coroutine builder.

> **Note:** If you aren't familiar with asynchronous programming concepts, it's just a fancy way to say that more than one thing is happening at a time. Normally, your code executes in a serial fashion on the main thread of execution. With asynchronous programming, multiple code paths are executed simultaneously by using background threads. To learn more about asynchronous programming with Android, please check out the following link: https://developer.android.com/guide/components/processes-and-threads.html

A coroutine represents a suspendable computation. Suspendable means that the computation may be *suspended* without stopping the main execution thread.

The **launch** coroutine builder is used to start a coroutine. It takes in a coroutine context and is following by a block of code known as a suspending lambda expression. Kotlin provides a **CommonPool** context that automatically dispatches your lambda expression in a background thread.

Having the call to `addBookmarkFromPlace()` run in the background is as easy as wrapping it with the launch coroutine builder.

Adding Coroutine libraries

Coroutine support is provided as a separate library and must be added to the project dependencies before being used.

Open the app **build.gradle (Module: app)** and add the following line in the `dependencies` section.

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:0.19.3"
```

> **Note:** After making the above change to your gradle file, don't forget to click **Sync Now** so that Gradle loads the new dependencies.

Creating a Coroutine

Open **MapsActivity** and replace the call to addBookmarkFromPlace in handleInfoWindowClick() with the following:

```
launch(CommonPool) {
  mapsViewModel.addBookmarkFromPlace(placeInfo.place,
      placeInfo.image)
}
```

You use the launch coroutine builder to build a coroutine as a lambda expression. The CommonPool context is used so the code inside the lambda expressions runs in the background.

Build and run the app again and repeat the process of tapping an info window. Check the Logcat window and this time you'll see the "New bookmark 1 added to the database." message.

```
I/MapsViewModel: New bookmark 1 added to the database.
```

Observing database changes

You've made a huge step forward by saving bookmarks to the database, but the user has no way of identifying places that have been bookmarked. The goal is to have the UI automatically reflect the current state of the bookmark database. This is where your use of the ViewModel starts to pay off.

You're going to add a LiveData property to the ViewModel and then observe this LiveData from the MapsActivity. You'll display blue colored markers for all bookmarks stored in the database.

ViewModel changes

Add the following internal class to **MapsViewModel.kt**:

```
data class BookmarkMarkerView(
    var id: Long? = null,
    var location: LatLng = LatLng(0.0, 0.0))
```

> **Note:** If Android Studio can't resolve LatLng, add import com.google.android.gms.maps.model.LatLng to the top of **MapsViewModel.kt**.

This will hold the information needed by the view to plot a marker for a single
bookmark.

Add the following property at the top of **MapsViewModel**:

```
private var bookmarks: LiveData<List<BookmarkMarkerView>>?
    = null
```

Here you are defining a `LiveData` object that wraps a list of `BookmarkMarkerView` objects.

Add the following method to **MapsViewModel**:

```
private fun bookmarkToMarkerView(bookmark: Bookmark):
    MapsViewModel.BookmarkMarkerView {
  return MapsViewModel.BookmarkMarkerView(
      bookmark.id,
      LatLng(bookmark.latitude, bookmark.longitude))
}
```

This is a helper method that converts a `Bookmark` object into a `BookmarkMarkerView`
object. This will be used by the next method.

Now, add the following method:

```
private fun mapBookmarksToMarkerView() {
  // 1
  val allBookmarks = bookmarkRepo.allBookmarks
  // 2
  bookmarks = Transformations.map(allBookmarks) { bookmarks ->
    val bookmarkMarkerViews = bookmarks.map { bookmark ->
      bookmarkToMarkerView(bookmark)
    }
    bookmarkMarkerViews
  }
}
```

This method maps the `LiveData<List<Bookmark>>` objects provided by **BookmarkRepo**
into `LiveData<List<BookmarkMarkerView>>` objects that can be used by **MapsActivity**.
Although you could remove the mapping and return the `LiveData<List<Bookmark>>`
directly to `MapsActivity`, you don't want to expose `MapsActivity` to the details of the
underlying `Bookmark` object.

1. Assign a local variable `allBookmarks` to the `BookmarkRepo.allBookmarks` LiveData
    object.

2. Use the `Transformations` class to dynamically map `Bookmark` objects into
    `BookmarkMarkerView` objects as they get updated in the database. This is a
    convenient way to make changes to LiveData before it gets passed along to any
    observers.

You use a map method to iterate over the bookmarks and call the previously defined `bookmarkToMarkerView()` on each one.

Add the following method to `MapsViewModel`:

```
fun getBookmarkMarkerViews() :
    LiveData<List<BookmarkMarkerView>>? {
  if (bookmarks == null) {
    mapBookmarksToMarkerView()
  }
  return bookmarks
}
```

This method returns the `LiveData` object that will be observed by `MapsActivity`. `bookmarks` will be `null` the first time this method is called. If it is `null`, then it calls `mapBookmarksToMarkerView()` to set up the initial mapping.

That's all of the changes required in `MapsViewModel`.

MapsActivity changes

Now you're ready to update `MapsActivity` to listen for changes in the view model.

Open **MapsActivity** and add the following method:

```
private fun addPlaceMarker(
    bookmark: MapsViewModel.BookmarkMarkerView): Marker? {

  val marker = map.addMarker(MarkerOptions()
      .position(bookmark.location)
      .icon(BitmapDescriptorFactory.defaultMarker(
        BitmapDescriptorFactory.HUE_AZURE))
      .alpha(0.8f))

  marker.tag = bookmark

  return marker
}
```

This is a helper method that adds a single blue marker to the map based on a `BookmarkMarkerView`. This is very similar to the code that adds a marker when tapping on a place. The main difference is that it doesn't use the default red color.

Now, add the following method:

```
private fun displayAllBookmarks(
    bookmarks: List<MapsViewModel.BookmarkMarkerView>) {
  for (bookmark in bookmarks) {
    addPlaceMarker(bookmark)
  }
}
```

This method walks through a list of `BookmarkMarkerView` objects and calls `addPlaceMarker()` for each one.

Add the following method to **MapsActivity**:

```
private fun createBookmarkMarkerObserver() {
  // 1
  mapsViewModel.getBookmarkMarkerViews()?.observe(
      this, android.arch.lifecycle
      .Observer<List<MapsViewModel.BookmarkMarkerView>> {
        // 2
        map.clear()
        // 3
        it?.let {
          displayAllBookmarks(it)
        }
      })
}
```

This method observes changes to the `BookmarkMarkerView` objects from the `MapsViewModel` and updates the view when they change.

1. Start by using `getBookmarkMarkerViews()` on `MapsViewModel` to retrieve a `LiveData` object. To be notified when the underlying data changes on the LiveDat object, you call the `observe` method. The first argument is `this`, and it represents the LifeCycle Owner. The second argument is a new `Observer` lambda expression to process the updated bookmarks. The lambda expression will run each time the data changes.

2. Once you have the updated data, clear all existing markers on the map.

3. Call `displayAllBookmarks()` passing in the list of updated `BookmarkMarkerView` objects.

The only item left is to call `createBookmarkMarkerObserver()` when setting up the model view.

Add the following line to the end of `setupViewModel()`:

```
createBookmarkMarkerObserver()
```

Build and run the app. If you previously added some places to the database by tapping on the info windows, you'll see blue markers appear on the map.

Add a new bookmark for another place by tapping on it, and then tapping on the info window. You'll notice that the map automatically updates to display the new blue marker for the saved bookmark.
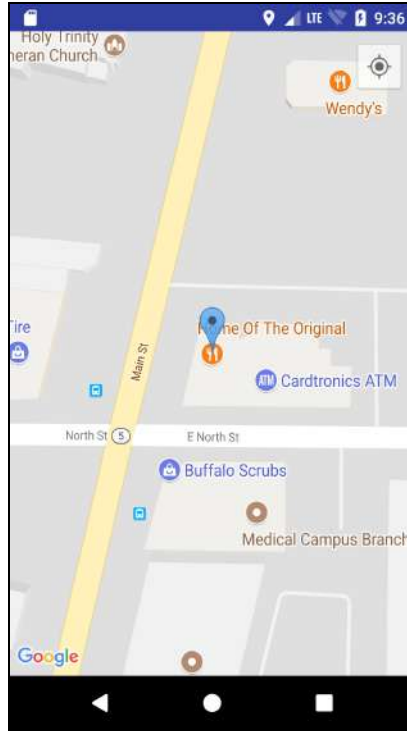
This happened even though you didn't make any direct calls to display markers when the application started!

The following illustrates how this is working:
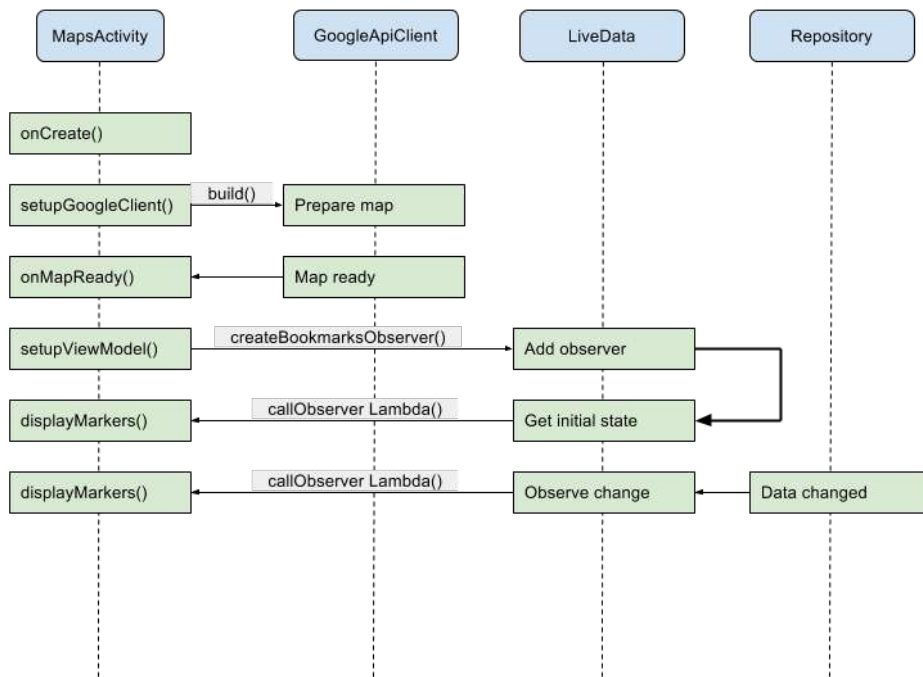
When you first observe a LiveData, it will call your observer immediately with the current set of data. From then on, the observer will be notified anytime the underlying data changes.

Where to go from here?

There is one problem with this new implementation: if you tap on any of the blue markers the app will crash. Can you guess why? Never fear, we'll fix this crash in the next chapter as well as add some new features to `MapsActivity` and give the user the ability to edit bookmarks.

Chapter 17: Detail Activity

By Tom Blankenship

In this chapter you'll add the ability to edit bookmarks. This will involve creating a new activity to display the bookmark details with editable fields.

Getting started

If you were following along with your own app, open it and copy this resource from the **starter** project into your project:

- **res/drawable/ic_action_done.png**

Make sure to copy the files from all of the drawable folders, including everything with the `.hdpi`, `.mdpi`, `.xhdpi` and `.xxhdpi` extensions.

If you would rather use the **starter**, locate the **projects** folder for this chapter and open the PlaceBook app under the **starter** folder. If you do use the starter app, don't forget to add your `google_maps_key` in **google_maps_api.xml**. Check out Chapter 13 for more details about the Google Maps key. The first time you open the project, Android Studio takes a few minutes to set up your environment and update dependencies.

Fixing the info window

Before moving on, you'll track down and fix that pesky bug left over from the last chapter. The app currently crashes when tapping on a blue marker. The desired behavior should be:

- If the user taps a new place, it shows a red marker and the info window. If the user then taps on the info window, a bookmark is saved to the database and the marker turns blue.

- If the user taps on a blue marker, it displays the saved bookmark info, including the image.

Build and run the app again, and tap on an existing bookmark icon. After the app crashes, check out Logcat. Look for the most recent stack trace line that has your app's package name. You'll find the following line:

```
com.raywenderlich.placebook.adapter.BookmarkInfoWindowAdapter.getInfoCont
ents
```

```
java.lang.ClassCastException: com.raywenderlich.placebook.viewmodel.MapsViewModel$BookmarkMarkerView cannot be cast to
    at com.raywenderlich.placebook.adapter.BookmarkInfoWindowAdapter.getInfoContents(BookmarkInfoWindowAdapter.kt:33)
```

The error is a **ClassCastException** informing you that a `BookmarkMarkerView` cannot be cast to a `MapActivityPlaceInfo` class.

What's going on here? Click on the blue link for **BookmarkAdapter.kt** and it will take you to the offending line of code:

```
imageView.setImageBitmap(
    (marker.tag as MapsActivity.PlaceInfo).image)
```

The problem is that this code assumes that `marker.tag` contains an object of type `MapsActivity.PlaceInfo`, but that's not always the case. A marker can now represent two types of places: one is a temporary place that hasn't been bookmarked yet, and the other is a place that has an existing bookmark.

To fix this, you'll update the code to take a different action based on the marker tag type.

Open **BookmarkInfoWindowAdapter.kt** and replace the line in `getInfoContents()` that calls `setImageBitmap()` with the following:

```
when (marker.tag) {
  // 1
  is MapsActivity.PlaceInfo -> {
    imageView.setImageBitmap(
```

```
          (marker.tag as MapsActivity.PlaceInfo).image)
    }
    // 2
    is MapsViewModel.BookmarkMarkerView -> {
      var bookMarkview = marker.tag as
          MapsViewModel.BookmarkMarkerView
      // Set imageView bitmap here
    }
  }
```

The `when` statement is used to run conditional code based on the class type of `marker.tag`.

1. If `marker.tag` is a `MapsActivity.PlaceInfo`, you set the `imageView` bitmap directly from the `PlaceInfo.image` object.

2. If `marker.tag` is a `MapsViewModel.BookmarkMarkerView`, you set the `imageView` bitmap from the `BookmarkMarkerView`.

The only problem is that `BookmarkMarkerView` doesn't contain a bookmark image, because you've never saved images with the bookmarks.

Saving an image

Although you can add an image directly to the `Bookmark` model class and let the Room library save it to the database, it's not best practice to store large chunks of data in the database. A better method is to store the image as a file that is linked to the record in the database.

Android doesn't provide a simple way to save images to a file, so you'll first create a new image utility class, and add a method to save an image to a file.

Create a new package named **util**, and then a new Kotlin class named **ImageUtils.kt**, inside the **util** package.

Replace the contents of **ImageUtils.kt** with the following:

```
// 1
object ImageUtils {
  // 2
  fun saveBitmapToFile(context: Context, bitmap: Bitmap,
      filename: String) {
    // 3
    val stream = ByteArrayOutputStream()
    // 4
    bitmap.compress(Bitmap.CompressFormat.PNG, 100, stream)
    // 5
    val bytes = stream.toByteArray()
    // 6
    ImageUtils.saveBytesToFile(context, bytes, filename)
```

```
    }
    // 7
    private fun saveBytesToFile(context: Context, bytes:
        ByteArray, filename: String) {
      val outputStream: FileOutputStream
      // 8
      try {
        // 9
        outputStream = context.openFileOutput(filename,
            Context.MODE_PRIVATE)
        // 10
        outputStream.write(bytes)
        outputStream.close()
      } catch (e: Exception) {
        e.printStackTrace()
      }
    }
  }
```

1. `ImageUtils` is declared as an object, so it behaves like a singleton. This lets you directly call the methods within `ImageUtils` without creating a new `ImageUtils` object each time.

2. `saveBitmapToFile()` takes in a `Context`, `Bitmap` and `String` object `filename`, and saves the `Bitmap` to permanent storage.

3. `ByteArrayOutputStream` is created to hold the image data.

4. You write the image `bitmap` to the `stream` object using the lossless PNG format. Note that the second parameters is a quality setting, but it's ignored for the PNG format.

5. the `stream` is converted into an array of bytes.

6. `saveBytesToFile()` is called to write the bytes to a file.

7. `saveBytesToFile()` takes in a `Context`, `ByteArray`, and a `String` object `filename` and saves the `bytes` to a file.

8. The next few calls may throw exceptions so they're wrapped in a try/catch to prevent a crash.

9. `openFileOutput` is used to open a `FileOutputStream` using the given `filename`. The `Context.MODE_PRIVATE` flag causes the file to be written in the private area where only the PlaceBook app can access it.

10. The `bytes` are written to the `outputStream` and then the stream is closed.

Now that you have the `saveBitmapToFile` utility method, you can give the `Bookmark` object the ability to save a bitmap image for itself. This method will automatically generate a filename for the bitmap that matches the bookmark ID.

Open the **Bookmark** class in the **model** package and add the following code as the body of the class:

```
{
  // 1
  fun setImage(image: Bitmap, context: Context) {
    // 2
    id?.let {
      ImageUtils.saveBitmapToFile(context, image,
        generateImageFilename(it))
    }
  }
  //3
  companion object {
    fun generateImageFilename(id: Long): String {
      // 4
      return "bookmark$id.png"
    }
  }
}
```

1. `setImage()` provides the public interface for saving an image for a `Bookmark`.

2. If the `bookmark` has an `id`, then the image is saved to a file. The filename incorporates the bookmark ID to make sure it's unique.

3. `generateImageFilename()` is placed in a companion object so it will be available at the class level. This allows another object to load an image without having to load the bookmark from the database.

4. `generateImageFilename()` returns a filename based on a `Bookmark` ID. It uses a simple algorithm that simply appends the bookmark ID to the word "bookmark". For example, for bookmark ID 5, the associated image will be named **bookmark5.png**. Since you can always infer the bookmark image filename from the bookmark ID, there is no need to save the filename as a separate field in the database.

Adding the image to the bookmark

Now you need to set the image for a bookmark when it's added to the database.

Open **MapsViewModel.kt** and add the following line in `addBookmarkFromPlace()` after the call to set `bookmarkRepo.addBookmark()`.

```
bookmark.setImage(image, getApplication())
```

Here you update `addBookmarkFromPlace()` to call the new `setImage()` method.

It's important to call this after the bookmark has been saved to the database so the bookmark has a unique ID assigned.

`setImage()` is used to save the image to the bookmark. The application context is passed into `setImage()` using `getApplication()`.

Simplifying the bookmark process

Before testing this new functionality, there's a small change you can make to simplify the process of adding a new bookmark. Currently, when selecting a place, a marker is displayed, and then the user has to tap on the marker again to display the info box. This change will automatically display the info box when showing the marker.

Open **MapsActivity.kt** and add the following line to the end of `displayPoiDisplayStep()`:

```
marker?.showInfoWindow()
```

This instructs the map to display the info window for the marker.

Build and run the app. Tap on a new place to display a marker and info window, and then tap on the info window. This will trigger a new bookmark to be saved, and this time it will store the bitmap image to a file.

Using Device File Explorer

If you want to verify the image was saved, and take a peek behind the scenes at how Android stores files, you can use the **Device File Explorer** in Android Studio. This is a handy tool for working directly with the Android file system.

Click on the **Device File Explorer** tool on the right-side of the Android Studio window. If you don't see it there, click **View ▸ Tool Windows ▸ Device File Explorer**.

In the newly displayed window, select the device on which you're running PlaceBook, and then navigate to **data/data/com.raywenderlich.placebook/files**. If the image save

worked correctly, you'll see at least one **bookmark?.png** image in the directory. Double-click on the image to preview it.
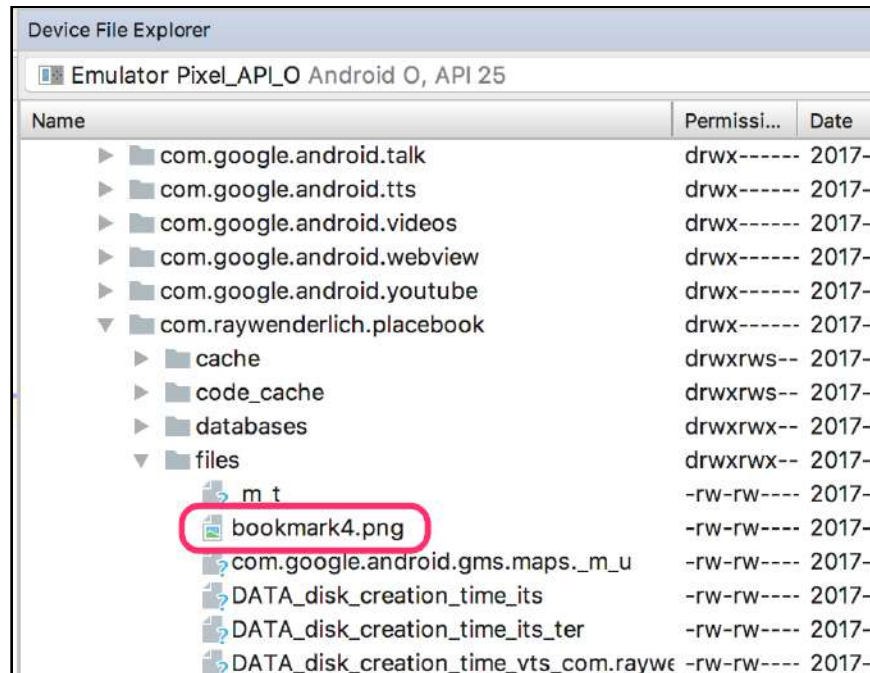
Loading an image

Now you need to load the image back from a file. This is considerably easier than saving an image, because Android provides a method on the `BitmapFactory` class for loading images from files.

In **ImageUtils.kt** add the following method:

```
fun loadBitmapFromFile(context: Context, filename: String):
    Bitmap? {
  val filePath = File(context.filesDir, filename).absolutePath
  return BitmapFactory.decodeFile(filePath)
}
```

This method is passed a `context` and a `filename` and returns a `Bitmap` image by loading the image from the specified `filename`. A `File` object is used to combine the files directory for the given context with the `filename`. A `filePath` is constructed from the absolute path of the `File`. The `BitmapFactory.decodeFile()` does the work of loading the image from the file, and the image is returned to the caller.

Updating BookmarkMarkerView

Now that you have the ability to load the image from where it was stored, let's update `BookmarkMarkerView` to provide the image for the view.

Your first instinct might be to add a new `Bitmap` object to `BookmarkMarkerView` and store it alongside the other properties. While this might work fine for a small set of bookmarks, you'll start eating up a lot of memory if a user has bookmarked hundreds of places! A better solution is to load the images on-demand.

Loading images on-demand

Open **MapsViewModel.kt** and add the following body to `BookmarkMarkerView`:

```
{
  fun getImage(context: Context): Bitmap? {
    id?.let {
      return ImageUtils.loadBitmapFromFile(context,
        Bookmark.generateImageFilename(it))
    }
    return null
  }
}
```

You first check to make sure the `BookmarkMarkerView` has a valid ID. Then you call `generateImageFilename()`, and pass in the bookmark ID represented as `it`. `loadBitmapFromFile()` is called with the current `context` and `Bookmark` image filename, and it returns the resulting `Bitmap` to the caller. You'll update the info window adapter to load the image when it's being rendered.

First, you need a `Context` object to load the image. You can take advantage of the fact that the `BookmarkInfoWindowAdapter` constructor already has a context passed in.

Open **BookmarkInfoWindowAdapter.kt** and replace the constructor with the following:

```
class BookmarkInfoWindowAdapter(val context: Activity) :
    GoogleMap.InfoWindowAdapter {
```

The only difference is the addition of the `val` modifier. This makes `context` a property so you can use it later to load the image.

Add the following code in `getInfoContents()` after the comment `// Set image bitmap here`:

```
imageView.setImageBitmap(bookMarkview.getImage(context))
```

Build and run the app. Tap on a blue marker for a saved bookmark that was created after you added the save image capability. This will display the info window with the bookmark image.
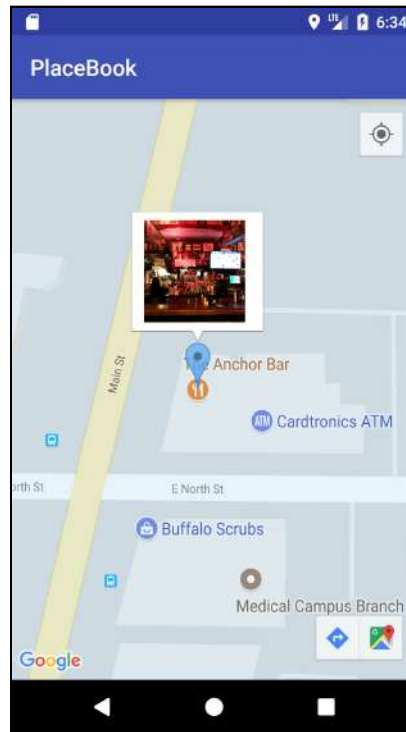
The image is showing, but there's no bookmark information displayed. This is easily fixed by adding the bookmark name and phone number to `BookmarkMarkerView`.

Updating the info window

Open **MapsViewModel.kt** and update the `BookmarkMarkerView` declaration to match the following:

```
data class BookmarkMarkerView(
    var id: Long? = null,
    var location: LatLng = LatLng(0.0, 0.0),
    var name: String = "",
    var phone: String = "") {
```

This adds new properties for `name` and `phone` to the `BookmarkMarkerView` class.

Update `bookmarkToMarkerView()` to match the following:

```
private fun bookmarkToMarkerView(bookmark: Bookmark):
    MapsViewModel.BookmarkMarkerView {
  return MapsViewModel.BookmarkMarkerView(
      bookmark.id,
      LatLng(bookmark.latitude, bookmark.longitude),
```

```
        bookmark.name,
        bookmark.phone)
  }
```

The only change is that the bookmark `name` and `phone` properties are passed into the new `BookmarkMarkerView` constructor.

Open **MapsActivity.kt**. In `addPlaceMarker()`, update the call to `map.addMarker()` with the following:

```
val marker = map.addMarker(MarkerOptions()
    .position(bookmark.location)
    .title(bookmark.name)
    .snippet(bookmark.phone)
    .icon(BitmapDescriptorFactory.defaultMarker(
        BitmapDescriptorFactory.HUE_AZURE))
    .alpha(0.8f))
```

The only change here is that the `title` and `snippet` items are set to the bookmark `name` and `phone`.

Build and run the app. Tap on a blue marker for a saved bookmark. This time it will display the name and phone number beside the image.
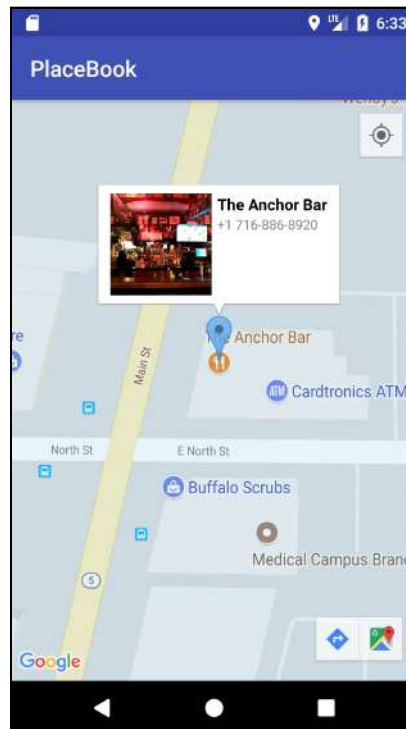
If you tapped on the info window, it most likely crashed on you. You'll fix that soon.

Bookmark detail activity

You've waited patiently, and it's finally time to build out the detail activity for editing a bookmark! You'll add a new screen that allows the user to edit key details about the bookmark, along with a custom note. You'll do this by creating a new activity that's displayed when a user taps on an info window.

Designing the edit screen

Before creating the activity, let's go over the screen layout and the main elements that will be incorporated.
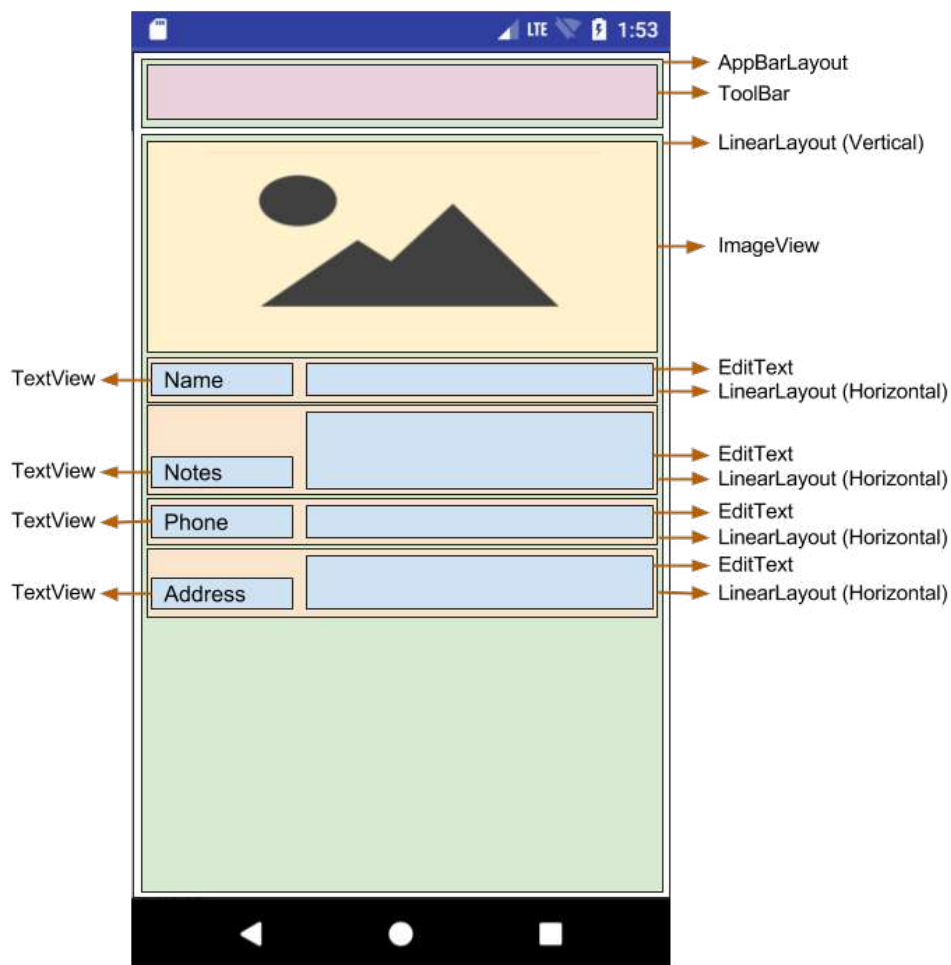
The Bookmark Edit Layout

- The top of the activity contains an `AppBarLayout`.

- Within the `AppBarLayout` is a Toolbar.

- Below the `AppBarLayout` is another vertical `LinearLayout` to hold the main list of Bookmark items.

- The first item in the vertical layout is the image view.

- Below the image view is a series of horizontal `LinearLayouts`. Each `LinearLayout` holds the label and edit control for a single item. The weights are set so the label takes 20% of the layout width.

Defining styles

First, you'll define some standard styles that are required when using the support library version of the toolbar.

Add the following to **res/values/styles.xml**:

```xml
<style name="AppTheme.NoActionBar">
  <item name="windowActionBar">false</item>
  <item name="windowNoTitle">true</item>
</style>

<style name="AppTheme.AppBarOverlay"
    parent="ThemeOverlay.AppCompat.Dark.ActionBar"/>
<style name="AppTheme.PopupOverlay"
    parent="ThemeOverlay.AppCompat.Light"/>
```

The `NoActionBar` style will be used to hide the native ActionBar. `AppBarOverlay` will give the toolbar layout a dark theme, and `PopupOverlay` will give the toolbar content a light theme.

The bookmark details activity will contain a list of text labels and fields that all have the same style.

You'll capitalize on this by defining a couple of styles that can be applied to the labels and fields without repeating information in the activity layout definition. This will also make it easier in the future to update the styles of all labels and text fields with a single change.

Add the following to **res/values/styles.xml**:

```xml
<style name="BookmarkLabel">
  <item name="android:layout_width">0dp</item>
  <item name="android:layout_height">wrap_content</item>
  <item name="android:layout_weight">0.2</item>
  <item name="android:layout_gravity">bottom</item>
  <item name="android:layout_marginStart">8dp</item>
  <item name="android:layout_marginLeft">8dp</item>
  <item name="android:layout_marginBottom">4dp</item>
  <item name="android:gravity">bottom</item>
```

```
    </style>

    <style name="BookmarkEditText">
      <item name="android:layout_width">0dp</item>
      <item name="android:layout_weight">0.8</item>
      <item name="android:layout_height">wrap_content</item>
      <item name="android:layout_marginEnd">8dp</item>
      <item name="android:layout_marginRight">8dp</item>
      <item name="android:layout_marginStart">8dp</item>
      <item name="android:layout_marginLeft">8dp</item>
      <item name="android:ems">10</item>
    </style>
```

The `BookmarkLabel` style defines the attributes for all bookmark labels.
`BookmarkEditText` defines the attributes for all bookmark edit fields.

Creating the details layout

Finally, you'll create the bookmark details layout based on the design. The activity will use all of the new styles just added to the project.

Create a new layout resource file at **res/layout/activity_bookmark_details.xml** and replace its contents 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="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <android.support.design.widget.AppBarLayout
        android:id="@+id/app_bar"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:fitsSystemWindows="true"
        android:theme="@style/AppTheme.AppBarOverlay">

      <android.support.v7.widget.Toolbar
          android:id="@+id/toolbar"
          android:layout_width="match_parent"
          android:layout_height="?attr/actionBarSize"
          app:popupTheme="@style/AppTheme.PopupOverlay"/>

    </android.support.design.widget.AppBarLayout>

    <ImageView
        android:id="@+id/imageViewPlace"
        android:layout_margin="0dp"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
```

```
            android:maxHeight="300dp"
            android:scaleType="fitCenter"
            android:adjustViewBounds="true"
            app:srcCompat="@drawable/default_photo"/>

    </LinearLayout>
```

This defines the basic layout for the bookmark details screen. The activity is contained within a vertical linear layout. The toolbar is defined as the first item in the layout, and the styles you defined earlier are used to theme the toolbar. The bookmark image is placed below the toolbar.

The layout up to this point looks like this:

Next, you'll add a series of form rows that represent the editable bookmark details. Each of these rows will be represented by a horizontal `LinearLayout` with a `TextView` on the left and an `EditText` element on the right.

First, you'll add a row for the bookmark name.

Add the following code below the `<ImageView>` element:

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="8dp"
    android:orientation="horizontal">

  <TextView
```

```
        android:id="@+id/textViewName"
        style="@style/BookmarkLabel"
        android:text="Name"/>

    <EditText
        android:id="@+id/editTextName"
        style="@style/BookmarkEditText"
        android:hint="Name"
        android:inputType="text"
        />
</LinearLayout>
```

You're using the `BookmarkLabel` and `BookmarkEditText` styles defined earlier to apply the layout details to the items.

Next, add a row for the bookmark notes. Add the following code after the bookmark name row:

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <TextView
        android:id="@+id/textViewNotes"
        style="@style/BookmarkLabel"
        android:text="Notes"/>

    <EditText
        android:id="@+id/editTextNotes"
        style="@style/BookmarkEditText"
```

```
        android:hint="Enter notes"
        android:inputType="textMultiLine"/>
</LinearLayout>
```

This repeats the formula used for the name row. The only difference is the `inputType` is set to allow multiple input lines. Next, add a row for the bookmark phone number.

Add the following code after the bookmark notes row:

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

  <TextView
      android:id="@+id/textViewPhone"
      style="@style/BookmarkLabel"
      android:text="Phone"/>

  <EditText
      android:id="@+id/editTextPhone"
      style="@style/BookmarkEditText"
      android:hint="Phone number"
      android:inputType="phone"
      />
</LinearLayout>
```

Next, add a row for the bookmark address. Add the following code after the bookmark phone number row:

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

  <TextView
      android:id="@+id/textViewAddress"
      style="@style/BookmarkLabel"
      android:text="Address"/>

  <EditText
      android:id="@+id/editTextAddress"
      style="@style/BookmarkEditText"
      android:hint="Address"
      android:inputType="textMultiLine"
      />
</LinearLayout>
```

The final layout after adding all of the rows will look like this:
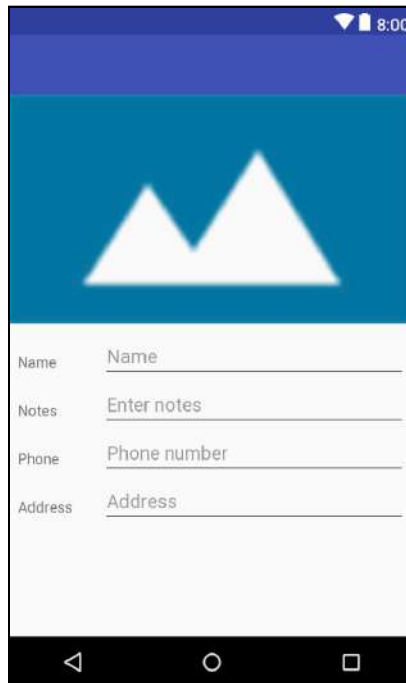
Details activity class

Now that the bookmark details layout is complete, you can create the details activity to go along with it.

Create a new Kotlin file under the **ui** package named **BookmarkDetailsActivity.kt** and replace the contents with the following:

```kotlin
class BookmarkDetailsActivity : AppCompatActivity() {
  override fun onCreate(savedInstanceState:
      android.os.Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_bookmark_details)
    setupToolbar()
  }

  private fun setupToolbar() {
    setSupportActionBar(toolbar)
  }
}
```

This is a fairly standard Activity class that uses the support action bar. `setupToolbar()` calls the built-in `setSupportActionBar()` to make the Toolbar act as the `ActionBar` for this Activity.

> **Note:** Because the app **build.gradle** file contains the `kotlin-android-extensions` plugin, Android Studio will automatically recognize the toolbar view from the **activity_bookmark_details** layout. If you look at the top of the file, you'll notice that it has included an `import` `kotlinx.android.synthetic.main.activity_bookmark_details.*` statement. This import includes the auto synthesized properties for the views in the layout.

Support design library

In order to use `setSupportActionBar()` you need to include the support design library.

Add the following line to the `buildscript.ext` section in the project **build.gradle** file.

```
support_lib_version = '26.1.0'
```

In the app **build.gradle** file, replace the `support:appcompat` line with the following:

```
implementation "com.android.support:appcompat-v7:$support_lib_version"
```

Add the following line after the `support:appcompat` line:

```
implementation "com.android.support:design:$support_lib_version"
```

This includes the design library in the app.

Updating the manifest

Next, make Android aware of the new `BookmarkDetailsActivity` class. Add the activity to **AndroidManifest.xml** within the `<application>` section:

```
<activity
    android:name=
        "com.raywenderlich.placebook.ui.BookmarkDetailsActivity"
    android:label="Bookmark"
    android:theme="@style/AppTheme.NoActionBar"
    android:windowSoftInputMode="stateHidden">
</activity>
```

Note that the theme with `NoActionBar` is required when using the support `Toolbar`. `android:windowSoftInputMode` is set to `stateHidden` to prevent the soft keyboard from displaying when the activity is first displayed.

Starting the details activity

Now you can hook up the new details activity to the main maps activity. You'll detect when the user taps on a bookmark info window, and then start the details activity. Add the following method to **MapsActivity.kt**:

```kotlin
private fun startBookmarkDetails(bookmarkId: Long) {
  val intent = Intent(this, BookmarkDetailsActivity::class.java)
  startActivity(intent)
}
```

Here, `startBookmarkDetails()` is used to start the `BookmarkDetailsActivity` using an explicit `Intent`. You'll call this method when the user taps on an info window for an existing bookmark.

Replace `handleInfoWindowClick()` with the following:

```kotlin
private fun handleInfoWindowClick(marker: Marker) {
  when (marker.tag) {
    is MapsActivity.PlaceInfo -> {
      val placeInfo = (marker.tag as PlaceInfo)
      if (placeInfo.place != null && placeInfo.image != null) {
        launch(CommonPool) {
          mapsViewModel.addBookmarkFromPlace(placeInfo.place,
              placeInfo.image)
        }
      }
      marker.remove();
    }
    is MapsViewModel.BookmarkMarkerView -> {
      val bookmarkMarkerView = (marker.tag as
          MapsViewModel.BookmarkMarkerView)
      marker.hideInfoWindow()
      bookmarkMarkerView.id?.let {
        startBookmarkDetails(it)
      }
    }
  }
}
```

This method handles the action when a user taps a place info window. Previously, it was designed to save the bookmark to the database. Now, it will save the bookmark if it hasn't been saved before, or it will start the bookmark details activity if the bookmark has already been saved.

Previously, this method assumed that the `marker.tag` would always be a `PlaceInfo` object. Now you're using the `when` construct to take a different action based on the `marker.tag` type. If it's a `BookmarkMarkerView`, then the info window is hidden and you start the bookmark details activity.

Build and run the app. Tap on a blue bookmark marker, and then tap on the info window. The new bookmark details screen will be shown.
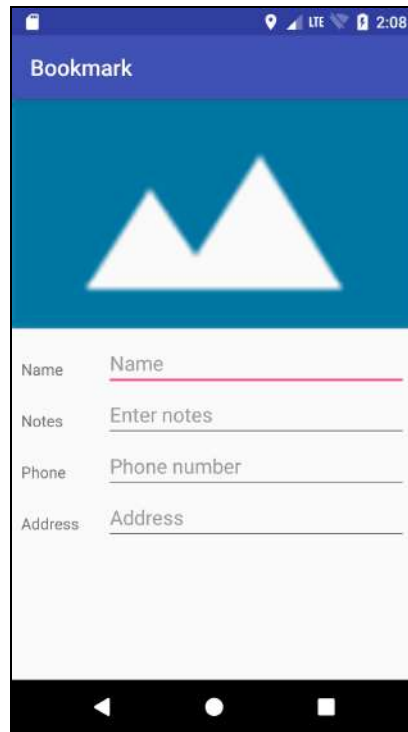
This is a good chance to verify the layout is working before populating the dialog with the actual bookmark content. Everything looks good in portrait, but rotate the device to landscape and you may see something like this:
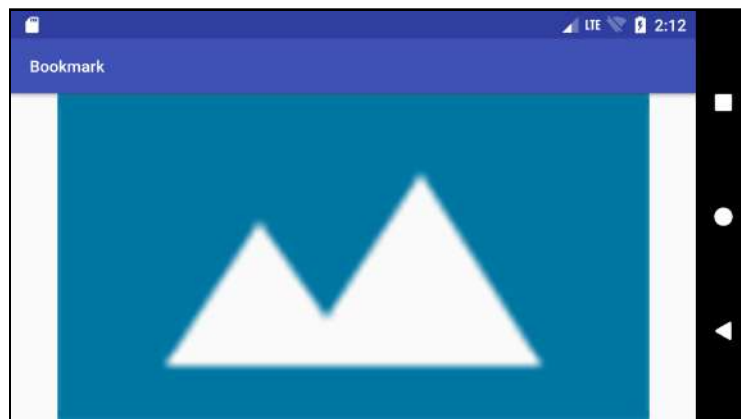
Whoops! On many Android devices, you'll only see the image with no way to scroll down and view the edit fields. This can be easily fixed by surrounding the main content with a **ScrollView**.

Open **activity_bookmark_details.xml** and add the following after `</`
`android.support.design.widget.AppBarLayout>`:

```
<ScrollView
    android:layout_width="match_parent"
    android:layout_height="match_parent">

  <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:orientation="vertical">
```

Add the following closing tags before the last `</LinearLayout>`:

```
    </LinearLayout>
</ScrollView>
```

By enclosing the main content in a `ScrollView`, you allow the user to scroll to see the entire details form.

Build and run the app again, and display the details for a place. Rotate to landscape mode and scroll the view to see the edit fields.
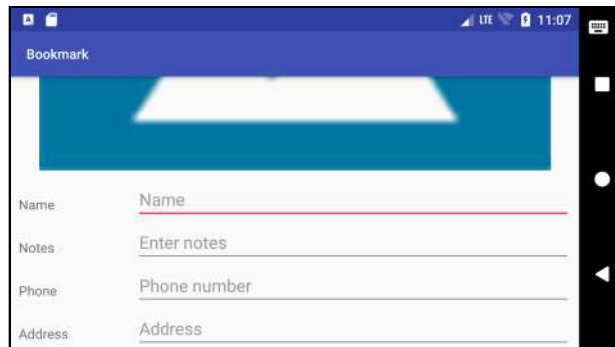
That looks much better!

Populating the bookmark

The activity has the general look you want, but it's lacking any knowledge about the bookmark. You'll pass the bookmark ID to the activity so it can display the bookmark data.

Open **MapsActivity.kt** and add the following to the top of the `companion object`:

```
const val EXTRA_BOOKMARK_ID =
    "com.raywenderlich.placebook.EXTRA_BOOKMARK_ID"
```

This defines a key for storing the bookmark ID in the intent extras.

Add the following line before the call to `startActivity()` in `startBookmarkDetails()`:

```
intent.putExtra(EXTRA_BOOKMARK_ID, bookmarkId)
```

This adds the `bookmarkId` as an extra parameter on the `intent`. Now you'll retrieve this parameter in the bookmark details activity, and use it to load the bookmark details.

Open **BookmarkRepo.kt** and add the following method:

```
fun getLiveBookmark(bookmarkId: Long): LiveData<Bookmark> {
  val bookmark = bookmarkDao.loadLiveBookmark(bookmarkId)
  return bookmark
}
```

This method returns a live bookmark from the bookmark DAO.

Just like `MapsActivity`, `BookmarkDetailsActivity` will use a `ViewModel` to coordinate the data between the view and the model.

You'll need to create a new view model class for the details activity. This class will use the bookmark repo to retrieve the bookmark details and format it for the details activity.

Create a new Kotlin file named **BookmarkDetailsViewModel.kt** in the **viewmodel** package, and replace the contents with the following:

```
class BookmarkDetailsViewModel(application: Application) :
    AndroidViewModel(application) {

  private var bookmarkRepo: BookmarkRepo =
      BookmarkRepo(getApplication())
}
```

`BookmarkDetailsViewModel` inherits from `AndroidViewModel` just like the `MapsViewModel` class. A private `BookmarkRepo` property is defined and initialized with a new `BookmarkRepo` instance.

You'll follow a similar pattern as you did with `MapsViewModel` to return data for the view. This pattern can be repeated anytime you need to return live data for a view, and it can be generalized as follows:

1. Define a new data class to hold the info required by the view class.

2. Define a `LiveData` property with the new data class.

3. Define a method to transform `LiveData` model data to `LiveData` view data.

4. Define a method to return the view data to the view.

Add the following internal class to **BookmarkDetailsViewModel**:

```kotlin
data class BookmarkDetailsView(
    var id: Long? = null,
    var name: String = "",
    var phone: String = "",
    var address: String = "",
    var notes: String = ""
) {

  fun getImage(context: Context): Bitmap? {
    id?.let {
      return ImageUtils.loadBitmapFromFile(context,
          Bookmark.generateImageFilename(it))
    }
    return null
  }
}
```

The `BookmarkDetailsView` class defines the data needed by the
`BookmarkDetailsActivity` view. `getImage()` loads the image associated with the
bookmark.

Adding notes to the database

Before continuing, you'll need a way to store notes for a bookmark.

Open **Bookmark.kt** and update the `Bookmark` declaration to add in the notes property,
like so:

```kotlin
data class Bookmark(
    @PrimaryKey(autoGenerate = true) var id: Long? = null,
    var placeId: String? = null,
    var name: String = "",
    var address: String = "",
    var latitude: Double = 0.0,
    var longitude: Double = 0.0,
    var phone: String = "",
    var notes: String = ""
)
```

Now that you've changed the `Bookmark` class, the main database class needs to be made
aware of it.

Open **PlaceBookDatabase.kt** and update the `@Database` annotation version to 2 as
follows:

```kotlin
@Database(entities = arrayOf(Bookmark::class), version = 2)
```

The change to `Bookmark` requires a change to the underlying database structure managed by Room. Setting the version to 2 lets Room know that something is different about the database.

The first time the app is launched after updating the version, Room will try to migrate data from the old structure to the new structure. It does this by looking for **Migrations** that you have added to the database builder. If you haven't added any Migrations, then an exception will be thrown and the app will crash.

Rather than providing Migrations, you can prevent the crash by telling Room to create the new database from scratch and discard all old data.

In the companion object `getInstance` method, replace the call to `Room.databaseBuilder` with the following:

```
instance = Room.databaseBuilder(context.applicationContext,
    PlaceBookDatabase::class.java, "PlaceBook")
    .fallbackToDestructiveMigration()
    .build()
```

This adds the `fallbackToDestructiveMigration()` call the builder. This tells Room to create a new empty database if it can't find any Migrations.

> **Note**: If you want to learn how to handle database schema changes using Migrations, please see the official documentation at https://developer.android.com/topic/libraries/architecture/room.html#db-migration.

Bookmark view model

That's all you need to support the revised `Bookmark` model in the database. Now you need to convert the database model to a view model.

Go back to **BookmarkDetailsViewModel.kt** and add the following method to the `BookmarkDetailsViewModel` class:

```
private fun bookmarkToBookmarkView(bookmark: Bookmark):
BookmarkDetailsView {
  return BookmarkDetailsView(
      bookmark.id,
      bookmark.name,
      bookmark.phone,
      bookmark.address,
      bookmark.notes
  )
}
```

This method converts from a `Bookmark` model to a `BookmarkDetailsView` model.

You'll need a property to hold the current bookmark view object.

Add the following to the top of the class:

```
private var bookmarkDetailsView: LiveData<BookmarkDetailsView>? = null
```

The `bookmarkDetailsView` property holds the `LiveData<BookmarkDetailsView>` object. This will allow the view to stay updated anytime the view model changes.

You have defined a method to convert from the database bookmark to the view bookmark, now you just need to convert from a live database bookmark object to a live bookmark view object.

Add the following method:

```
private fun mapBookmarkToBookmarkView(bookmarkId: Long) {
  val bookmark = bookmarkRepo.getLiveBookmark(bookmarkId)
  bookmarkDetailsView = Transformations.map(bookmark) { bookmark ->
      val bookmarkView = bookmarkToBookmarkView(bookmark)
      bookmarkView
    }
}
```

Here you get the live `Bookmark` from the `BookmarkRepo` and then transform it to the live `BookmarkDetailsView`.

Finally, you can bring it all together by exposing a method to return a live bookmark view based on a bookmark ID.

Next, add the following method:

```
fun getBookmark(bookmarkId: Long): LiveData<BookmarkDetailsView>? {
  if (bookmarkDetailsView == null) {
    mapBookmarkToBookmarkView(bookmarkId)
  }
  return bookmarkDetailsView
}
```

`getBookmark()` returns the `BookmarkDetailsView` object. If this is the first time `getBookmark()` is called, `mapBookmarkToBookmarkView()` is used to create the `bookmarkDetailsView`, otherwise the previously created `bookmarkDetailsView` is returned.

Retrieving the bookmark view

You're ready to add the code to retrieve the `BookmarkDetailsView` LiveData object.

First, you'll need some properties to hold the view model data.

Open **BookmarkDetailsActivity.kt** and add the following properties:

```
private lateinit var bookmarkDetailsViewModel:
    BookmarkDetailsViewModel
private var bookmarkDetailsView:
    BookmarkDetailsViewModel.BookmarkDetailsView? = null
```

And you'll need a method to initialize the view model. Add the following method:

```
private fun setupViewModel() {
  bookmarkDetailsViewModel =
      ViewModelProviders.of(this).get(
      BookmarkDetailsViewModel::class.java)
}
```

`setupViewModel()` creates the `bookmarkDetailsViewModel` using the `ViewModelProviders` class. This is the standard procedure for initializing a view model.

Add the following method to populate the fields in the view:

```
private fun populateFields() {
  bookmarkDetailsView?.let { bookmarkView ->
    editTextName.setText(bookmarkView.name)
    editTextPhone.setText(bookmarkView.phone)
    editTextNotes.setText(bookmarkView.notes)
    editTextAddress.setText(bookmarkView.address)
  }
}
```

This method populates all of the UI fields using the current `bookmarkView` if it's not null. You can also take the bookmark image from the view model and assign it to the image UI element.

Add the following method:

```
private fun populateImageView() {
  bookmarkDetailsView?.let { bookmarkView ->
    val placeImage = bookmarkView.getImage(this)
    placeImage?.let {
      imageViewPlace.setImageBitmap(placeImage)
    }
  }
}
```

This method loads the image from `bookmarkView` and then uses it to set the `imageViewPlace`.

Using the intent data

When the user taps on the info window for a bookmark on the maps activity, it will pass the bookmark ID to the details activity. You'll add a method to read this intent data, and use it to populate the UI.

Add the following method:

```
private fun getIntentData() {
  // 1
  val bookmarkId = intent.getLongExtra(
      MapsActivity.Companion.EXTRA_BOOKMARK_ID, 0)
  // 2
  bookmarkDetailsViewModel.getBookmark(bookmarkId)?.observe(
      this, Observer<BookmarkDetailsViewModel.BookmarkDetailsView> {
    // 3
    it?.let {
      bookmarkDetailsView = it
      // Populate fields from bookmark
      populateFields()
      populateImageView()
    }
  })
}
```

> **Note:** If Android Studio gives you two choices of imports for the Observer class, make sure to choose `import android.arch.lifecycle.Observer`.

This method will be called when the activity is created. Let's break it down:

1. The `bookmarkId` is pulled from the intent data.

2. The `BookmarkDetailsView` is retrieved from `BookmarkDetailsViewModel`, and then observed for changes.

3. Whenever the `BookmarkDetailsView` is loaded or changed, the `bookmarkDetailsView` property is assigned to it and the bookmark fields are populated from the data.

Finishing the detail activity

Now you're ready to pull everything together by adding the following calls to the end of `onCreate()` in **BookmarkDetailsActivity**.

```
setupViewModel()
getIntentData()
```

When the bookmark details activity is started, it will create the view model and process the intent data passed in from the maps activity.

Build and run the app. The previous data will be cleared out because of the database schema change.

Add a new bookmark and view the details. The bookmark info will now be displayed.
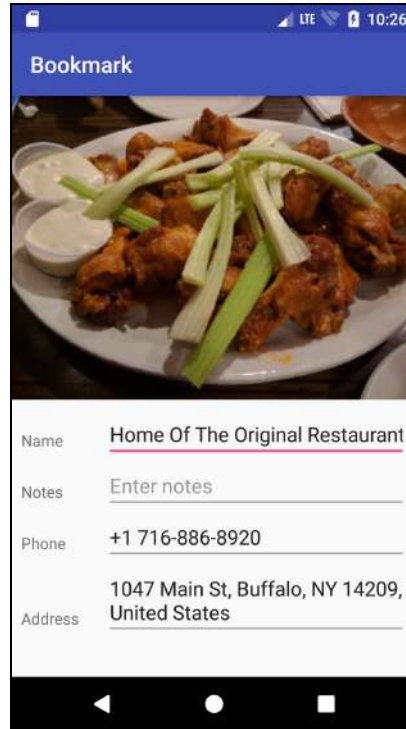
Saving changes

The only major feature left is to save the user's edits. You'll add a checkmark toolbar item to trigger the save.

First, you'll need a menu resource file to define a checkmark.

Create a new menu resource folder using **File ▸ New ▸ Android resource directory** with a name of **menu** and a resource type of menu.
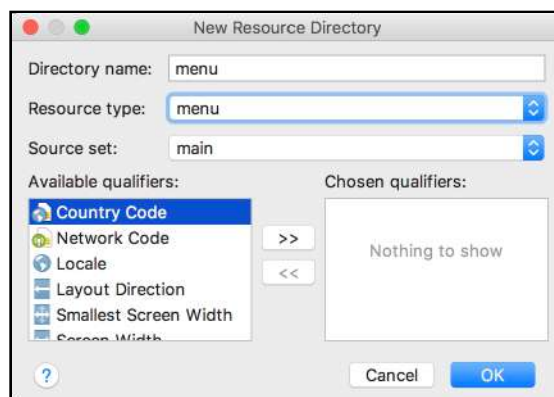
Create a new menu resource file named **menu_bookmark_details.xml** in **res/menu** and replace the contents with the following:

```xml
<?xml version="1.0" encoding="utf-8"?>
<menu
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=
    "com.raywenderlich.placebook.ui.BookmarkDetailsActivity">

  <item
      android:id="@+id/action_save"
      android:icon="@drawable/ic_action_done"
      android:title="Save"
      app:showAsAction="ifRoom"/>
</menu>
```

This defines a single menu item with an id of action_save for the detail activity toolbar.

Now you need to inflate the menu resource in the details activity.

Open **BookmarkDetailsActivity.kt** and add the following method:

```kotlin
override fun onCreateOptionsMenu(menu: android.view.Menu):
    Boolean {
  val inflater = menuInflater
  inflater.inflate(R.menu.menu_bookmark_details, menu)
  return true
}
```

You override onCreateOptionsMenu and provide items for the toolbar by loading in the menu_bookmark_details menu.

To save an updated bookmark to the database, you'll need a couple of new methods in BookmarkRepo.

Open **BookmarkRepo.kt** and add the following methods:

```kotlin
fun updateBookmark(bookmark: Bookmark) {
  bookmarkDao.updateBookmark(bookmark)
}

fun getBookmark(bookmarkId: Long): Bookmark {
  return bookmarkDao.loadBookmark(bookmarkId)
}
```

updateBookmark() takes in a bookmark and saves it using the bookmark DAO. getBookmark() takes in a bookmark ID and uses the bookmark DAO to load the corresponding bookmark.

When the user makes changes to a bookmark, you'll update the bookmark view model class. You'll need a method to convert a bookmark view model to the database bookmark model.

Open **BookmarkDetailsViewModel.kt** and add the following method:

```kotlin
private fun bookmarkViewToBookmark(bookmarkView: BookmarkDetailsView):
    Bookmark? {
  val bookmark = bookmarkView.id?.let {
    bookmarkRepo.getBookmark(it)
  }
  if (bookmark != null) {
    bookmark.id = bookmarkView.id
    bookmark.name = bookmarkView.name
    bookmark.phone = bookmarkView.phone
    bookmark.address = bookmarkView.address
    bookmark.notes = bookmarkView.notes
  }
  return bookmark
}
```

This method takes a `BookmarkDetailsView` and returns a `Bookmark` with the updated parameters from the `BookmarkDetailsView`. The original bookmark values are loaded from the `BookmarkRepo` before updating them with the `BookmarkDetailsView`. It's important to load in the original bookmark to retain the values that aren't updated by the `BookmarkDetailsView`.

You can now utilize `bookmarkViewToBookmark()` to create a new public method to update a bookmark in the background.

Add the following method:

```kotlin
fun updateBookmark(bookmarkView: BookmarkDetailsView) {
  // 1
  launch(CommonPool) {
    // 2
    val bookmark = bookmarkViewToBookmark(bookmarkView)
    // 3
    bookmark?.let { bookmarkRepo.updateBookmark(it) }
  }
}
```

This method updates the bookmark from a `BookmarkDetailsView`.

1. A coroutine is used to run the method in the background. This allows calls to be made by the bookmark repo that access the database.

2. The `BookmarkDetailsView` is converted to a `Bookmark`.

3. If the bookmark is not null, it's updated in the bookmark repo. This will update the bookmark in the database.

Now you can modify the bookmark details activity and make use of the new
`updateBookmark()` method provided by the view model.

Open **BookmarkDetailsActivity.kt** and add the following method:

```kotlin
private fun saveChanges() {
  val name = editTextName.text.toString()
  if (name.isEmpty()) {
    return
  }
  bookmarkDetailsView?.let { bookmarkView ->
    bookmarkView.name = editTextName.text.toString()
    bookmarkView.notes = editTextNotes.text.toString()
    bookmarkView.address = editTextAddress.text.toString()
    bookmarkView.phone = editTextPhone.text.toString()
    bookmarkDetailsViewModel.updateBookmark(bookmarkView)
  }
  finish()
}
```

This method takes the current changes from the text fields and updates the bookmark.
The method doesn't do anything if the `editTextName` field is blank. After updating the
`bookmarkView` with the data from the EditText fields, `updateBookmark()` is called to
update the bookmark model. Finally, the activity is closed with the `finish()` call.

Next, you'll add code to respond to the user tapping the checkmark menu item and then
call `saveChanges()`.

Add the following method:

```kotlin
override fun onOptionsItemSelected(item: MenuItem): Boolean {
  when (item.itemId) {
    R.id.action_save -> {
      saveChanges()
      return true
    }
    else -> return super.onOptionsItemSelected(item)
  }
}
```

This method is called when the user selects a toolbar checkmark item. You check the
`item.itemId` to see if it matches `action_save`, and if so, `saveChanges()` is called.

Build and run the app. Go into the details activity of an existing bookmark and change
some the data. Tap the checkmark in the toolbar to save your changes. Now, display the
details for the same bookmark, and you'll see that the data reflects your changes.
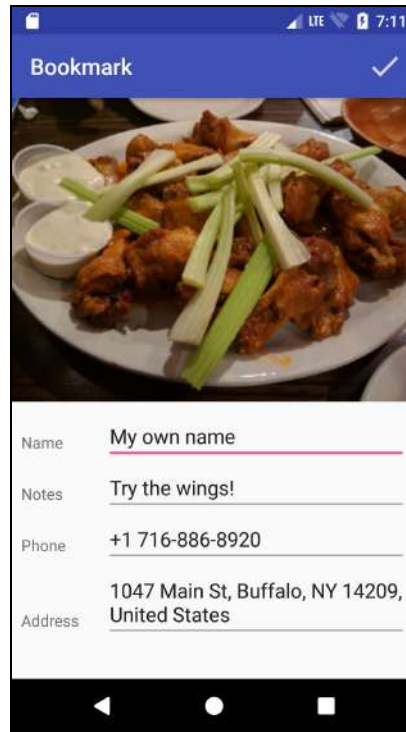
Congratulations! You can now edit bookmarks, but there's still more work to do. The next chapter will wrap things up by adding some additional features and putting the finishing touches on the app.

Chapter 18: Navigation & Photos

By Tom Blankenship

In this chapter, you'll add the ability to navigate directly to bookmarks, and you'll replace the photo for a bookmark.

Getting started

The starter project for this chapter includes an additional icon that you'll need in order to complete the chapter. You can either begin this chapter with the starter project, or copy the following resource from the starter project into yours:

- **src/main/res/drawable/ic_other.png**

Make sure to copy the files from all of the drawable folders, including everything with the `.hdpi, .mdpi, .xhdpi, .xxhdpi` and `.xxxhdpi` extensions.

If you do use the starter app, don't forget to add your `google_maps_key` in **google_maps_api.xml**. Check out Chapter 13 for more details about the Google Maps key.

Bookmark navigation

Currently, the only way to find an existing bookmark is to locate its pin on the map. Let's save a little skin on the user's fingertips by creating a **Navigation Drawer** that can be used to jump directly to any bookmark.

Navigation drawer design

It's hard to use Android very long without encountering a navigation drawer. Although the uses vary, they share a common design pattern. The drawer is hidden to the left of the main content view and is activated with either a swipe from the left edge of the screen or by tapping a navigation drawer icon. Once the drawer is activated, it slides out over-top of the main content, and slides back in once an action has been taken by the user.

Adding a navigation drawer can be done in three steps:

- Make `DrawerLayout` the root view of your layout.

- Make the first view within `DrawerLayout` your main content.

- Make the second view within `DrawerLayout` your navigation drawer content.

The final navigation drawer will look like this:
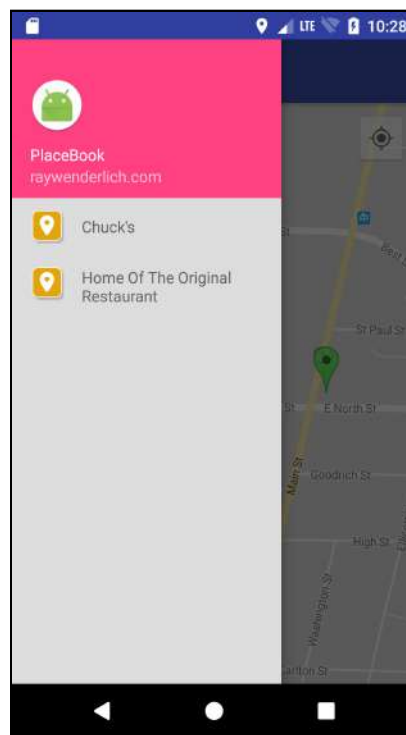
Navigation drawer layout

To create the drawer layout, you'll create a new layout file for the navigation drawer, move the map fragment from **activity_maps.xml** to its own layout file, and update **activity_maps.xml** to contain the `DrawerLayout` element.

First you'll move the map fragment to a separate layout.

Create a new Layout resource file in the **res/layout** folder named
main_view_maps.xml 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"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context="com.raywenderlich.placebook.ui.MapsActivity"
  android:orientation="vertical">

  <android.support.design.widget.AppBarLayout
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:theme="@style/AppTheme.AppBarOverlay">

    <android.support.v7.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="?attr/colorPrimary"
        app:popupTheme="@style/AppTheme.PopupOverlay"/>

  </android.support.design.widget.AppBarLayout>

  <fragment
    android:id="@+id/map"
    android:name="com.google.android.gms.maps.SupportMapFragment"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:map="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.raywenderlich.placebook.ui.MapsActivity"
    />

</LinearLayout>
```

This file will be included in **activity_maps.xml**. A root `LinearLayout` is defined to hold
a standard action bar just like the one you created for the detail activity. The action bar
is required in order to hold the navigation drawer toggle icon.

You'll eventually add code in **MapsActivity.kt** to dynamically create the navigation
drawer toggle icon for the action bar.

Next, you need a layout to define the navigation drawer.

Create a new Layout resource file in the **res/layout** folder named
drawer_view_maps.xml and replace the contents with the following:

```xml
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/drawerView"
    android:layout_width="240dp"
    android:layout_height="match_parent"
    android:layout_gravity="start"
    android:orientation="vertical"
    android:background="#ddd">

  <LinearLayout
      xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto"
      android:layout_width="match_parent"
      android:layout_height="140dp"
      android:background="@color/colorAccent"
      android:gravity="bottom"
      android:orientation="vertical"
      android:paddingBottom="10dp"
      android:paddingLeft="16dp"
      android:paddingRight="16dp"
      android:paddingTop="10dp"
      android:theme="@style/ThemeOverlay.AppCompat.Dark">

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingTop="10dp"
        app:srcCompat="@mipmap/ic_launcher_round"/>

    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:paddingTop="10dp"
        android:text="PlaceBook"
        android:textAppearance=
            "@style/TextAppearance.AppCompat.Body1"/>

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="raywenderlich.com"/>

  </LinearLayout>

  <android.support.v7.widget.RecyclerView
      android:id="@+id/bookmarkRecyclerView"
      android:scrollbars="vertical"
      android:layout_width="match_parent"
      android:layout_height="match_parent"/>

</LinearLayout>
```

This layout defines the contents of the navigation drawer. There are a few key elements that are important to note:

- The main `layout_width` is set to `240dp`. This is a safe width that ensures some of the underlying view will be visible when the drawer is fully open. For mobile devices, the maximum size recommended by the design guidelines is 280dp.

- The main layout specifies a `layout_gravity` of "start" instead of "left". This will place the drawer on the right side of the screen if the user's language is RTL (right-to-left).

- The layout defines a top header area used to display the app icon and some basic application information.

- The area below the header contains a `RecyclerView`. This view will be used to display the list of stored bookmarks.

Now you need a layout for each bookmark item that will be shown in the navigation drawer.

Create a new Layout resource file in the **res/layout** folder named **bookmark_item.xml** 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"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:paddingTop="10dp"
    android:paddingBottom="10dp"
    android:paddingLeft="16dp"
    android:paddingRight="16dp">

  <ImageView
      android:id="@+id/bookmarkIcon"
      android:layout_width="30dp"
      android:layout_height="30dp"
      android:layout_marginEnd="16dp"
      android:adjustViewBounds="true"
      android:scaleType="fitStart"/>

  <TextView
      android:id="@+id/bookmarkNameTextView"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:layout_gravity="center_vertical"
      tools:text="Name"/>

</LinearLayout>
```

This defines the layout for a single bookmark entry in the `RecyclerView`. You define a simple layout with a bookmark category icon on the left and the bookmark title on the right.

That completes the new layout files needed for the navigation drawer. Now you just need to update the main maps activity to use the new layouts. Open **activity_maps.xml** and replace the contents with the following:

```xml
<?xml version="1.0" encoding="utf-8"?>

<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawerLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:openDrawer="start"
    >

  <include layout="@layout/main_view_maps"/>
  <include layout="@layout/drawer_view_maps"/>

</android.support.v4.widget.DrawerLayout>
```

Your main activity layout previously contained a single map fragment that filled the entire screen. Now it has a root `DrawerLayout` that includes the `main_view_maps` and the `drawer_view_maps`.

To make the navigation drawer and action bar work properly, you need to do a few more things.

Open **AndroidManifest.xml** and update the `MapsActivity` `<activity>` entry to match the following:

```xml
<activity
    android:name=".ui.MapsActivity"
    android:label="@string/title_activity_maps"
    android:theme="@style/AppTheme.NoActionBar">
  <intent-filter>
    <action android:name="android.intent.action.MAIN"/>
    <category android:name="android.intent.category.LAUNCHER"/>
  </intent-filter>
</activity>
```

The only change is to add the `AppTheme.NoActionBar` theme style. This is standard procedure when using the support library version of the toolbar as the action bar.

The final piece is to activate support for the support toolbar in the maps activity. Open **MapsActivity.kt** and add the following method to `MapsActivity`:

```
private fun setupToolbar() {
   setSupportActionBar(toolbar)
}
```

> **Note**: Make sure to use `import kotlinx.android.synthetic.main.`
> `main_view_maps.*` for the toolbar reference.

Again, this is standard setup code that's required when using the support library version of the toolbar as the action bar. Build and run the app. Swipe right starting on the left edge of the screen; the navigation drawer will slide out. To close it, swipe left on the navigation drawer.
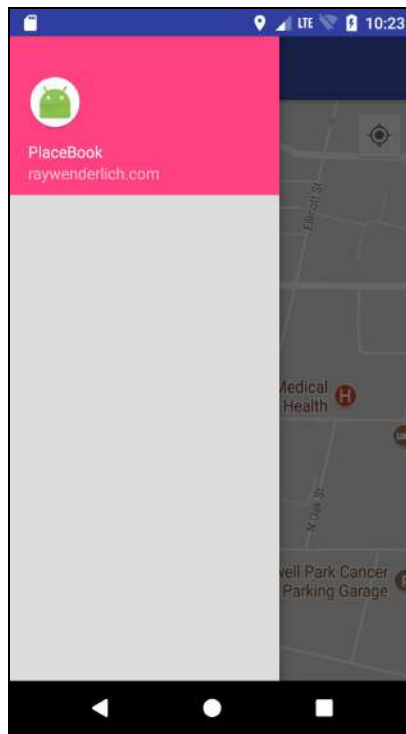
Navigation toolbar toggle

Add a toggle button for the navigation drawer by creating an `ActionBarDrawerToggle`.

The constructor for `ActionBarDrawerToggle` requires two string resources for the open and closed drawer states.

Add the following lines to **res/values/strings.xml**:

```
<string name="open_drawer">Open Drawer</string>
<string name="close_drawer">Close Drawer</string>
```

Add the following to the end of `setupToolbar()` in **MapsActivity.kt**:

```
val toggle = ActionBarDrawerToggle(
    this,  drawerLayout, toolbar,
    R.string.open_drawer, R.string.close_drawer)
toggle.syncState()
```

The `ActionBarDrawerToggle` will take your `drawerLayout` and `toolbar` and fully manage the display and functionality of the toggle icon. `toggle.syncState` is called to ensure the toggle icon is displayed initially. The last two arguments set the content descriptor on the action bar, based on the navigation drawer state.

> **Note:** If given the choice for imports on `ActionBarDrawerToggle`, choose `android.support.v7.app.ActionBarDrawerToggle`.

All that's left to do is call `setupToolbar` when the activity is created.

Add the following lines before `setupGoogleClient()` in `onCreate()`:

```
setupToolbar()
```

This calls the two new methods to bind the controls and setup the toolbar with the toggle icon.

Build and run the app. Tap the toggle (hamburger) icon to test the navigation drawer slide.
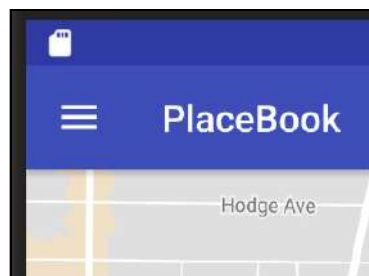
Populating the navigation bar

To populate the navigation bar, you'll provide an adapter to the `RecyclerView` and use **LiveData** to update the adapter any time bookmarks change in the database.

The adapter will require some view data — one option would be to create a new data class in `MapsViewModel`. You already have the `BookmarkMarkerView` class used by the `MapsActivity` for the map markers, so you'll take advantage of the existing class and the code that observes changes to the data.

Since you'll be using `BookmarkMarkerView` to display markers and the navigation drawer items, it needs a more generic name. This is a good opportunity to use Android Studio's convenient refactoring capabilities.

Open **MapsViewModel.kt** and find the `BookmarkMarkerView` declaration. **Right-click** on the word `BookmarkMarkerView`, and then select **Refactor ▸ Rename…** or place the cursor on `BookmarkMarkerView` and press **Shift+F6**.
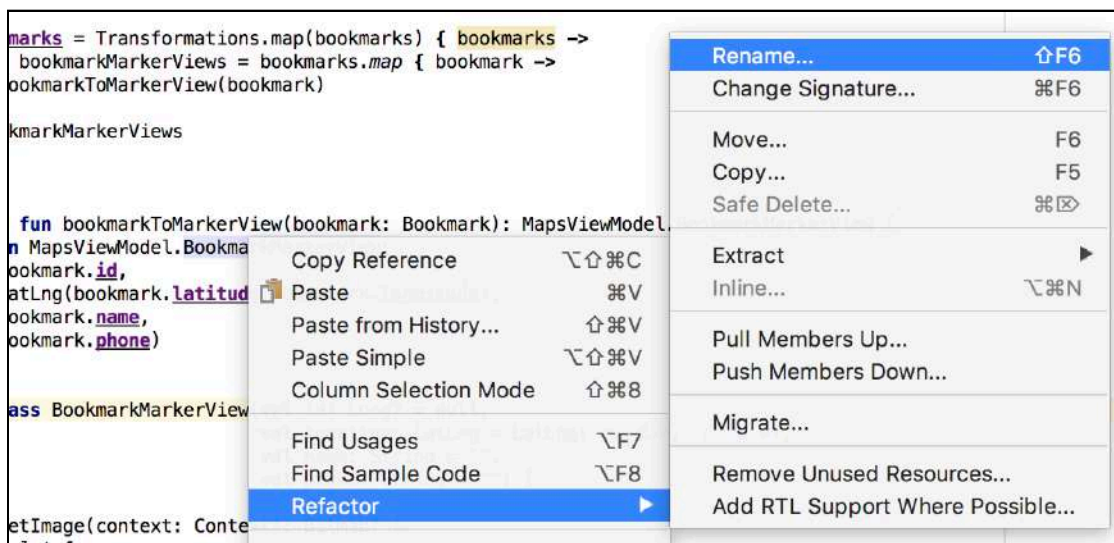
BookmarkMarkerView will be highlighted. Change the name to **BookmarkView** and press **Enter**.
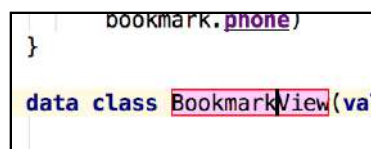
This will automatically update all references to use `BookmarkView` instead of `BookmarkMarkerView`. This is great feature that can save a lot of time when renaming classes, methods or variables.

Use the same rename feature to change the following:

- `getBookmarkMarkerViews()` ⇢ `getBookmarkViews()`

- `mapBookmarksToMarkerView()` ⇢ `mapBookmarksToBookmarkView()`

- `bookmarkToMarkerView()` ⇢ `bookmarkToBookmarkView()`.

In **MapsActivity.kt**, use the rename feature to change
`createBookmarkMarkerObserver()` to `createBookmarkObserver()`.

In order to populate the recycler view in the navigation drawer, you'll need to create a new recycler view adapter class.

Create a new Kotlin class in the **adapter** package and name it
BookmarkListAdapter.kt. Now replace the contents with the following:

```kotlin
// 1
class BookmarkListAdapter(
    private var bookmarkData: List<BookmarkView>?,
    private val mapsActivity: MapsActivity) :
    RecyclerView.Adapter<BookmarkListAdapter.ViewHolder>() {
  // 2
  class ViewHolder(v: View,
      private val mapsActivity: MapsActivity) :
      RecyclerView.ViewHolder(v) {
    val nameTextView: TextView =
        v.findViewById(R.id.bookmarkNameTextView) as TextView
    val categoryImageView: ImageView =
        v.findViewById(R.id.bookmarkIcon) as ImageView
  }
  // 3
  fun setBookmarkData(bookmarks: List<BookmarkView>) {
    this.bookmarkData = bookmarks
    notifyDataSetChanged()
  }
  // 4
  override fun onCreateViewHolder(
      parent: ViewGroup,
      viewType: Int): BookmarkListAdapter.ViewHolder {
    val vh = ViewHolder(
        LayoutInflater.from(parent.context).inflate(
        R.layout.bookmark_item, parent, false), mapsActivity)
    return vh
  }

  override fun onBindViewHolder(holder: ViewHolder,
      position: Int) {
    // 5
    val bookmarkData = bookmarkData ?: return
    // 6
    val bookmarkViewData = bookmarkData[position]
    // 7
    holder.itemView.tag = bookmarkViewData
```

```
      holder.nameTextView.text = bookmarkViewData.name
      holder.categoryImageView.setImageResource(
          R.drawable.ic_other)
  }

  // 8
  override fun getItemCount(): Int {
    return bookmarkData?.size ?: 0
  }
}
```

`BookmarkListAdapter` is a standard `RecyclerView` adapter you learned about in Chapter 7, "RecyclerViews".

Note: Android Studio may not import the View class automatically. If it doesn't, add `import android.view.View` to the top of the file.

1. The adapter constructor takes two arguments; a list of `BookmarkView` items and a reference to the `MapsActivity`. Both arguments are defined as class properties.

2. A `ViewHolder` class is defined to hold the view widgets.

3. `setBookmarkData` is designed to be called when the bookmark data changes. It assigns `bookmarks` to the new `BookmarkView` List and refreshes the `RecyclerView` by calling `notifyDataSetChanged()`.

4. `onCreateVieHolder` is overridden and used to create a `ViewHolder` by inflating the `bookmark_item` layout and passing in the `mapsActivity` property.

5. `bookmarkData` is assigned to `bookmarkData` if it's not null, otherwise you return early.

6. `bookmarkViewData` is assigned to the bookmark data for the current item `position`.

7. A reference to the `bookmarkViewData` is assigned to the holder's `itemView.tag`, and the `ViewHolder` items are populated from the `bookmarkViewData`. For now, a default icon is used to represent the bookmark category.

8. `getItemCount()` is overridden to return the number of items in the `bookmarkData` list.

Now you can use the adapter in the maps activity. Open **MapsActivity.kt** and add the following property to `MapsActivity`:

```
private lateinit var bookmarkListAdapter: BookmarkListAdapter
```

Add the following method to `MapsActivity`:

```
private fun setupNavigationDrawer() {
  val layoutManager = LinearLayoutManager(this)
  bookmarkRecyclerView.layoutManager = layoutManager
  bookmarkListAdapter = BookmarkListAdapter(null, this)
  bookmarkRecyclerView.adapter = bookmarkListAdapter
}
```

This method sets up the adapter for the bookmark recycler view. It gets the
`RecyclerView` from the layout, sets a default `LinearLayoutManager` for the
`RecyclerView`, then creates a new `BookmarkListAdapter` and assigns it to the
`RecyclerView`.

You'll need to set up the navigation drawer at the time the activity is created. Add the
following line to the end of `onCreate()`:

```
setupNavigationDrawer()
```

In addition, you need to make sure the list adapter is updated any time the list of
bookmarks changes. This can be handled in `createBookmarkObserver()`.

Add the following line to `createBookmarkObserver()`, after the call to
`displayAllBookmarks(it)`:

```
bookmarkListAdapter.setBookmarkData(it)
```

This sets the new list of `BookmarkView` items on the recycler view adapter whenever the
bookmark data changes. This will cause the navigation drawer items to update and
reflect the current state of the database.

Build and run the app. Make sure you have some bookmarks saved and then open the
navigation drawer. You'll see it populated with the list of bookmark names. Add a new
bookmark, and the navigation drawer should update to reflect the addition.
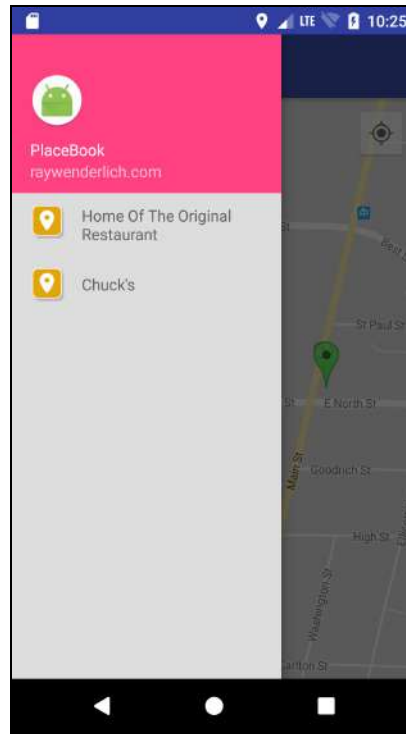
Navigation bar selections

It's great that users can now see a list of bookmark names, but it's not very functional! Let's add the ability to zoom to a bookmark when the user taps an item in the navigation drawer.

First, you'll add a method that centers the map on a bookmark marker and opens the marker's info window.

Before writing this method, you need a way to get a handle on a map marker for a given bookmark instance. Unfortunately, there's no direct way to get a list of all markers managed by the `GoogleMap` object — you'll just have to take matters into your own hands!

An easy way to manage the markers is to use a **HashMap** that associates bookmark IDs to map markers.

Open **MapsActivity.kt** and add the following property:

```
private var markers = HashMap<Long, Marker>()
```

This creates and initializes a `HashMap` to map a bookmark ID (`Long`) to a `Marker`.

Add the following line before the return in `addPlaceMarker()`:

```
bookmark.id?.let { markers.put(it, marker) }
```

This will add a new entry to `markers` when a new marker is added to the map.

In `createBookmarkObserver()`, add the following line after the call to `map.clear()`:

```
markers.clear()
```

This clears `markers` when the bookmark data changes. `markers` will be populated again as all of the bookmarks are added to the map.

You'll also need a way to update the map to the location of a bookmark.

Start by adding a helper method to zoom the map to a specific location.

Add the following method to `MapsActivity`:

```
private fun updateMapToLocation(location: Location) {
  val latLng = LatLng(location.latitude, location.longitude)
  map.animateCamera(
      CameraUpdateFactory.newLatLngZoom(latLng, 16.0f))
}
```

This pans and zooms the map to center over a `Location`. a `LatLng` is created from the Location and is used to create the `LatLngZoom` object for `animateCamera()`. `animateCamera()` is similar to the `moveCamera()` method that you used before, but it smoothly pans the map instead of abruptly jumping to the new location.

With that in place, you can now make a new method that moves the map to a bookmark location.

Finally, add the following method to `MapsActivity`:

```
fun moveToBookmark(bookmark: MapsViewModel.BookmarkView) {
  // 1
  drawerLayout.closeDrawer(drawerView)
  // 2
  val marker = markers[bookmark.id]
  // 3
  marker?.showInfoWindow()
  // 4
  val location = Location("")
  location.latitude =  bookmark.location.latitude
  location.longitude = bookmark.location.longitude
  updateMapToLocation(location)
}
```

1.   Before zooming the bookmark, the navigation drawer is closed.

2. The `markers HashMap` is used to look up the `Marker`.

3. If the marker is found, its info window is shown.

4. A `Location` object is created from the bookmark, and `updateMapToLocation()` is called to zoom the map to the bookmark.

The final step is to call `moveToBookmark()` when the user taps on a bookmark. This will be handled by the bookmark list adapter class.

Open **BookmarkListAdapter.kt** and add the following method to the `ViewHolder` class:

```
init {
  v.setOnClickListener {
    val bookmarkView = itemView.tag as BookmarkView
    mapsActivity.moveToBookmark(bookmarkView)
  }
}
```

This method is called when a `ViewHolder` is initialized. It sets an `onClickListener` on the `ViewHolder`. When the click event is fired you get the `bookmarkView` associated with the `ViewHolder` and call `moveToBookmark()` to zoom the map to the bookmark.

Before wrapping up this feature, let's add one simple change to sort the bookmarks by name. The simplest place to do this is in the bookmark data access object.

Open **BookmarkDao.kt** and update the `@Query` attribute on `loadAll()` to match the following:

```
@Query("SELECT * FROM Bookmark ORDER BY name")
```

Build and run the app. Open the navigation drawer and notice how the bookmarks are now sorted by name. Tap on a bookmark item; the navigation drawer will close, and the map will zoom to the selected bookmark with its info window already open.
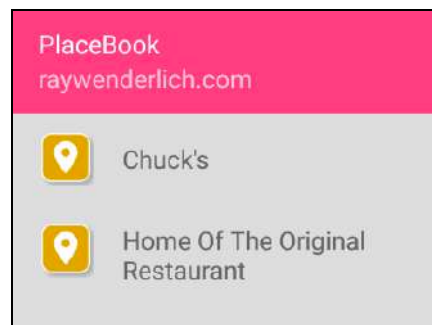
Custom photos

While Google provides a default photo for each place, your users may prefer to use that perfect selfie instead! In this section, you'll add the ability to replace the place photo with one from the photo library or one you take on-the-fly with the camera.

Image option dialog

You'll start by creating a dialog to let the user choose between an existing image or capturing a new one.

Create a new Kotlin file under the **ui** package named **PhotoOptionDialogFragment.kt** and set the contents as follows:

```kotlin
class PhotoOptionDialogFragment : DialogFragment() {
  // 1
  interface PhotoOptionDialogListener {
    fun onCaptureClick()
    fun onPickClick()
  }
  // 2
  private lateinit var listener: PhotoOptionDialogListener
  // 3
  override fun onCreateDialog(savedInstanceState: Bundle?):
      Dialog {
    // 4
    listener = activity as PhotoOptionDialogListener
    // 5
    var captureSelectIdx = -1
    var pickSelectIdx = -1
    // 6
    val options = ArrayList<String>()
    // 7
    if (canCapture(this.context)) {
      options.add("Camera")
      captureSelectIdx = 0
    }
    // 8
    if (canPick(this.context)) {
      options.add("Gallery")
      pickSelectIdx = if (captureSelectIdx == 0) 1 else 0
    }
    // 9
    return AlertDialog.Builder(activity)
        .setTitle("Photo Option")
        .setItems(options.toTypedArray<CharSequence>()) {
            _, which ->
          if (which == captureSelectIdx) {
            // 10
            listener.onCaptureClick()
          } else if (which == pickSelectIdx) {
            // 11
            listener.onPickClick()
```

```
        }
      }
      .setNegativeButton("Cancel", null)
      .create()
}

companion object {
  // 12
  fun canPick(context: Context) : Boolean {
    val pickIntent = Intent(Intent.ACTION_PICK,
        MediaStore.Images.Media.EXTERNAL_CONTENT_URI)
    return (pickIntent.resolveActivity(
        context.packageManager) != null)
  }
  // 13
  fun canCapture(context: Context) : Boolean {
    val captureIntent = Intent(
        MediaStore.ACTION_IMAGE_CAPTURE)
    return (captureIntent.resolveActivity(
        context.packageManager) != null)
  }
  // 14
  fun newInstance(context: Context):
      PhotoOptionDialogFragment? {
    // 15
    if (canPick(context) || canCapture(context)) {
      val frag = PhotoOptionDialogFragment()
      return frag
    } else {
      return null
    }
  }
}
}
```

> **Note:** Make sure to import the `android.support.v4.app.DialogFragment` and `android.support.v7.app.AlertDialog` when given options for imports.

This class defines a dialog fragment that will show an `AlertDialog` with one or two options, based on the device capabilities. If the device can select images from the gallery, then a Gallery option will be included. If the device has a camera to capture new images, then a Camera option will be included.

1. The class defines an interface that must be implemented by the parent activity. You'll implement this interface in `BookmarkDetailsActivity`.

2. A property is defined to hold an instance of `PhotoOptionDialogListener`.

3. This is the standard `onCreateDialog` method for a `DialogFragment`.

4. The listener property is set to the parent activity.

5. The two possible option indices are initialized to -1. The option indices will be defined dynamically, because the position Gallery and Camera options may change based on the device capabilities.

6. An option `ArrayList` is defined to hold the `AlertDialog` options.

7. If the device has a camera capable of capturing images, then a Camera option is added to the options array. The `captureSelectIdx` variable is set to 0 to indicate the Camera option will be at position 0 in the option list.

8. If the device can pick an image from a gallery, then a Gallery option is added to the options array. The `pickSelectIdx` variable is set to 0 if it's the first option, or to 1 if it's the second option.

9. The `AlertDialog` is built using the options list, and an `onClickListener` is provided to respond to the user selection.

10. If the Camera option was selected, then `onCaptureClick()` is called on `listener`.

11. If the Gallery option was selected, then `onPickClick()` is called on `listener`.

12. `canPick()` determines if the device can pick an image from a gallery. It determines this by creating an intent for picking images, and then it checks to see if the Intent can be resolved. This is a standard method for detecting if a particular Intent option is possible on the current device.

13. `canCapture()` determines if the device has a camera to capture a new image. It uses the same technique as `canPick()` but with a different Intent action.

14. newInstance is a helper method intended to be used by the parent activity when creating a new `PhotoOptionDialogFragment`.

15. If the device can pick from a gallery or snap a new image, then the `PhotoOptionDialogFragment` is created and returned, otherwise `null` is returned.

Open **BookmarkDetailsActivity.kt** and update the class declaration as follows, so that it implements the `PhotoOptionDialogListener` interface:

```
class BookmarkDetailsActivity : AppCompatActivity(),
    PhotoOptionDialogFragment.PhotoOptionDialogListener {
```

This will cause an error until you implement the `PhotoOptionDialogListener` interface.

Add the following methods:

```
override fun onCaptureClick() {
  Toast.makeText(this, "Camera Capture",
```

```
        Toast.LENGTH_SHORT).show()
  }
  override fun onPickClick() {
    Toast.makeText(this, "Gallery Pick",
        Toast.LENGTH_SHORT).show()
  }
```

You'll soon implement the code to snap a photo or pick one from the gallery, but for now they're just place holders.

Now you can add a method that creates the photo option dialog and displays it to the user.

Still in **BookmarkDetailsActivity.kt** and add the following method:

```
private fun replaceImage() {
  val newFragment = PhotoOptionDialogFragment.newInstance(this)
  newFragment?.show(supportFragmentManager, "photoOptionDialog")
}
```

You'll call `replaceImage()` when the user taps on the bookmark image. You attempt to create the `PhotoOptionDialogFragment` fragment. If `newFragment` is not `null`, then it's displayed.

All that's left is to listen for the `imageViewPlace` to be tapped and call `replaceImage()`.

Add the following code at the end of `populateImageView()`:

```
imageViewPlace.setOnClickListener {
  replaceImage()
}
```

This sets a click listener on `imageViewPlace` and calls `replaceImage()` when the image is tapped.

Build and run the app. Bring up the details for a bookmark and tap the photo. The options dialog will display. Tap on one of the options and the appropriate toast should be displayed.
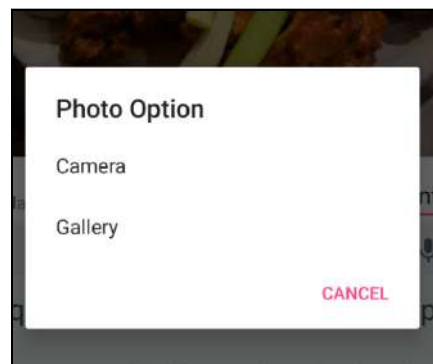
Now you're ready to implement the code to capture or pick the image. You'll start with the capture option.

Capturing an image

Capturing a full-size image from Android consists of the following steps:

1. Create a unique filename to store the captured image.

2. Create an Intent with the `MediaStore.ACTION_IMAGE_CAPTURE` action.

3. Add the Uri to the unique filename as an extra on the Intent.

4. Invoke the Intent using `startActivityForResult`.

5. Respond to the activity result, and process the captured image, which will be located at the filename Uri you provided.

Generate a unique filename

First you'll create a helper method to generate a unique image filename.

Open **ImageUtils.kt** and add the following method:

```kotlin
@Throws(IOException::class)
fun createUniqueImageFile(context: Context): File {
  val timeStamp =
      SimpleDateFormat("yyyyMMddHHmmss").format(Date())
  val filename = "PlaceBook_" + timeStamp + "_"
  val filesDir = context.getExternalFilesDir(
      Environment.DIRECTORY_PICTURES)
  return File.createTempFile(filename, ".jpg", filesDir)
}
```

> **Note**: Make sure to use `import java.text.SimpleDateFormat` for `SimpleDateFormat` and `jave.util.Date` for `Date`.

This method returns an empty `File` in the app's private pictures folder using a unique filename. The filename is created by using the current timestamp with "PlaceBook_" prepended.

The method is flagged with `@Throws` to account for `File.createTempFile()` possibly throwing an `IOException`.

Next, you'll add a property to the details activity in order to keep track of the image File.

Open **BookmarkDetailsActivity.kt** and add the following property:

```
private var photoFile: File? = null
```

This will be used to hold a reference to the temporary image file when capturing an image.

Start the capture activity

Before you can call the image capture activity, you need to define a request code. This can be any number you choose. It will be used to identify the request when the image capture activity returns the image.

You'll define this request code as a constant value in a companion object.

Add the following internal companion object to the bottom of
BookmarkDetailsActivity:

```
companion object {
  private const val REQUEST_CAPTURE_IMAGE = 1
}
```

This defines the request code to use when processing the camera capture intent. Now it's time to replace the temporary onCaptureClick() method with one that actually captures an image. Replace the contents of onCaptureClick() with the following:

```
// 1
photoFile = null
try {
  // 2
  photoFile = ImageUtils.createUniqueImageFile(this)
  // 3
  if (photoFile == null) {
    return
  }
} catch (ex: java.io.IOException) {
  // 4
  return
}
// 5
val captureIntent =
Intent(android.provider.MediaStore.ACTION_IMAGE_CAPTURE)
// 6
val photoUri = FileProvider.getUriForFile(this,
    "com.raywenderlich.placebook.fileprovider",
    photoFile)
// 7
captureIntent.putExtra(android.provider.MediaStore.EXTRA_OUTPUT,
    photoUri)
// 8
val intentActivities = packageManager.queryIntentActivities(
    captureIntent, PackageManager.MATCH_DEFAULT_ONLY)
```

```
intentActivities.map { it.activityInfo.packageName }
    .forEach { grantUriPermission(it, photoUri,
    Intent.FLAG_GRANT_WRITE_URI_PERMISSION) }
// 9
startActivityForResult(captureIntent, REQUEST_CAPTURE_IMAGE)
```

1. Any previously assigned `photoFile` is cleared.

2. `photoFile` is assigned to a new unique file.

3. If `photoFile` is `null` the method returns without doing anything.

4. If an exception is thrown, the method returns without doing anything.

5. A new `Intent` is created with the `ACTION_IMAGE_CAPTURE` action. This intent is used to display the camera viewfinder and allow the user to snap a new photo.

6. `FileProvider.getUriForFile()` is called to get a Uri for the temporary photo file.

7. The `photoUri` is added as an extra on the `Intent` so the intent knows where to save the full-size image captured by the user.

8. Temporary write permissions on the `photoUri` are given to the `Intent`.

9. The `Intent` is invoked, and the request code `REQUEST_CAPTURE_IMAGE` is passed in.

> **Note:** `FileProvider` works by creating a content:// Uri for a file versus a file:// Uri. This is important to allow granting of temporary access permissions to read and write files. You can read more about `FileProvider` and why it is more secure than using file:// Uris by going to https://developer.android.com/reference/android/support/v4/content/FileProvider.html.

Using a `FileProvider` requires that it be registered in the **AndroidManifest.xml** file.

Register the FileProvider

Open **AndroidManifest.xml** and add the following to the `<application>` section:

```xml
<provider
    android:name="android.support.v4.content.FileProvider"
    android:authorities="com.raywenderlich.placebook.fileprovider"
    android:exported="false"
    android:grantUriPermissions="true">
  <meta-data
      android:name="android.support.FILE_PROVIDER_PATHS"
      android:resource="@xml/file_paths"/>
</provider>
```

This declares your `FileProvider` with the authority of
"com.raywenderlich.placebook.fileprovider". You can choose any unique name here; by
convention, this should start with your app's package name. Notice that it matches the
name used when calling `FileProvider.getUriForFile()`.

The `FileProvider` references an xml resource file that defines the allowed file paths.
You'll create this resource file now.

Select **File ▸ New ▸ Android resource file** and set the File name to **file_paths** and the
Resource type to **XML**. The directory name should change to **xml**. Tap **OK**.

This will create a new res directory named **xml** containing the new `file_paths.xml` file.

Now you can fill in the **file_paths.xml** with the allowed file paths.

Replace the contents of **file_paths.xml** with the following:

```xml
<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android="http://schemas.android.com/apk/res/android">
  <external-path
    name="placebook_images"
    path=
    "Android/data/com.raywenderlich.placebook/files/Pictures" />
</paths>
```

This defines a single path to the `Pictures` directory within the `PlaceBook` file container.

Build and run the app. Tap the photo on a bookmark photo, then select the Camera
option. Verify that the device camera is activated and you are able to snap a photo.
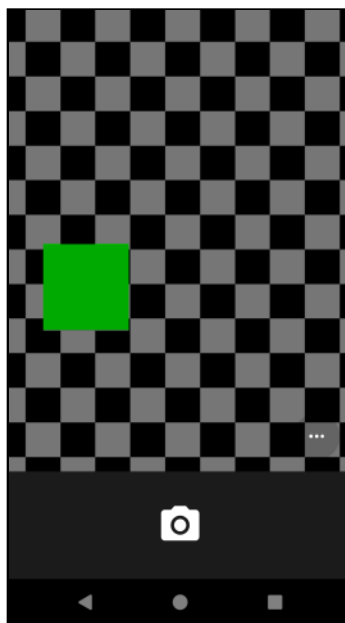
Emulator Camera View

The photo will not update when the camera view is closed, because you haven't written the code to process the capture intent results yet.

Process the capture results

The images captured from the camera can be much larger than what's needed to display in the app. As part of the processing of the newly captured photo, you'll downsample the photo to match the default bookmark photo size. This calls for some new methods in the **ImageUtils.kt** class.

Open **ImageUtils.kt** and add the following private method:

```kotlin
private fun calculateInSampleSize(
    width: Int, height: Int,
    reqWidth: Int, reqHeight: Int): Int {

  var inSampleSize = 1

  if (height > reqHeight || width > reqWidth) {
    val halfHeight = height / 2
    val halfWidth = width / 2
    while (halfHeight / inSampleSize >= reqHeight &&
        halfWidth / inSampleSize >= reqWidth) {
      inSampleSize *= 2
    }
  }

  return inSampleSize
}
```

This method is used to calculate the optimum `inSampleSize` that can be used to resize an image to a specified width and height. The `inSampleSize` must be specified as a power of two. This method starts with an `inSampleSize` of 1 (no downsampling), and it increases the `inSampleSize` by a power of two until it reaches a value that will cause the image to be downsampled to no larger than the requested image width and height.

Now that you can calculate the proper sample size for any width and height, a new method can be added to decode a file. This method is called when an image needs to be downsampled.

Add the following method:

```kotlin
fun decodeFileToSize(filePath: String,
    width: Int, height: Int): Bitmap {
  // 1
  val options = BitmapFactory.Options()
  options.inJustDecodeBounds = true
  BitmapFactory.decodeFile(filePath, options)
  // 2
  options.inSampleSize = calculateInSampleSize(
```

```
      options.outWidth, options.outHeight, width, height)
  // 3
  options.inJustDecodeBounds = false
  // 4
  return BitmapFactory.decodeFile(filePath, options)
}
```

This method will be called by `BookmarkDetailsActivity` to get the downsampled image with a specific `width` and `height` from the captured photo file.

1. The size of the image is loaded using `BitmapFactory.decodeFile()`. The `inJustDecodeBounds` setting tells `BitmapFactory` to not load the actual image, just its size.

2. `calculateInSampleSize()` is called with the image width and height and the requested `width` and `height`. Options is updated with the resulting `inSampleSize`.

3. `inJustDecodeBounds` is set to false in order to load the full image this time.

4. `BitmapFactory.decodeFile()` loads the downsampled image from the file and it is returned.

The `BookmarkView` class now needs a new method to replace the image for a bookmark.

Add the following method to the **BookmarkDetailsView** class in the **BookmarkDetailsViewModel.kt** file:

```
fun setImage(context: Context, image: Bitmap) {
  id?.let {
    ImageUtils.saveBitmapToFile(context, image,
        Bookmark.generateImageFilename(it))
  }
}
```

This takes in a `Bitmap` image and saves it to the associated image file for the current `BookmarkView`.

Now that `BookmarkView` is able to replace its own image, you'll create a method in the details activity to replace the image in the `imageViewPlace` control and update the bookmark view object.

Open **BookmarkDetailsActivity.kt** and add the following method:

```
private fun updateImage(image: Bitmap) {
  val bookmarkView = bookmarkDetailsView ?: return
  imageViewPlace.setImageBitmap(image)
  bookmarkView.setImage(this, image)
}
```

This method assigns an image to the `imageViewPlace` and saves it to the bookmark image file using `bookmarkDetailsView.setImage()`.

In order to read in and process the image captured by the system, you need to a method that will take a file path and return the downsized image as a Bitmap.

Add the following method:

```
private fun getImageWithPath(filePath: String): Bitmap? {
  return ImageUtils.decodeFileToSize(filePath,
      resources.getDimensionPixelSize(
          R.dimen.default_image_width),
      resources.getDimensionPixelSize(
          R.dimen.default_image_height))
}
```

This method uses the new `decodeFileSize` method to load the downsampled image and return it.

With all of the supporting code in place, you're ready to process the camera results.

Add the following method:

```
override fun onActivityResult(requestCode: Int, resultCode: Int,
    data: Intent?) {
  super.onActivityResult(requestCode, resultCode, data)
  // 1
  if (resultCode == android.app.Activity.RESULT_OK) {
    // 2
    when (requestCode) {
      // 3
      REQUEST_CAPTURE_IMAGE -> {
        // 4
        val photoFile = photoFile ?: return
        // 5
        val uri = FileProvider.getUriForFile(this,
            "com.raywenderlich.placebook.fileprovider",
            photoFile)
        revokeUriPermission(uri,
            Intent.FLAG_GRANT_WRITE_URI_PERMISSION)
        // 6
        val image = getImageWithPath(photoFile.absolutePath)
        image?.let { updateImage(it) }
      }
    }
  }
}
```

`onActivityResult()` is called by Android when an Activity returns a result such as the Camera capture activity.

1. First the `resultCode` is checked to make sure the user didn't cancel the photo capture.

2. The `requestCode` is checked to see what type of activity is returning a result.

3. If the `requestCode` matches `REQUEST_CAPTURE_IMAGE`, then processing continues.

4. You return early from the method if there is no `photoFile` defined.

5. The permissions you set before are now revoked since they're no longer needed.

6. `getImageWithPath()` is called to get the image from the new photo path, and `updateImage()` is called to update the bookmark image.

Build and run the app. Edit a bookmark and tap on the photo. Tap on Camera and then snap a new photo. The bookmark photo will update to show the new photo. Go back to the map view, and then edit the same bookmark again to verify that the new photo is displayed.
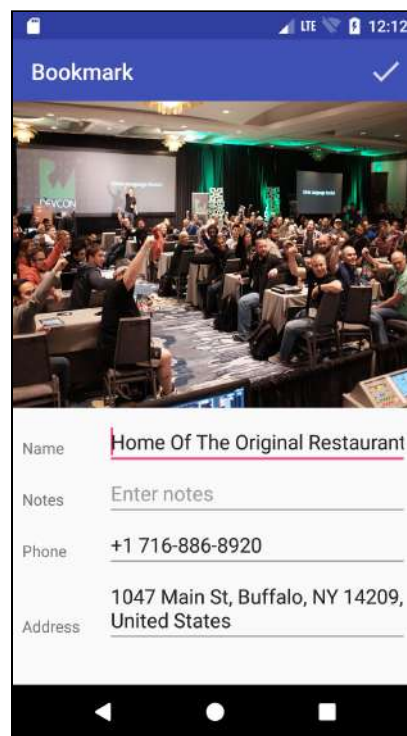
Select an existing image

Now you'll add the option to pick an existing image from the device's gallery.

When selecting from the device gallery, you don't provide a temporary file for the image storage. Instead, the image selection activity gives you a Uri to the selected image.

You'll need a new method that reads an image from a Uri input stream.

Open **ImageUtils.kt** and add the following method:

```kotlin
fun decodeUriStreamToSize(uri: Uri,
    width: Int, height: Int, context: Context): Bitmap? {
  var inputStream: InputStream? = null
  try {
    val options: BitmapFactory.Options
    // 1
    inputStream = context.contentResolver.openInputStream(uri)
    // 2
    if (inputStream != null) {
      // 3
      options = BitmapFactory.Options()
      options.inJustDecodeBounds = false
      BitmapFactory.decodeStream(inputStream, null, options)
      // 4
      inputStream.close()
      inputStream = context.contentResolver.openInputStream(uri)
      if (inputStream != null) {
        // 5
          options.inSampleSize = calculateInSampleSize(
              options.outWidth, options.outHeight,
              width, height)
        options.inJustDecodeBounds = false
        val bitmap = BitmapFactory.decodeStream(
            inputStream, null, options)
        inputStream.close()
        return bitmap
      }
    }
    return null
  } catch (e: Exception) {
    return null
  } finally {
    // 6
    inputStream?.close()
  }
}
```

This uses the same technique as `decodeFileToSize()` to read in the size of the image first, calculate the sample size and then load in the downsampled image. The main difference is that it reads from the Uri stream instead of a file.

1. `inputStream` is opened for the `Uri`.

2. If the `inputStream` is not `null`, then processing continues.

3. The image size is determined.

4. The input stream is closed and opened again, and checked for `null`.

5. The image is loaded from the stream using the downsampling options and is returned to the caller.

6. You must close the `inputStream` once it's opened, even if an exception is thrown.

Now you'll need a new request code to identify the results from the image selection activity.

Open **BookmarkDetailsActivity.kt** and add the following to the companion object:

```
private const val REQUEST_GALLERY_IMAGE = 2
```

You can now replace the empty `onPickClick()` with a version that kicks off Android's image selection activity.

Replace the contents of `onPickClick()` with the following:

```
val pickIntent = Intent(Intent.ACTION_PICK,
    MediaStore.Images.Media.EXTERNAL_CONTENT_URI)
startActivityForResult(pickIntent, REQUEST_GALLERY_IMAGE)
```

In order to process the results of the image selection, you'll need a method that returns a downsampled Bitmap from a Uri path.

Add the following method:

```
private fun getImageWithAuthority(uri: Uri): Bitmap? {
  return ImageUtils.decodeUriStreamToSize(uri,
      resources.getDimensionPixelSize(
          R.dimen.default_image_width),
      resources.getDimensionPixelSize(
          R.dimen.default_image_height),
      this)
}
```

This method uses the new `decodeUriStreamToSize` method to load the downsampled image and return it.

Now you just need to add a new case to handle existing images in `onActivityResult()`. This time you'll handle the result of the image selection activity.

In `onActivityResult()`, add the following new clause to the `when` conditional block:

```
REQUEST_GALLERY_IMAGE -> if (data != null && data.data != null) {
  val imageUri = data.data
  val image = getImageWithAuthority(imageUri)
  image?.let { updateImage(it) }
}
```

If the activity result is from selecting a gallery image, and the data returned is valid, then `getImageWithAuthority()` is called to load the selected image. `updateImage()` is called to update the bookmark image.

Build and run the app. Edit a bookmark and tap on the photo. Tap on Gallery and then select an existing photo. The bookmark photo will update to show the selected photo.

Go back to the map view and then edit the same bookmark again to verify that the new photo is displayed.

Where to go from here?

Great job! You've added some key features to the app, and have completed the primary bookmarking features. In the next chapter you'll add some finishing touches that will kick the app up a notch.

Chapter 19: Finishing Touches

By Tom Blankenship

In this chapter, you'll add some finishing touches that improve both the look and usability of the PlaceBook app. Even though PlaceBook is perfectly functional as-is, it's often the little touches that make an app go from good to great. With that in mind, you'll wrap things up by making the following changes:

- Add categories for bookmarks

- Display category specific icons on the map

- Add place search

- Add ad-hoc bookmark creation

- Add bookmark deletions

- Add bookmark sharing

- Update the color scheme and display progress

Getting started

The starter project for this chapter includes some additional resources and an updated application icon. You can either begin this chapter with the starter project, or copy the following resources from the starter project into yours:

- src/main/ic_launcher_round-web.png

- src/main/ic_launcher-web.png

- src/main/res/drawable/ic_gas.png

- src/main/res/drawable/ic_lodging.png

- src/main/res/drawable/ic_restaurant.png

- src/main/res/drawable/ic_search_white.png

- src/main/res/drawable/ic_shopping.png

- src/main/res/mipmap/ic_launcher_round.png

- src/main/res/mipmap/ic_launcher.png

Make sure to copy the files from all of the drawable folders, including everything with the .hdpi, .mdpi, .xhdpi and .xxhdpi extensions.

If you're using the starter project, remember to replace the key in **google_maps_api.xml**.

Bookmark categories

Assigning categories to bookmarks gives you the opportunity to show different icons on the map for each type of place. Google already provides category information for Places. You'll use this to set a default category, and let the user assign a different category if they choose.

Update the model

Start by adding a new category property to the `Bookmark` model.

Open **Bookmark.kt** and update the `Bookmark` declaration to add a category property:

```
data class Bookmark(
    @PrimaryKey(autoGenerate = true) var id: Long? = null,
```

```
      var placeId: String? = null,
      var name: String = "",
      var address: String = "",
      var latitude: Double = 0.0,
      var longitude: Double = 0.0,
      var phone: String = "",
      var notes: String = "",
      var category: String = ""
  )
```

Open **PlaceBookDatabase.kt** and update the `@Database` annotation version to 3:

```
@Database(entities = arrayOf(Bookmark::class), version = 3)
```

Note: As mentioned in Chapter 17, if you don't update the version number after modifying the model, an exception will be thrown by `Room`. By changing the version number, `Room` will create a brand new database on the first run using the new version number.

Converting place types

If you examine the `Place` class defined by the Google Play Services, you'll notice that it provides a fairly long list of place types such as:

```
int TYPE_OTHER = 0;
int TYPE_ACCOUNTING = 1;
int TYPE_AIRPORT = 2;
int TYPE_AMUSEMENT_PARK = 3;
int TYPE_AQUARIUM = 4;
int TYPE_ART_GALLERY = 5;
...
```

To keep things manageable, PlaceBook will support only four categories: **Gas**, **Lodging**, **Restaurant** and **Shopping**. All other types of places will be assigned to the **Other** category.

You'll need a method to map a Google Place type to a supported PlaceBook category. You'll only convert the `Place` types that can easily map to one of the four categories. All other types will map to the **Other** category.

Open **BookmarkRepo.kt** and add the following method:

```
private fun buildCategoryMap() : HashMap<Int, String> {
  return hashMapOf(
      Place.TYPE_BAKERY to "Restaurant",
      Place.TYPE_BAR to "Restaurant",
      Place.TYPE_CAFE to "Restaurant",
      Place.TYPE_FOOD to "Restaurant",
```

```
        Place.TYPE_RESTAURANT to "Restaurant",
        Place.TYPE_MEAL_DELIVERY to "Restaurant",
        Place.TYPE_MEAL_TAKEAWAY to "Restaurant",
        Place.TYPE_GAS_STATION to "Gas",
        Place.TYPE_CLOTHING_STORE to "Shopping",
        Place.TYPE_DEPARTMENT_STORE to "Shopping",
        Place.TYPE_FURNITURE_STORE to "Shopping",
        Place.TYPE_GROCERY_OR_SUPERMARKET to "Shopping",
        Place.TYPE_HARDWARE_STORE to "Shopping",
        Place.TYPE_HOME_GOODS_STORE to "Shopping",
        Place.TYPE_JEWELRY_STORE to "Shopping",
        Place.TYPE_SHOE_STORE to "Shopping",
        Place.TYPE_SHOPPING_MALL to "Shopping",
        Place.TYPE_STORE to "Shopping",
        Place.TYPE_LODGING to "Lodging",
        Place.TYPE_ROOM to "Lodging"
    )
  }
```

This builds a `HashMap` that relates the `Place` types to the category names. Any types not included in the list will end up mapping to the "Other" category as you'll see in `placeTypeToCategory()`.

Add the following property to `BookmarkRepo`:

```
  private var categoryMap:
      HashMap<Int, String> = buildCategoryMap()
```

`categoryMap` is initialized to hold the mapping of place types to category names.

Add the following method:

```
  fun placeTypeToCategory(placeType: Int): String {
    var category = "Other"
    if (categoryMap.containsKey(placeType)) {
      category = categoryMap[placeType].toString()
    }
    return category
  }
```

This method will take in a `Place` type and convert it to a valid category. The `category` variable is initialized to "Other" by default. If the `categoryMap` contains a key matching the `placeType` then it's assigned to `category`.

You may be wondering why `toString()` is used on the value retrieved from the `categoryMap` HashMap. The reason is that accessing a `HashMap` with a missing key will return a `null` value. To satisfy the compiler, you must force the value to be a string. In this case, you've used `containsKey()` to ensure that the key is in the `HashMap`, so you're safe.

Now it's time to make use of the new icons provided in the starter project. The icons will correspond to the categories like so:

- ic_other = Other

- ic_gas = Gas

- ic_lodging = Lodging

- ic_restaurant = Restaurant

- ic_shopping = Shopping

First, you'll map the category names to the drawable resource files.

Add the following method to `BookmarkRepo`:

```kotlin
private fun buildCategories() : HashMap<String, Int> {
  return hashMapOf(
      "Gas" to R.drawable.ic_gas,
      "Lodging" to R.drawable.ic_lodging,
      "Other" to R.drawable.ic_other,
      "Restaurant" to R.drawable.ic_restaurant,
      "Shopping" to R.drawable.ic_shopping
  )
}
```

This builds a `HashMap` that relates the category names to the category icon resource IDs.

Add the following property to `BookmarkRepo`:

```kotlin
private var allCategories: HashMap<String, Int> =
    buildCategories()
```

`allCategories` is initialized to hold the mapping of category names to resource IDs.

Add the following method:

```kotlin
fun getCategoryResourceId(placeCategory: String): Int? {
    return allCategories[placeCategory]
}
```

This method provides a public method to convert a category name to a resource ID.

Updating the view model

Now you're ready to update the map's view model to support bookmark categories. Open **MapsViewModel.kt** and add the following private method:

```kotlin
private fun getPlaceCategory(place: Place): String {
  // 1
```

```
    var category = "Other"
    val placeTypes = place.placeTypes
    // 2
    if (placeTypes.size > 0) {
      // 3
      val placeType = placeTypes[0]
      category = bookmarkRepo.placeTypeToCategory(placeType)
    }
    // 4
    return category
  }
```

This method converts a place type to a bookmark category. The task is slightly complicated due to the possibility of multiple types being assigned to a single place.

1. The `category` is defaulted to `"Other"` just in case there's no type assigned to the place.

2. The method first checks the `placeTypes` List to see if it is populated.

3. If so, the first type is extracted from the List and `placeTypeToCategory()` is called to make the conversion.

4. Finally, the category is returned.

Update `addBookmarkFromPlace()` and add the following assignment before the call to `addBookmark()`:

```
  bookmark.category = getPlaceCategory(place)
```

This assigns the category to the newly created `bookmark`.

Update the `BookmarkView` class declaration to include a new category resource ID property:

```
  data class BookmarkView(val id: Long? = null,
                          val location: LatLng = LatLng(0.0, 0.0),
                          val name: String = "",
                          val phone: String = "",
                          val categoryResourceId: Int? = null) {
```

`categoryResourceId` is added and will hold the resource icon for the bookmark's category.

Update `bookmarkToBookmarkView()` to reflect the new `BookmarkView` declaration:

```
  private fun bookmarkToBookmarkView(bookmark: Bookmark):
      MapsViewModel.BookmarkView {
    return MapsViewModel.BookmarkView(
        bookmark.id,
        LatLng(bookmark.latitude, bookmark.longitude),
```

```
        bookmark.name,
        bookmark.phone,
        bookmarkRepo.getCategoryResourceId(bookmark.category))
}
```

Updating the UI

You can now update the user interface to show the category icons.

Open **MapsActivity.kt** and replace the call to `map.addMarker()` in `addPlaceMarker()` with the following:

```
val marker = map.addMarker(MarkerOptions()
    .position(bookmark.location)
    .title(bookmark.name)
    .snippet(bookmark.phone)
    .icon(bookmark.categoryResourceId?.let {
        BitmapDescriptorFactory.fromResource(it)
    })
    .alpha(0.8f))
```

The change here is that you're setting the icon to a bitmap which is loaded from the `categoryResourceId` property on the `bookmark`.

Build and run the app. Add bookmarks for a variety of place types and you'll see the different icons that are displayed on the map.

The next UI update is a simple one. You'll update the navigation drawer to display the new category icons.

Open **BookmarkListAdapter.kt**. In `onBindViewHolder()`, replace the call to `setImageResource()` with the following:

```
bookmarkViewData.categoryResourceId?.let {
    holder.categoryImageView.setImageResource(it) }
```

This first checks to see if the `categoryResourceId` has been set, and if so, it sets the image resource to the `categoryResourceId`.

Build and run the app. Open the navigation drawer and marvel at the beautiful category icons beside each bookmark!
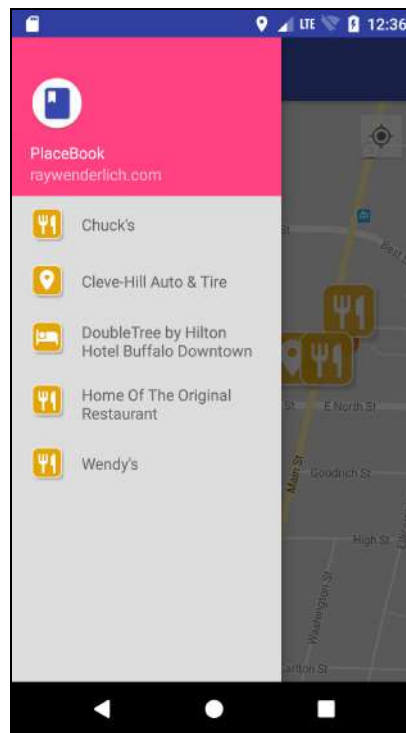
There's one last feature before moving on, and that's to allow the user to change the category assigned to a place.

You'll start by adding a new spinner UI widget to the bookmark details activity, allowing the user to select from the available categories.

Open **activity_bookmark_details.xml** and add the following after the closing </
LinearLayout> tag for the editTextName EditText control:

```xml
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">
  <TextView
      android:id="@+id/textViewCategoryLabel"
      style="@style/BookmarkLabel"
      android:layout_weight='0.4'
      android:text="Category"/>
  <ImageView
      android:id="@+id/imageViewCategory"
      android:layout_width="24dp"
      android:layout_height="24dp"
      android:src="@drawable/ic_other"
      android:layout_marginStart="16dp"
      android:layout_marginLeft="16dp"
      android:layout_gravity="bottom"
      />
  <Spinner
      android:id="@+id/spinnerCategory"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_weight='1.4'
      android:layout_marginStart="8dp"
      android:layout_marginEnd="8dp"
      android:layout_marginTop="16dp"
      />
</LinearLayout>
```

This defines a row underneath the bookmark name which displays the currently
selected category icon using an ImageView, and it allows the user to select a new
category using a Spinner.

Before you can use set the image and populate the spinner, you need to add support for
bookmark categories in the view model for the detail view.

Open **BookmarkDetailsViewModel.kt** and update the BookmarkDetailsView
declaration to include a category property:

```kotlin
data class BookmarkDetailsView(var id: Long? = null,
                               var name: String = "",
                               var phone: String = "",
                               var address: String = "",
                               var notes: String = "",
                               var category: String = "") {
```

Update the return call in bookmarkToBookmarkView() to include the category:

```kotlin
return BookmarkDetailsView(
    bookmark.id,
```

```
    bookmark.name,
    bookmark.phone,
    bookmark.address,
    bookmark.notes,
    bookmark.category
)
```

Update `bookmarkViewToBookmark()` to include the category assignment after the `bookmark.notes` assignment line:

```
bookmark.category = bookmarkDetailsView.category
```

Add a new method to return a category resource ID from a category name:

```
fun getCategoryResourceId(category: String): Int? {
  return bookmarkRepo.getCategoryResourceId(category)
}
```

This is a simple pass-through to a similar method in the bookmark repo.

In order to fill the spinner with options, you'll also need a method to return a list of all possible category names.

Open **BookmarkRepo.kt** and add the following property:

```
val categories: List<String>
  get() = ArrayList(allCategories.keys)
```

This defines a `get()` accessor on the `categories` property that will take all of the `HashMap` keys, which are the category names, and return them as an `ArrayList` of strings.

Open **BookmarkDetailsViewModel.kt** and add the following method:

```
fun getCategories(): List<String> {
  return bookmarkRepo.categories
}
```

This is another simple pass-through method that returns the categories list from the bookmark repo.

Open **BookmarkDetailsActivity.kt** and add the following new method:

```
private fun populateCategoryList() {
  // 1
  val bookmarkView = bookmarkDetailsView ?: return
  // 2
  val resourceId =
      bookmarkDetailsViewModel.getCategoryResourceId(
          bookmarkView.category)
  // 3
```

```
    resourceId?.let { imageViewCategory.setImageResource(it) }
    // 4
    val categories = bookmarkDetailsViewModel.getCategories()
    // 5
    val adapter = ArrayAdapter(this,
        android.R.layout.simple_spinner_item, categories)
    adapter.setDropDownViewResource(
        android.R.layout.simple_spinner_dropdown_item)
    // 6
    spinnerCategory.adapter = adapter
    // 7
    val placeCategory = bookmarkView.category
    spinnerCategory.setSelection(
        adapter.getPosition(placeCategory))
  }
```

1. The method returns immediately if `bookmarkDetailsView` is `null`.

2. The category icon `resourceId` is retrieved from the view model.

3. If the `resourceId` is not `null` `imageViewCategory` is updated to the category icon.

4. The list of categories is retrieved from the view model.

5. This is the standard way to populate a `Spinner` control in Android. You first create an adapter, in this case a simple `ArrayAdapter` built from the list of category names. Then, using `setDropDownViewResource()`, the adapter is assigned a standard built-in layout resource.

6. The adapter is then assigned to the `spinnerCategory` control.

7. `spinnerCategory` is updated to reflect the current category selection.

Add a call to `populateCategoryList()` in `getIntentData()` after the `populateImageView()` call:

```
  populateCategoryList()
```

Build and run the app. Open the details for a bookmark and you'll notice the spinner displays the assigned category and the appropriate icon is displayed to the left.
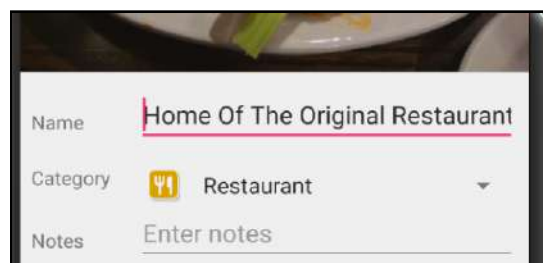
If you change the category and save the bookmark, you'll discover a couple of issues: the category icon does not update when the value is changed, and the category change is not saved. Let's fix that now.

Add the following to the end of `populateCategoryList()`:

```
// 1
spinnerCategory.post {
  // 2
  spinnerCategory.onItemSelectedListener = object :
AdapterView.OnItemSelectedListener {
    override fun onItemSelected(parent: AdapterView<*>, view: View,
position: Int, id: Long) {
      // 3
      val category = parent.getItemAtPosition(position) as String
      val resourceId =
bookmarkDetailsViewModel.getCategoryResourceId(category)
      resourceId?.let {
        imageViewCategory.setImageResource(it) }
    }
    override fun onNothingSelected(parent: AdapterView<*>) {
      // NOTE: This method is required but not used.
    }
  }
}
```

This new block of code is setting up a listener to respond when the user changes the category selection.

1. The need to use `spinnerCategory.post` is due to an unfortunate side effect in Android where `onItemSelected()` is always called once with an initial position of 0. This causes the spinner to reset back to the first category regardless of the selection you set programmatically.

    Using `post` causes the code block to be placed on the main thread queue, and the execution of the code inside the braces gets delayed until the next message loop. This eliminates the initial call by Android to `onItemSelected()`.

2. The `spinnerCategory onItemSelectedListener` property is assigned to an instance of the `onItemSelectedListener` class that implements `onItemsSelected()` and `onNothingSelected()`.

3. When the user selects a new category, `onItemSelected()` is called. The new `category` is determined by the current spinner selection position, and `imageViewCategory` is updated to reflect the new category.

Update `saveChanges()` to add the following line after the assignment of `bookmarkView.phone`:

```
bookmarkView.category = spinnerCategory.selectedItem as String
```

This grabs the currently selected category and assigns it to the `bookmarkView` category.

Build and run the app. This time the category icon on the details screen will update as you change selections, and the new category will persist when you save the changes.

Searching for places

What if the user is looking for a specific place and can't find it on the map? No worries! The Google Places API provides a powerful search feature that you'll take advantage of next. You'll add a new search button overlay on the map to trigger the search feature.

The Google Places API provides an autocomplete search widget that you can easily display within your app. As the user types in a place name or address, the search widget will display a dynamic list of choices.

> **Note**: If you want to completely customize the user experience, you can also use the autocomplete feature programmatically. See the developer document here https://developers.google.com/places/android-api/autocomplete#get_place_predictions_programmatically for more details.

You can choose to either embed the autocomplete widget as a fragment, or you can launch it as an activity with an intent. If you want a permanent search bar within your activity, then the fragment approach is more appropriate. In our case, a search button will be provided, and the autocomplete widget will be shown as an activity.

First, a method is needed to kick off the search feature.

Use PlaceAutocomplete search

Open **MapsActivity.kt** and the following property to the companion object:

```
private const val AUTOCOMPLETE_REQUEST_CODE = 2
```

Add the following method:

```
private fun searchAtCurrentLocation() {
  // 1
  val bounds = map.projection.visibleRegion.latLngBounds
```

```
   try {
     // 2
     val intent = PlaceAutocomplete.IntentBuilder(
         PlaceAutocomplete.MODE_OVERLAY)
         .setBoundsBias(bounds)
         .build(this)
     // 3
     startActivityForResult(intent, AUTOCOMPLETE_REQUEST_CODE)
   } catch (e: GooglePlayServicesRepairableException) {
     //TODO: Handle exception
   } catch (e: GooglePlayServicesNotAvailableException) {
     //TODO: Handle exception
   }
 }
```

1. The bounds of the current visible region of the map is computed.

2. `PlaceAutocomplete` provides an `IntentBuilder` method to build up the intent. `PlaceAutocomplete.MODE_OVERLAY` is passed to indicate that the search widget can overlay the current activity. The other option is `PlaceAutocomplete.FULLSCREEN`, which will cause the search interface to replace the entire screen.

   The map `bounds` is passed to `setBoundBias()`. This tells the search widget to look for places within the current map window before searching other areas.

3. The activity starts and is passed a request code of `AUTOCOMPLETE_REQUEST_CODE`. When the user finishes the search, the results will be identified by this request code.

The code is surrounded by a try/catch block because `IntentBuilder` can throw exceptions if Google Play services are not working.

Add the following method:

```
override fun onActivityResult(requestCode: Int, resultCode: Int,
    data: Intent?) {
  // 1
  when (requestCode) {
    AUTOCOMPLETE_REQUEST_CODE ->
      // 2
      if (resultCode == Activity.RESULT_OK && data != null) {
        // 3
        val place = PlaceAutocomplete.getPlace(this, data)
        // 4
        val location = Location("")
        location.latitude =  place.latLng.latitude
        location.longitude = place.latLng.longitude
        updateMapToLocation(location)
        // 5
        displayPoiGetPhotoMetaDataStep(place)
      }
  }
}
```

onActivityResult() is called by Android when the user completes the search.

1. First the requestCode is checked to make sure it matches the
    AUTOCOMPLETE_REQUEST_CODE passed into startActivityForResult().

2. If the resultCode indicates the user found a place, and the data is not null, then
    you continue to process the results.

3. How do you get the actual place that was found by the user? Fortunately, the
    PlaceAutocomplete class provides a handy method, getPlace(), that will take the
    data and return back a populated Place object.

4. The place latLng is converted to a location and passed to the existing
    updateMapToLocation method. This causes the map to zoom to the place.

5. Previously, when the user tapped on a place, several steps were created to process
    the data. In this case, you already have the place loaded, so you don't need all of the
    steps, but you can start at the displayPoiGetPhotoMetaDataStep() and pass it the
    found place. This will load the place photo and display the place info window.

Update the UI

Next, you'll surround the main map view with a frame layout and add a floating search
button on top of the map.

Open **main_view_maps.xml** and the following before the top <LinearLayout> line:

```
<FrameLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
```

Add the following after the closing </LinearLayout> line:

```
    <android.support.design.widget.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_margin="16dp"
        app:srcCompat="@drawable/ic_search_white"/>

</FrameLayout>
```

This tells the layout engine to place the search button in the bottom right corner of the
map, with a margin of 16dp on each side.

Now it's just a matter of connecting the button to a variable and listening for a user tap.

Open **MapsActivity.kt** and add the following to `setupMapListeners()`:

```
fab.setOnClickListener {
  searchAtCurrentLocation()
}
```

Build and run the app. Tap on the search icon and search for a place by name. Tap on one of the results and the map will zoom to the place and display the info window.
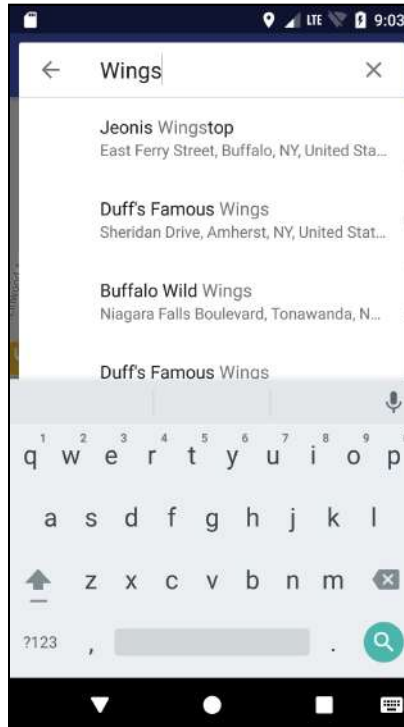
Create ad-hoc bookmarks

Google's database of places is impressive, but it's not perfect. What if the user wants to add a bookmark for a place that doesn't show up on the map? You'll make this possible by allowing the user to "drop a pin" at any location on the map.

Currently, the `MapsViewModel` class has a method to create a bookmark from a place, but now you need one to create a bookmark from a map location only.

Open **MapsViewModel.kt** and add the following method:

```kotlin
fun addBookmark(latLng: LatLng) : Long? {
  val bookmark = bookmarkRepo.createBookmark()
  bookmark.name = "Untitled"
  bookmark.longitude = latLng.longitude
  bookmark.latitude = latLng.latitude
  bookmark.category = "Other"
  return bookmarkRepo.addBookmark(bookmark)
}
```

This takes in a `LatLng` location and creates a new untitled bookmark at the given location. It returns the new bookmark ID to the caller.

Now you need a method in **MapsActivity.kt** to take advantage of this new method:

Open **MapsActivity.kt** and add the following method:

```kotlin
private fun newBookmark(latLng: LatLng) {
  launch(CommonPool) {
    val bookmarkId = mapsViewModel.addBookmark(latLng)
    bookmarkId?.let {
      startBookmarkDetails(it)
    }
  }
}
```

This method creates a new bookmark from a location and then starts the bookmark details activity to allow editing of the new bookmark. The call to `addBookmark` runs within a coroutine block, because it accesses the database and can't run on the main thread.

Now you just need to listen for the user to long tap on the map.

Add the following to `setupMapListeners()`:

```kotlin
map.setOnMapLongClickListener { latLng ->
  newBookmark(latLng)
}
```

Build and run the app. Long tap anywhere on the map, and the bookmark activity screen will display with a new untitled bookmark using a default photo.
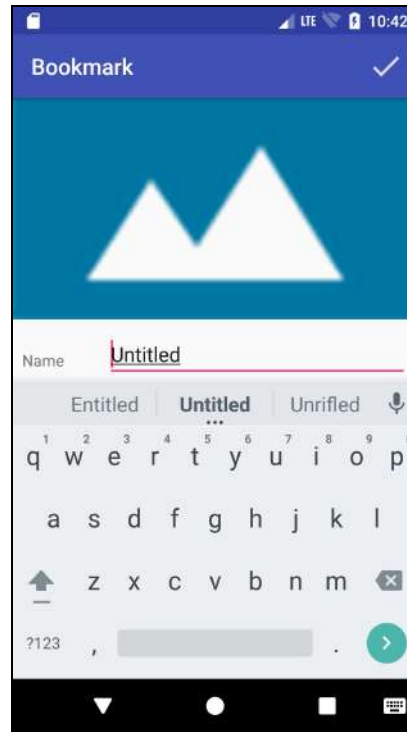
Name the bookmark, assign it a category and save the changes. The new bookmark will appear at the location where you tapped on the map.

Deleting bookmarks

Any full featured app needs to account for user mistakes. In PlaceBook, this means letting the user remove a bookmark that is no longer needed, or one that was added by accident. You'll add a trashcan action bar icon to the detail activity to let the user delete a bookmark.

Open **menu_bookmark_details.xml**. Add the following before the `action_save` `<item>`:

```xml
<item
    android:id="@+id/action_delete"
    android:icon="@android:drawable/ic_menu_delete"
    android:title="Delete"
    app:showAsAction="ifRoom"/>
```

This adds a delete icon (trashcan) to the action bar menu to the left of the save icon.

You'll work your way up from the bottom-level code to the top, adding in basic support for deleting bookmarks.

Create a new Kotlin file named **FileUtils.kt** in the **utils** package, and replace the
contents with the following:

```
object FileUtils {
  fun deleteFile(context: Context, filename: String) {
    val dir = context.filesDir
    val file = File(dir, filename)
    file.delete()
  }
}
```

This is a utility method that deletes a single file in the app's main files directory. This
will be used to delete the image associated with a deleted bookmark.

Open **Bookmark.kt** and add the following method:

```
fun deleteImage(context: Context) {
  id?.let {
    FileUtils.deleteFile(context, generateImageFilename(it))
  }
}
```

This method uses `FileUtils.deleteFile()` to delete **the image file** associated with the
current bookmark.

Open **BookmarkRepo.kt** and add the following method:

```
fun deleteBookmark(bookmark: Bookmark) {
  bookmark.deleteImage(context)
  bookmarkDao.deleteBookmark(bookmark)
}
```

This method deletes **the bookmark image** and **the bookmark** from the database.

Open **BookmarkDetailsViewModel.kt** and add the following method:

```
fun deleteBookmark(bookmarkDetailsView: BookmarkDetailsView) {
  launch (CommonPool) {
    val bookmark = bookmarkDetailsView.id?.let {
      bookmarkRepo.getBookmark(it)
    }
    bookmark?.let {
      bookmarkRepo.deleteBookmark(it)
    }
  }
}
```

This method takes in a `BookmarkView` and loads the bookmark from the repo. If the
bookmark is found, it calls `deleteBookmark()` on the repo. The code is wrapped in a
coroutine so it runs in the background.

Open **BookmarkDetailsActivity.kt** and add the following method:

```
private fun deleteBookmark()
{
  val bookmarkView = bookmarkDetailsView ?: return

  AlertDialog.Builder(this)
      .setMessage("Delete?")
      .setPositiveButton("Ok") { _, _ ->
        bookmarkDetailsViewModel.deleteBookmark(bookmarkView)
        finish()
      }
      .setNegativeButton("Cancel", null)
      .create().show()
}
```

This method displays a standard `AlertDialog` to ask the user if they want to delete the bookmark.

If they select OK, the bookmark is deleted and the activity is closed using `finish()`.

All the support code is in place, now you just need to respond to the delete menu action.

In `onOptionsItemSelected()` add the following additional case to the when statement before the final `else`:

```
R.id.action_delete -> {
  deleteBookmark()
  return true
}
```

This calls `deleteBookmark()` when the delete icon is tapped.

Since you are deleting a bookmark that's being observed with `LiveData`, some precautions are needed to prevent a crash.

Open **BookmarkDetailsViewModel.kt** and update `mapBookmarkToView()` as follows:

```
fun mapBookmarkToBookmarkView(bookmarkId: Long) {
  val bookmark = bookmarkRepo.getLiveBookmark(bookmarkId)
  bookmarkDetailsView = Transformations.map(bookmark) { bookmark ->
      bookmark?.let {
        val bookmarkView = bookmarkToBookmarkView(bookmark)
        bookmarkView
      }
    }
}
```

This ensures that a `null` bookmark is not passed to `bookmarkToView` by using the `bookmark?.let` statement.

Build and run the app. Edit an existing bookmark and use the delete icon to delete it. The bookmark will be deleted, and you'll return to the map activity.
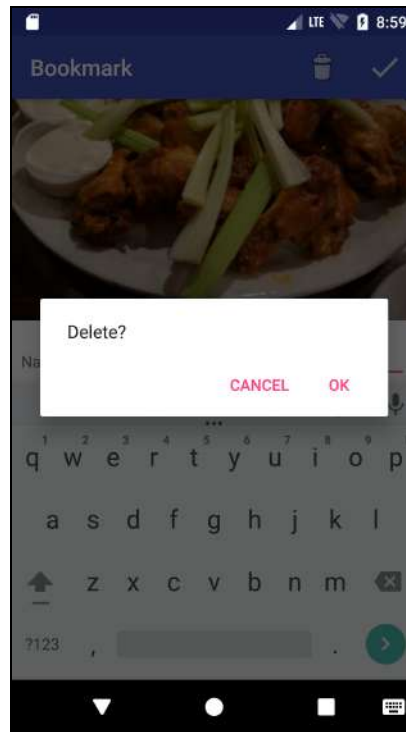
Sharing bookmarks

Your users have painstakingly bookmarked some fantastic places, so why not let them share their good finds with friends?

Android allows you to share data with other apps using an intent with an **ACTION_SEND** action. All you need to do is provide the data. Android will figure out the apps that support your data type and present the user with a list of choices.

You'll build out an intent that shares a URL providing directions to the bookmark place.

Open **BookmarkDetailsViewModel.kt** and update the `BookmarkDetailsView` declaration as follows:

```
data class BookmarkDetailsView(var id: Long? = null,
                               var name: String = "",
                               var phone: String = "",
                               var address: String = "",
                               var notes: String = "",
                               var category: String = "",
                               var longitude: Double = 0.0,
                               var latitude: Double = 0.0,
                               var placeId: String? = null) {
```

This adds longitude, latitude and placeId properties.

Update the return statement in `bookmarkToBookmarkView()` as follows:

```
return BookmarkDetailsView(
    bookmark.id,
    bookmark.name,
    bookmark.phone,
    bookmark.address,
    bookmark.notes,
    bookmark.category,
    bookmark.longitude,
    bookmark.latitude,
    bookmark.placeId
)
```

The new longitude, latitude and placeId values are added to the `BookmarkView` call.

Open **BookmarkDetailsActivity.kt** and add the following method:

```
private fun sharePlace() {
  // 1
  val bookmarkView = bookmarkDetailsView ?: return
  // 2
  var mapUrl = ""
  if (bookmarkView.placeId == null) {
    // 3
    val location = URLEncoder.encode("${bookmarkView.latitude},"
        + "${bookmarkView.longitude}", "utf-8")
    mapUrl = "https://www.google.com/maps/dir/?api=1" +
        "&destination=$location"
  } else {
    // 4
    val name = URLEncoder.encode(bookmarkView.name, "utf-8")
    mapUrl = "https://www.google.com/maps/dir/?api=1" +
        "&destination=$name&destination_place_id=" +
        "${bookmarkView.placeId}"
  }
  // 5
  val sendIntent = Intent()
  sendIntent.action = Intent.ACTION_SEND
  // 6
  sendIntent.putExtra(Intent.EXTRA_TEXT,
      "Check out ${bookmarkView.name} at:\n$mapUrl")
  sendIntent.putExtra(Intent.EXTRA_SUBJECT,
      "Sharing ${bookmarkView.name}")
  // 7
  sendIntent.type = "text/plain"
  // 8
  startActivity(sendIntent)
}
```

1.   An early return is taken if `bookmarkView` is `null`.

2. This section of code builds out a Google Maps URL to trigger driving directions to the bookmarked place. Check out the documentation at https://developers.google.com/maps/documentation/urls/guide for details about constructing map URLs.

   There are two different styles of URL to use depending on whether a place ID is available. If the user created an ad-hoc bookmark, then the directions will go directly to the latitude/longitude of the bookmark. If the bookmark was created from a place, then the directions will go to the place based on its ID.

3. A string with the latitude/longitude separated by a comma is constructed. It's encoded to allow the command to work in the URL. The final `mapUrl` is constructed using the `location` string. The final URL string will look like this: https://www.google.com/maps/dir/?api=1&destination=-84.56536026895046%2C35.+351035752390054

4. For the option with the place ID available, the destination will contain the place name. The name string is URL encoded to make the input safe. The final `mapUrl` is constructed using the `name` string and the place ID. The final URL string will look like this: https://www.google.com/maps/dir/?api=1&destination=Riverstone+Plaza&destination_place_id=ChIJAAAAAAAAAAR1tSJBrRUoKI

5. The sharing activity `Intent` is created and the action set to `ACTION_SEND`. This tells Android that this `Intent` is meant to share its data with another application installed on the device.

6. There are multiple types of extra data that can be added to the intent. The application that receives the intent can choose which of the data items to use and which to ignore. For example, an email app will use the `ACTION_SUBJECT`, but a messaging app will likely ignore it. There are several other extras available including `EXTRA_EMAIL`, `EXTRA_CC`, and `EXTRA_BCC`.

7. The intent type is set to a MIME type of "text/plain". This instructs Android that you intend to share plain text data. Any application in the system that registers an intent filter for the "text/plain" MIME type will be offered as a choice in the share dialog. If you were sharing binary data such as an image, you might use an MIME type of "image/jpeg".

8. Finally the sharing activity is started.

Now you'll add a floating share button to trigger the `sharePlace` method. Because you'll use the same technique as you did when adding the search button on the map activity, we'll move through this with minimal explanation.

Open **activity_bookmark_details.xml** and replace the top `<LinearLayout>` with the following:

```
<FrameLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

<LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
```

Add the following after the closing `</LinearLayout>` line:

```
    <android.support.design.widget.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:layout_gravity="bottom|end"
        app:srcCompat="@android:drawable/ic_dialog_email"/>

</FrameLayout>
```

Open **BookmarkDetailsActivity.kt** and add the following method:

```
private fun setupFab() {
   fab.setOnClickListener { sharePlace() }
}
```

Add the following to the end of `onCreate()`:

```
setupFab()
```

Build and run the app. Open a bookmark and tap the sharing button. You should see a share dialog similar the following. Your actual choices will vary depending on the applications installed on your device.
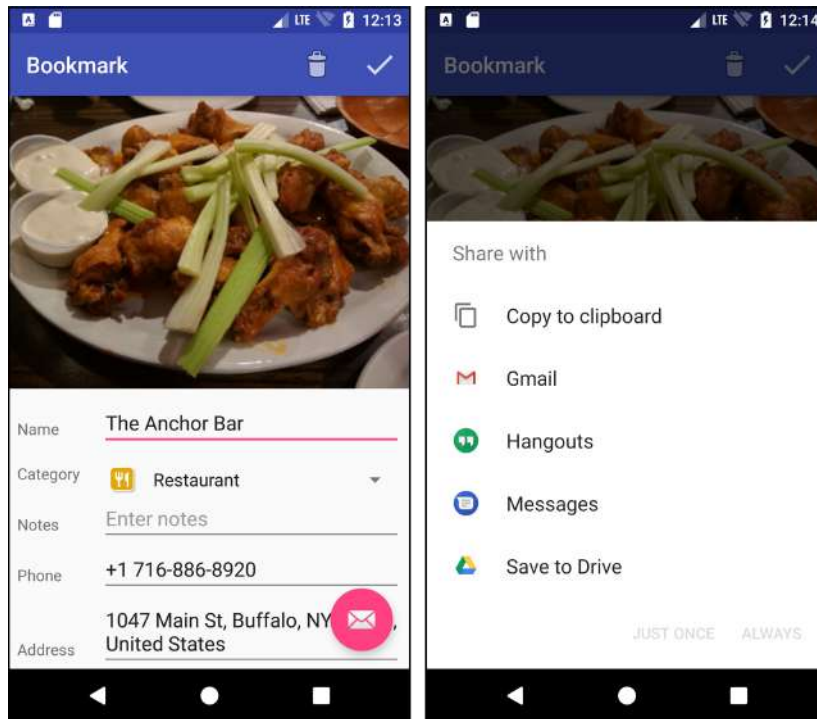
Tap on Gmail and it will launch the Gmail app and populate the subject and message body.
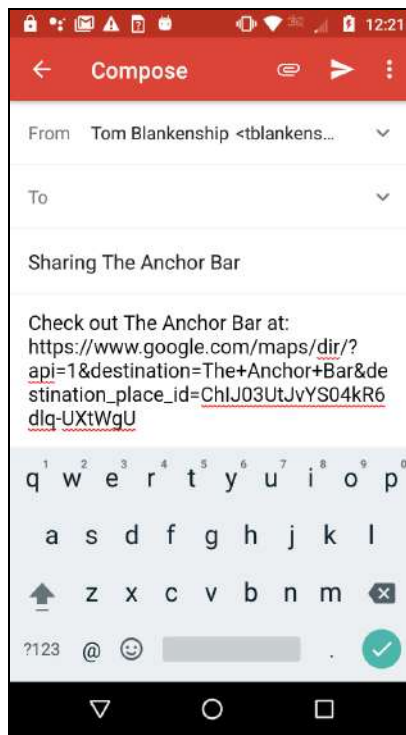
Color scheme

It's a minor change, but updating the color scheme to match our bookmark icon colors will make the app look much better.

Open **values/colors.xml** and update the three colors:

```xml
<color name="colorPrimary">#3748AC</color>
<color name="colorPrimaryDark">#2A3784</color>
<color name="colorAccent">#E3A60B</color>
```

The `Primary` color is a nice shade of blue and will be used by the main action bar. The `PrimaryDark` color is used by the status bar at the top and is a slightly darker version of the `Primary` color. The `Accent` color matches the yellow color of the bookmark icons. It will be used by the floating buttons and the highlight color when a field is in focus.

Build and run the app. The overall app colors should look much better now.
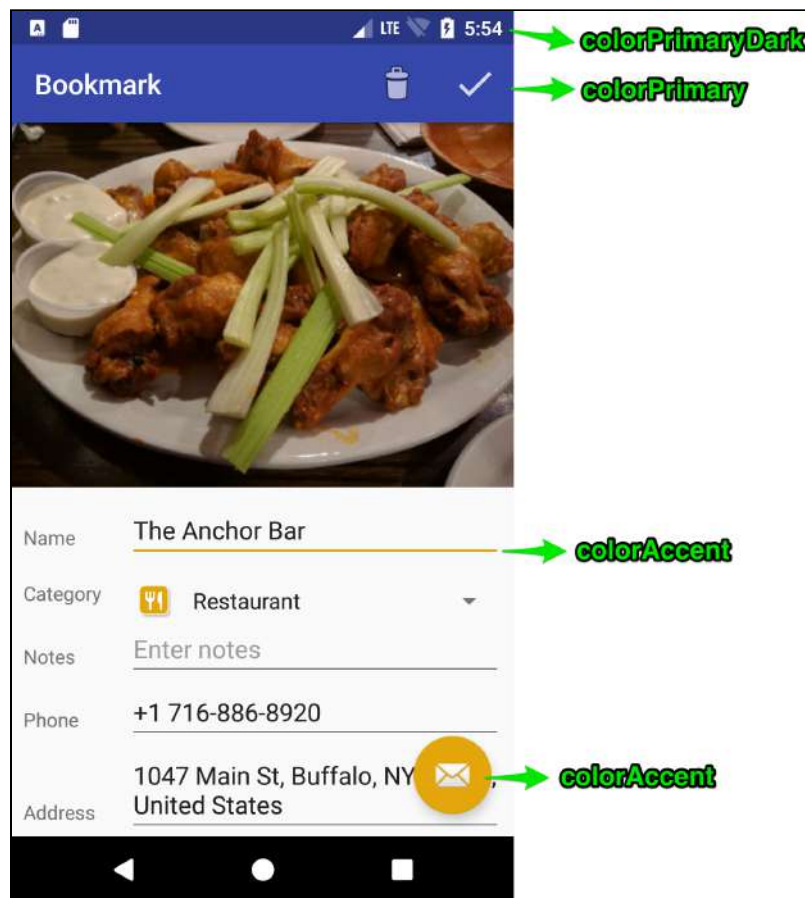
Progress indicator

It's always good practice to let the user know when a potentially long running operation is in progress. It also makes sense to prevent user interaction during this time. You'll accomplish both of these tasks next.

Open **main_view_maps.xml** and add the following before the final `</FrameLayout>`:

```xml
<ProgressBar
    android:id="@+id/progressBar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:visibility="gone"/>
```

This creates a hidden progress bar at the center of the `Activity`. In this case, "progress bar" is not the most appropriate term since what gets displayed is actually a circular progress indicator.

Open **MapsActivity.kt** and add the following new methods:

```kotlin
private fun disableUserInteraction() {
  window.setFlags(WindowManager.LayoutParams.FLAG_NOT_TOUCHABLE,
      WindowManager.LayoutParams.FLAG_NOT_TOUCHABLE)
}

private fun enableUserInteraction() {
  window.clearFlags(
      WindowManager.LayoutParams.FLAG_NOT_TOUCHABLE)
}
```

`disableUserInteraction()` sets a flag on the the main window to prevent user touches.

`enableUserInteraction()` clears the flag set by disableUserInteraction().

Add the following new methods:

```kotlin
private fun showProgress() {
  progressBar.visibility = ProgressBar.VISIBLE
  disableUserInteraction()
}

private fun hideProgress() {
  progressBar.visibility = ProgressBar.GONE
  enableUserInteraction()
}
```

`showProgress()` makes the progress bar visible and disables user interaction.

`hideProgress()` hides the progress bar and enables user interaction.

Now you just need to show and hide the progress bar in a few strategic locations.

You want to show progress when a place or place photo is being loaded. You must ensure that all calls to `showProgress()` are matched with a call to `hideProgress()` or the UI will remain frozen.

Add a call to `showProgress()` as the first line in `displayPoi()`:

```
showProgress()
```

This will display the progress bar when a place is tapped.

Add a call to `showProgress()` in `onActivityResult()`, after the call to `updateMapLocation()`:

```
showProgress()
```

This will display the progress bar after searching for a place but before the place photo is loaded.

That's it for showing the progress bar. Now you just need to ensure that it goes away whether the place is successfully loaded or not.

Add a call to `hideProgress()` in `displayPoiGetPlaceStep()`, after the call to `Log.e()`:

```
hideProgress()
```

This will hide the progress bar if the place cannot be retrieved and the `displayPoi` steps end here.

In `displayPoiGetPhotoMetaDataStep()`, add an `else` statement containing a call to `hideProgress()` for the `if (photoMetadataBuffer.count > 0)` clause.

```
} else {
  hideProgress()
}
```

In `displayPoiGetPhotoMetaDataStep()`, add an `else` statement containing a call to `hideProgress()` for the `if (placePhotoMetadataResult.status.isSuccess)` clause.

```
} else {
  hideProgress()
}
```

Both of these will hide the progress bar if there is no photo and the displayPoi steps end here.

In `displayPoiGetPhotoStep()` add a call to `hideProgress()` as the first line in the
`getScaledPhoto()` result callback:

```
hideProgress()
```

This will hide the progress bar after the final step to load the place photo is completed.

Build and run the app. Tap on a place to see the progress bar. Depending on the speed
of your internet connection, it may flash almost too quickly to see, or it may spin for a
couple of seconds.
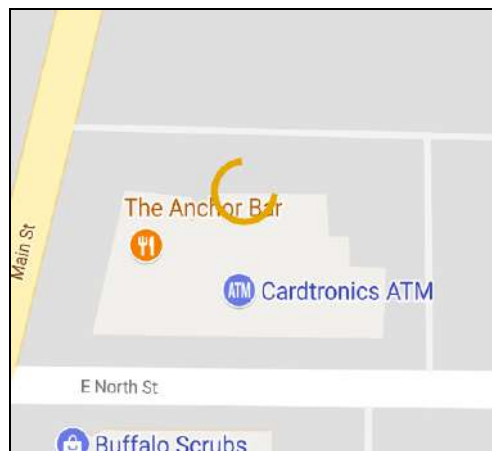
Where to go from here?

Congratulations! You've made it through the entire PlaceBook application section.
You've built a useful map-based application and learned a lot of new concepts along the
way.

In the following section, you'll take your Android skills to the next level and learn about
networking, media playback and more. Give yourself a well deserved break, and then
move on to the next section when you're ready!

Section IV: Building a Podcast Manager and Player

This section gets a bit more advanced. You're going to build a podcast manager and player app named **PodPlay**. You'll cover networking, working with REST and XML, and the Android media libraries.

Chapter 20: Networking

Chapter 21: Finding Podcasts

Chapter 22: Podcast Details

Chapter 23: Podcast Episodes

Chapter 24: Podcast Subscriptions Part One

Chapter 25: Podcast Subscriptions Part Two

Chapter 26: Podcast Playback

Chapter 27: Episode Player

Chapter 20: Networking

By Tom Blankenship

In this section, you're going to utilize many of the skills you've already learned and dive into some more advanced areas of Android development. You'll build a full-featured Podcast manager and player app named **PodPlay**. This app will allow searching and subscribing to podcasts from iTunes and provide a playback interface with speed controls.

The following new topics will be covered:

- Android networking

- Retrofit REST API library

- XML Parsing

- Search activity

- MediaPlayer library

Getting started

The app will contain these main features:

1. Quick searching of podcasts by keyword or name.
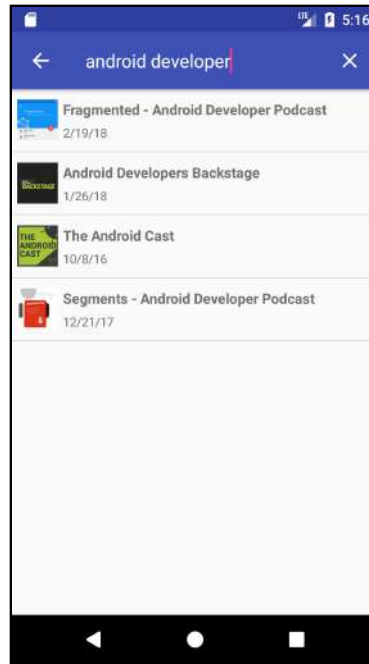
2. Display for previewing podcast episodes.
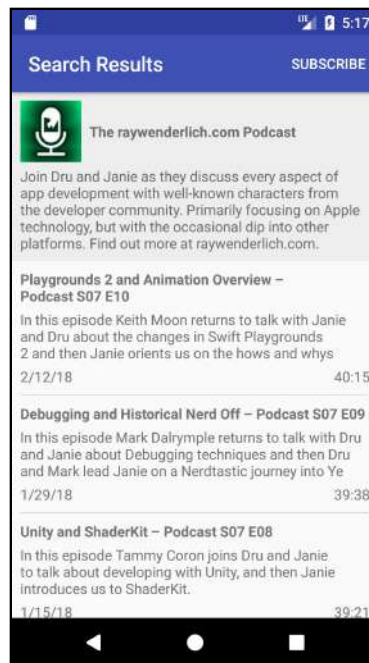
3.   Playback of audio and video podcasts.

4.   Subscribing to your favorite podcasts.
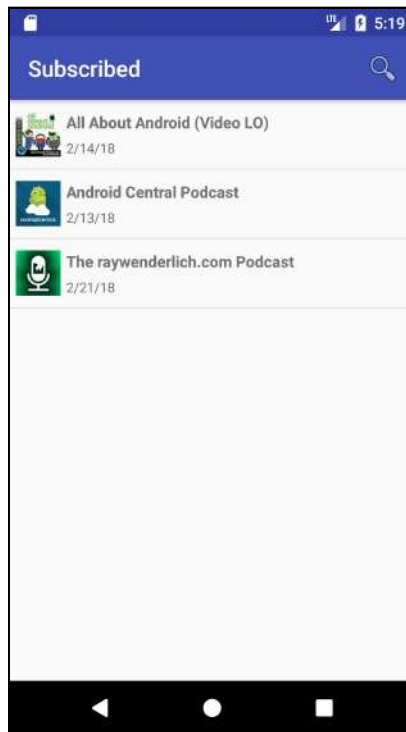
5.   Playback at various speeds.

Project set up

You'll start by creating a project with a single empty activity. This app will use the same structure as MapBook, but it will add a new services layer.

Open Android Studio and select **Start a new Android Studio project**.

Fill out the **Create Android Project** dialog:

- Application name: PodPlay

- Company domain: raywenderlich.com

- Project Location: *Select your own location*

- Include C++ support: unchecked

- Include Kotlin support: checked

Click **Next**.

Fill out the **Target Android Devices** dialog:

- Phone and Tablet: checked

- Minimum SDK: API 19

- Leave everything else unchecked.

Click **Next**.

In the **Add an Activity to Mobile** dialog, select **Empty Activity**, and then click **Next**.

Fill out the **Configure Activity** dialog:

- Activity Name: PodcastActivity

- Generate Layout File: checked

- Layout Name: activity_podcast

- Backwards Compatibility (AppCompat): checked

Click **Finish**. Next, under the **com.raywenderlich.podcast** package, create a new package named **ui**.
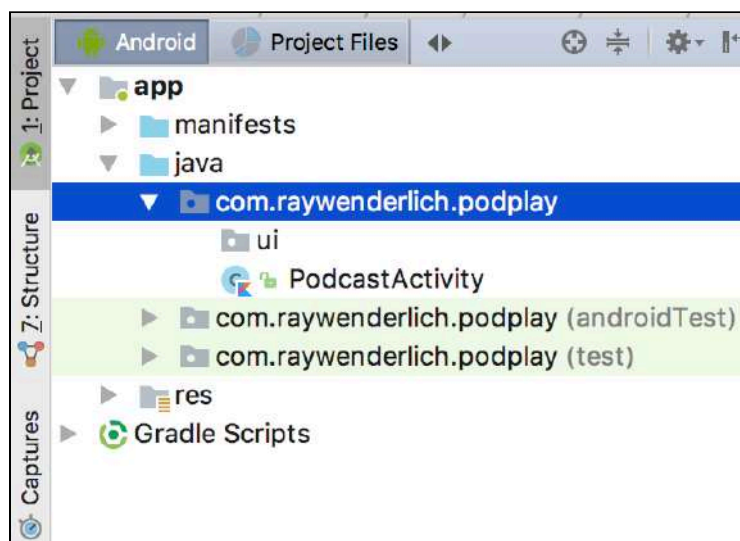
Move the **PodcastActivity.kt** file into the ui package using drag-and-drop. In the Move refactor dialog, leave all of the options set to their defaults.

If you have the option to "Compact/Hide" empty middle packages turned on in the Android Project view, then you'll see something like this:
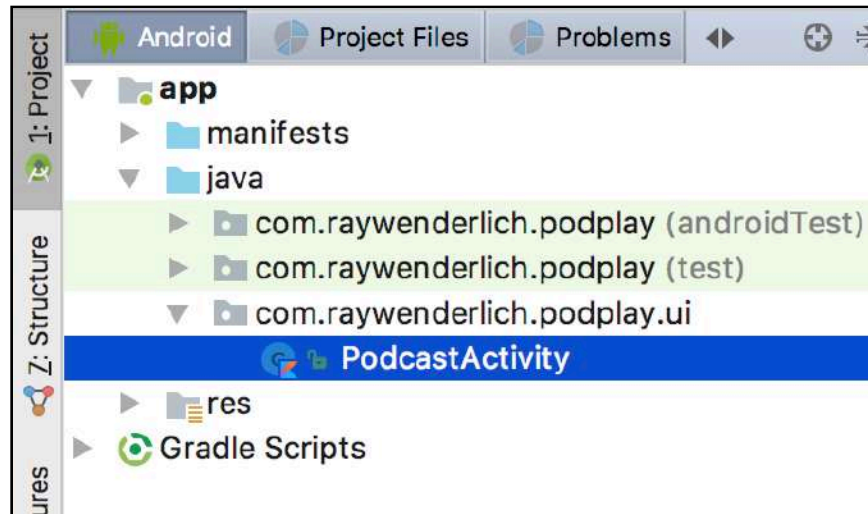
If you have the option to "Compact/Hide" empty middle packages turned off, it will look like this:
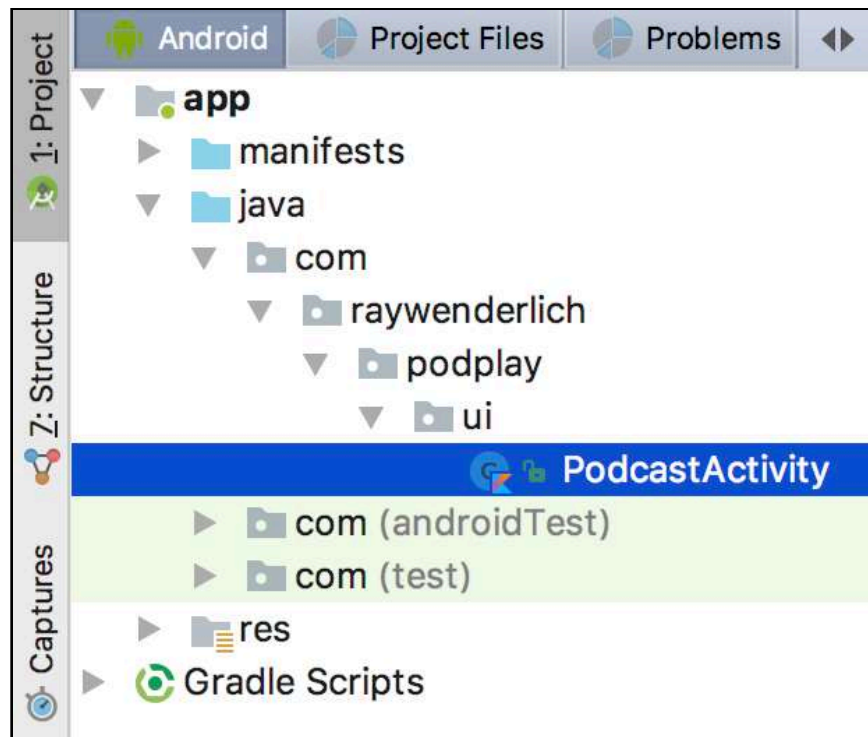
Where are the podcasts?

Before we get to the fun part of podcast playback, we need to answer a fundamental question: where do podcasts come from? The answer is just about anywhere. Podcasts are distributed using a standard format called RSS (Rich Site Summary, commonly referred to as Really Simple Syndication).

RSS feeds are based on the standard XML format and are used by websites to deliver a variety of content feeds. Most podcast feeds can be found on the main website that promotes or produces the podcast. There's normally a feed button that will provide a URL to the podcast feed.

In the XML returned by the RSS feed, you have access to a lot of information regarding the podcast, such as, the title of the podcast, the date is was published, associated artwork, a descriptive summary of the podcast and, obviously, a link to the audio file where the podcast is hosted.

For a podcast management app like PodPlay, it would great if there was a consolidated listing of all of the various podcast feeds spread throughout the internet. As it turns out, just about every podcast in existence is available through the iTunes podcast directory. Apple provides an API that you'll use to allow the user to search for podcast by keywords, making it easy to subscribe to a podcast.

Android networking

All of the apps you've built during your apprenticeship so far have been self-contained. They have not had to directly access any remote or network-based services. Although PlaceBook did access Google Places and download place photos, that was all handled by the Play Services library. That's all about to change with the PodPlay app.

PodPlay will require direct access to the iTunes podcast directory, as well as the ability to download individual RSS feeds. As with database access operations, network access operations are required to run in the background on Android. If you attempt to perform network operations on the UI thread, you'll be shamed with a **NetworkOnMainThreadException** error.

There are several built-in ways to handle network access in the background, including:

- AsyncTask

- Handler

- IntentService

- AsyncTaskLoader

- Executor

- JobScheduler

- Coroutines

Each of these options has a different level of complexity and its own benefits and drawbacks. The alternative is a third party library that handles the details and lets us concentrate on building app functionality.

There are few obvious choices in this area:

- **Volley**: Google provided library with a simple interface for accessing network resources asynchronously.

- **OkHttp**: Similar to Volley, and developed by Square Engineering.

- **Retrofit**: Also developed by Square Engineering, it builds on top of OkHttp.

You'll be using Retrofit for PodPlay. It's a popular library that makes it easy to do asynchronous network calls and process JSON data into model objects.

> **Note**: Although RSS feeds are formatted using an XML structure, iTunes will return a list of these feeds with a JSON structure.

PodPlay architecture

Continuing with our layered architecture, you'll create a service layer that will handle all network access to iTunes and hide the details of that communication. This will make it easy to swap out different methods for network access, without affecting any other parts of the code.

You'll start by creating a single service to search the iTunes podcast directory. This will be called when the user searches for podcasts in the app.

iTunes search service

If you regularly listen to, or have ever created a podcast, you're probably familiar with the iTunes podcast directory. This provides a single place to find just about any podcast from a huge variety of categories.

Lucky for us, Apple also provides an API to allow searching the podcast directory. The full API documentation can be found here:

https://affiliate.itunes.apple.com/resources/documentation/itunes-store-web-service-search-api/

There are a variety of options when calling the API, and it supports many types of media besides podcasts. The method you're interested in allows searching for podcasts by titles or keywords. It looks like this:

https://itunes.apple.com/search?term=Android+Developer&media=podcast

The `media=podcast` part tells iTunes to only search for podcasts. `term=Android+Developer` is the search term. The plus sign is used because the search term must be URL-encoded. URL-encoding replaces all spaces with plus symbols, and encodes all other special characters except: letters, numbers, periods (.), dashes (-), underscores (_) and asterisks (*).

You can plug this URL into your browser and get back the search results, but a better way to explore web APIs is to use the excellent open source Postman app. You can find Postman at **https://www.getpostman.com**. Download and install Postman for your OS and launch the app.

Using the default GET method, put in the search URL from above and click **Send**.
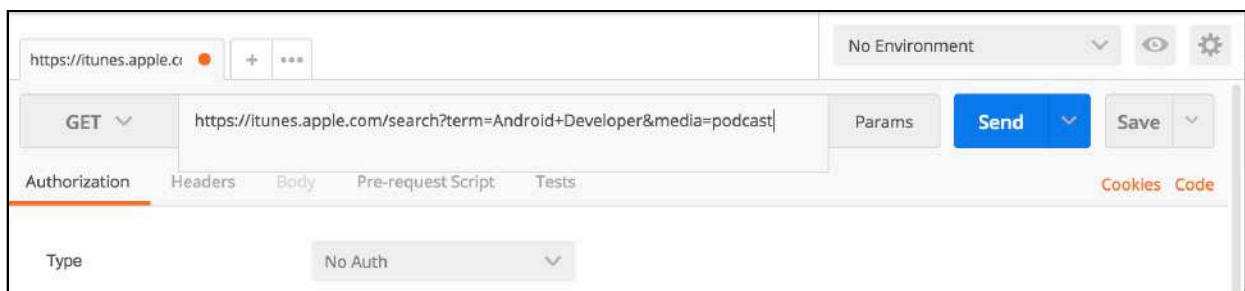
In the search results, set the output type to JSON and turn off line wrapping. You'll end up with a nicely formatted JSON display:

Scroll through the **results** array in the JSON output. There's a lot of information for each found podcast, but you'll only use a small number of items to display the search results to the user.

Retrofit

Now that you know how to get search results, the next step is to turn them into data models.

If you manually performed the steps to download and convert to a model, it would look something like this:

1. Initiate a network request to the iTunes search URL in a background process.

2. Capture the response to the network request as a JSON formatted string.

3. Parse the string based on JSON formatting rules.

4. Create a `PodcastResponse` object for each podcast item, and set the properties from the JSON data.

Here is a visual picture of mapping the JSON response to a `PodcastResponse` data model:
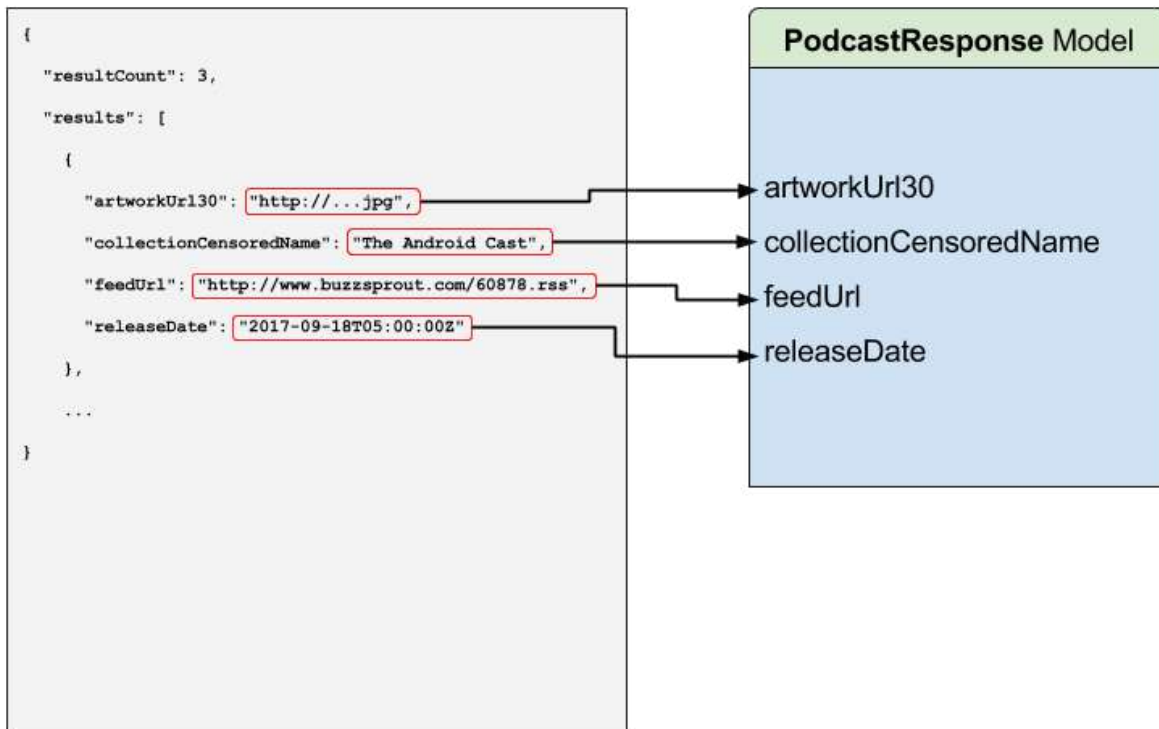
This is where **Retrofit** swoops in and makes your development life much easier! Retrofit lets you define a Kotlin interface that is a direct representation of the API you're accessing. Once you have defined the interface, you use the Retrofit **Builder** to create a concrete implementation of the interface. With the implementation in hand, you can make calls to the API and get back ready-to-use response objects.

Retrofit performs this magic with the help of **Annotations**. The annotation data is used by Retrofit to determine how to call the API endpoints and parse the returned data into model objects.

You'll create a simple service that encapsulates everything needed to define the service interface, and build the service implementation with Retrofit.

Defining Retrofit dependencies

First you need to define the Retrofit dependency.

Open the project **build.gradle** file and replace the `ext.kotlin_version` line with the following:

```
ext {
  kotlin_version = '1.2.21'
  retrofit_version = '2.3.0'
}
```

Open the app **build.gradle** file and add the following lines to the dependencies section:

```
implementation "com.squareup.retrofit2:retrofit:$retrofit_version"
implementation "com.squareup.retrofit2:converter-gson:$retrofit_version"
```

The `retrofit` dependency is the core Retrofit library. The `converter-gson` dependency adds support for JSON parsing.

A warning about changing Gradle files will be shown at the top of the editor. Click on **Sync Now**.

Creating the podcast response model

Now you'll create the model that represents a response from the iTunes service.

Create a new package named **service** under the project root.

> **Note**: In order to create the package under **com.raywenderlich.podplay,** you may need to change the settings in the project window to disable "**Compact/Hide Empty Middle Packages**".
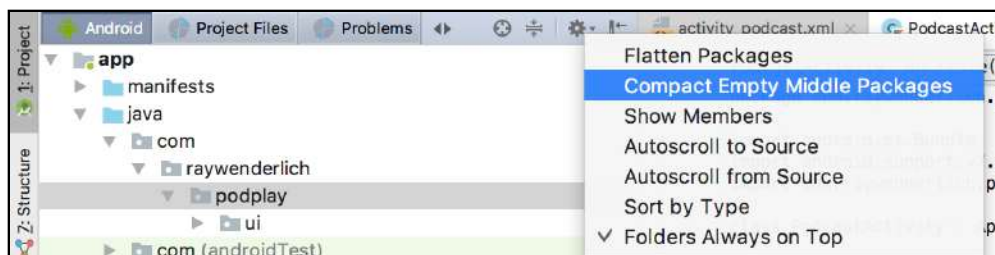
Once you add the **service** package, you can re-enable the **Hide/Compact Empty Middle Packages**, and your project structure should look like this:
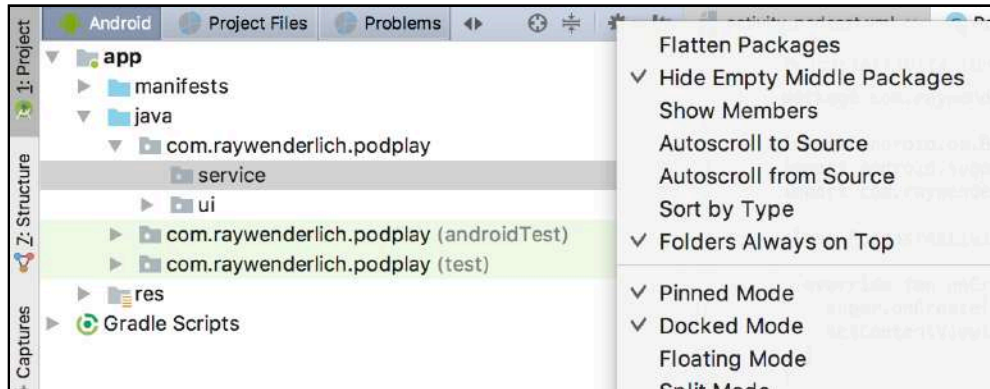
In the service package, create a new Kotlin file named **PodcastResponse.kt**, and then replace the contents with the following:

```kotlin
data class PodcastResponse(
    val resultCount: Int,
    val results: List<ItunesPodcast>) {

  data class ItunesPodcast(
      val collectionCensoredName: String,
      val feedUrl: String,
      val artworkUrl30: String,
      val releaseDate: String
  )
}
```

This defines a data class that directly mirrors the layout and hierarchy of the JSON data returned by the iTunes search API. Notice the variable names exactly match the keys in the iTunes search JSON data. While it's possible to use Annotations to allow different variable names than the JSON keys, this way is the most compact method to define the model. Also, it's not a problem to leave out the fields you don't need — the JSON parser used by Retrofit will ignore extra fields.

> **Note**: You may be wondering why the `PodcastResponse` model was created in the service package instead of a separate model package. This is really a matter of personal preference, but this particular model will be limited to handling responses from the iTunes Service, so it makes sense to keep it in the service package.

In the service package, create a new Kotlin file named **ItunesService.kt**, and then replace the contents with the following:

```kotlin
interface ItunesService {
  // 1
  @GET("/search?media=podcast")
  // 2
  fun searchPodcastByTerm(@Query("term") term: String):
      Call<PodcastResponse>
  // 3
  companion object {
    // 4
    val instance: ItunesService by lazy {
      // 5
      val retrofit = Retrofit.Builder()
          .baseUrl("https://itunes.apple.com")
          .addConverterFactory(GsonConverterFactory.create())
          .build()
      // 6
      retrofit.create<ItunesService>(ItunesService::class.java)
    }
  }
}
```

> **Note**: If you have any unresolved references, with multiple resolutions, make sure to resolve them from the retrofit library.

This defines an interface with a single method `searchPodcastByTerm`. This interface also contains a companion object that returns an instance of the interface as a singleton. This ensures that the interface is only instantiated once during the app's lifetime.

Let's go through this in detail:

1. This is your first encounter with a Retrofit annotation. Annotations always start with the @ symbol. This annotation is a "function" annotation, meaning that it applies to a function.

    Retrofit defines several function annotations that represent standard HTTP requests such as GET, POST and PUT. The `@GET` annotation takes a single parameter: the *path* of the endpoint that should be called. The annotation applies to the function that immediately follows.

2. The method `searchPodcastByTerm` takes a single parameter that has a Retrofit `@Query` annotation. This annotation tells Retrofit that this parameter should be added as a query term in the path defined by the `@GET` annotation. The annotation takes a single parameter that represents the name of the query term.

    You should always wrap the return type with the `Call` interface. When you call

searchPodcastByTerm(), it doesn't directly call the URL defined by the function annotation. Instead, it returns a Call object that then allows you to synchronously or asynchronously invoke the URL and get back a Response object containing the PodcastResponse.

As an example, calling **searchPodcastByTerm("Android Developer")** will result in Retrofit using a final URL of **/search?media=podcast&term=Android+Developer**. Retrofit will automatically URL-Encode the parameter names and values when constructing the URL.

3. A companion object is defined in the **ItunesService** interface.

4. The instance property of the companion object will hold the one and only application-wide instance of the ItunesService. This property looks a little different than ones you've defined in the past — and for good reason.

This definition allows the instance property to return a **Singleton** object. When the application needs to use ItunesService, it will simply reference ItunesService.instance.

Singleton objects are objects that have a single instance for the lifetime of the application. No matter how many times the instance property is accessed, it will only perform the initialization one time and will always return the same ItunesService object.

This is accomplished by using a Kotlin concept known as **property delegation**. As the name implies, property delegation allows you to delegate the property setters and getters to a class.

You specify a property delegate with the keyword by, followed by a delegate class instance. Here is a simple example (don't type this in code):

```kotlin
class SomeClass: {
    val someProperty: String by SomeDelegateClass()
}
```

The SomeDelegateClass class must provide a setValue() and a getValue() method. get() and set() for someProperty will be delegated to the setValue() and getValue() methods. Here's a simple implementation of the SomeDelegateClass class (don't type this in code):

```kotlin
class SomeDelegateClass {
    operator fun getValue(thisRef: Any?, property: KProperty<*>):
        String {
      return "A delegated return value"
    }
```

```
    operator fun setValue(thisRef: Any?, property: KProperty<*>,
        value: String) {
    // No body required
    }
  }
```

You won't be using a custom delegate class for PodPlay, but if want to learn more, please refer to https://kotlinlang.org/docs/reference/delegated-properties.html.

Kotlin provides some standard delegates that also come in handy. The one used for the `instance` property is the `Lazy<T>` delegate, and it's accompanied by the built-in `lazy` method. The `lazy` method takes a lambda and returns an instance of `Lazy<T>`.

The end result of using the lazy method is that the first time the instance property is accessed, it executes the lambda and stores the result (an instance of `ItunesService`). All subsequent calls to the instance property return the original result.

5. This is the first part of the lazy lambda method. `Retrofit.Builder()` is used to create a retrofit builder object. `Retrofit.Builder` allows you specify several options that let Retrofit know how it should ultimately create the concrete implementation of the `ItunesService` interface. In this case, you are specifying the following options:

    baseUrl: sets the base URL for the service. This will be prepended to the *path* specified in the function annotations.

    addConverterFactory: adds a converter factory to handle the translation of the JSON data to the `PodcastResponse` model object. A number of converter factories are available, but you'll use `GsonConverterFactory` to create an instance of the Gson Converter to handle the JSON parsing and conversion. Gson is a library developed by Google used to convert between Java objects and JSON.

6. Finally, `create<ItunesService>()` is called on the `retrofit` builder object to create the `ItunesService` instance. Since this is the last line evaluated in the lambda, it's used as the value assigned to the `instance` property.

Next, you'll add some temporary code in the **PodcastActivity.kt** to test the service call, but first you need to give the app permission to use the internet.

Open **AndroidManifest.xml** and add the following before the `<Application>` section:

```
  <uses-permission android:name="android.permission.INTERNET"/>
```

Open **PodcastActivity.kt** and add the following to the `onCreate()`:

```kotlin
val TAG = javaClass.simpleName

val itunesService = ItunesService.getServiceInstance()

val podcastCall = itunesService.searchPodcastByTerm(
    "Android Developer")

podcastCall.enqueue(object : Callback<PodcastResponse> {
  override fun onFailure(call: Call<PodcastResponse>?,
      t: Throwable?) {
    Log.i(TAG, "Call to ${call?.request()?.url()} " +
        "failed with ${t.toString()}")
  }

  override fun onResponse(call: Call<PodcastResponse>?,
                          response: Response<PodcastResponse>?) {
    Log.i(TAG, "Got response with status code " +
        "${response?.code()} and message " +
        "${response?.message()}")
    val body = response?.body()
    Log.i(TAG, "Response body = $body")
  }
})
```

Note: If you have any unresolved references with multiple choices for resolving, make sure to resolve them from the retrofit library.

This code will be explained further in the next section.

Build and run the app. The default "Hello World" screen will display.
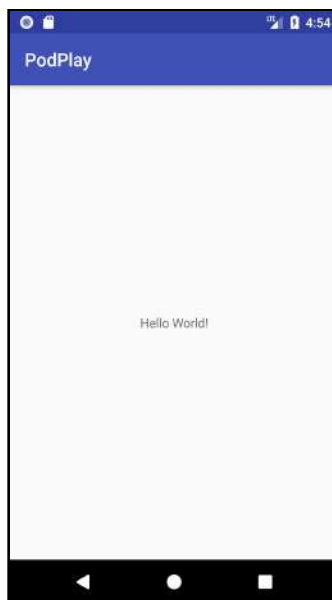
Check the Logcat window and you should see log results for I/PodcastActivity similar to this:

```
I/PodcastActivity: Got response with status code 200 and message OK
I/PodcastActivity: Response body = PodcastResponse(resultCount=3,
results=[ItunesPodcast(collectionCensoredName=Fragmented – Android
Developer Podcast, feedUrl=https://rss.simplecast.com/podcasts/1684/rss,
artworkUrl30=http://is5.mzstatic.com/image/thumb/Music62/v4/4a/6f/
e7/4a6fe7c8-7ca1-c43f-241d-f7e84a014f1b/source/30x30bb.jpg,
releaseDate=2017-09-18T05:00:00Z),
ItunesPodcast(collectionCensoredName=Android Developers Backstage,
feedUrl=http://feeds.feedburner.com/blogspot/AndroidDevelopersBackstage,
artworkUrl30=http://is3.mzstatic.com/image/thumb/Music62/v4/15/
c9/96/15c996fd-4856-79bb-12ba-1d25c67d77d7/source/30x30bb.jpg,
releaseDate=2017-09-11T17:20:00Z),
ItunesPodcast(collectionCensoredName=The Android Cast, feedUrl=http://
www.buzzsprout.com/60878.rss, artworkUrl30=http://is1.mzstatic.com/image/
thumb/Music71/v4/8d/b5/4c/8db54c53-75c0-b214-9606-a228e19f49f9/source/
30x30bb.jpg, releaseDate=2016-10-08T07:00:00Z)])
```

Great, the service is working as designed! The next step is to hide the service behind a repository just like you did with the database in the PlaceBook app. The repository will be the only part of the app that touches the `ItunesService`.

Create a new package named **repository** under the project root. Inside that package, create a new file named **ItunesRepo.kt**, and replace the contents with the following:

```kotlin
// 1
class ItunesRepo(private val itunesService: ItunesService) {
  // 2
  fun searchByTerm(term: String,
      callBack: (List<ItunesPodcast>?) -> Unit) {
    // 3
    val podcastCall = itunesService.searchPodcastByTerm(term)
    // 4
    podcastCall.enqueue(object : Callback<PodcastResponse> {
      // 5
      override fun onFailure(call: Call<PodcastResponse>?,
          t: Throwable?) {
        // 6
        callBack(null)
      }
      // 7
      override fun onResponse(
          call: Call<PodcastResponse>?,
          response: Response<PodcastResponse>?) {
        // 8
        val body = response?.body()
        // 9
        callBack(body?.results)
      }
    })
  }
}
```

1. The primary constructor for `ItunesRepo` is defined to require an existing instance of the `ItunesService` interface. This is an example of the Dependency Injection principle. By passing in the `ItunesService` to `ItunesRepo`, it makes it possible for the calling code to pass in a different implementation for the `ItunesService`. `ItuneRepo` doesn't care about the implementation, as long as it conforms to the `ItunesService` interface.

2. `ItunesRepo` contains a single method named `searchByTerm`. This method takes a search term as the 1st parameter and a method as the 2nd parameter. The method defines a single parameter as a `List` of `iTunesPodcast` objects.

3. The call to `searchPodcastByTerm()` is made, passing in a search term. This returns a Retrofit `Call` object.

4. The `enqueue` method is invoked on the `Call` object and it runs in the background to retrieve the response from the web service. The `enqueue` method takes a Retrofit CallBack interface that defines two callback methods: `onFailure()` and `onResponse()`.

5. `onFailure()` is called if anything goes wrong with the call such as a network error or an invalid URL.

6. The `callBack` method is called with a `null` value if there's an error.

7. `onResponse()` is called if the call succeeds.

8. The populated `PodcastResponse` model is retrieved with a call to `response.body()`.

9. The `callBack` method is called with the `results` object from the `PodcastResponse` model.

This gets rid of the extra objects from the raw `PodcastResponse` object that aren't needed and returns just the resulting `ItunesPodcast` object.

You can replace the temporary code in `PodcastActivity.onCreate()` with the following:

```
val TAG = javaClass.simpleName

val itunesService = ItunesService.getServiceInstance()

val itunesRepo = ItunesRepo(itunesService)

itunesRepo.searchByTerm("Android Developer", {
  Log.i(TAG, "Results = $it")
})
```

This now uses the new `ItunesRepo` to search for the podcast and prints the results to the Logcat window.

`ItunesService.getServiceInstance()` is called to get an instance of the `ItunesService` and it's passed to a new `ItunesRepo` instance. `searchByTerm()` is called with the search term and is passed an anonymous method to receive the results.

Build and run the app. After the default "Hello World" screen is displayed again, check your Logcat window for the following results:

```
I/PodcastActivity: Results =
[ItunesPodcast(collectionCensoredName=Fragmented — Android Developer
Podcast, feedUrl=https://rss.simplecast.com/podcasts/1684/rss,
artworkUrl30=http://is5.mzstatic.com/image/thumb/Music62/v4/4a/6f/
e7/4a6fe7c8—7ca1—c43f—241d—f7e84a014f1b/source/30x30bb.jpg,
releaseDate=2017—09—18T05:00:00Z),
ItunesPodcast(collectionCensoredName=Android Developers Backstage,
feedUrl=http://feeds.feedburner.com/blogspot/AndroidDevelopersBackstage,
artworkUrl30=http://is3.mzstatic.com/image/thumb/Music62/v4/15/
c9/96/15c996fd—4856—79bb—12ba—1d25c67d77d7/source/30x30bb.jpg,
releaseDate=2017—09—11T17:20:00Z),
ItunesPodcast(collectionCensoredName=The Android Cast, feedUrl=http://
www.buzzsprout.com/60878.rss, artworkUrl30=http://is1.mzstatic.com/image/
thumb/Music71/v4/8d/b5/4c/8db54c53—75c0—b214—9606—a228e19f49f9/source/
30x30bb.jpg, releaseDate=2016—10—08T07:00:00Z)]
```

Notice that this time the response contains just the list of `ItunesPodcast` objects. You can remove the test code in PodcastActivity.

Where to go from here?

In the next chapter, you'll start building out the user interface to allow the user to search for podcasts.

Chapter 21: Finding Podcasts

By Tom Blankenship

Now that the groundwork to search iTunes has been laid, you'll build out a user interface that allows the user to search for podcasts.

Your goal is to provide a search box at the top of the screen where the user can enter a search term. The `ItunesRepo` you created in the last chapter will be used to fetch the list of matching podcasts. The results will be shown in a `RecyclerView` and will include the podcast artwork.

Creating a search interface can be as simple as adding a text view, responding to the user entering text, and populating a `RecyclerView` with the results. While this method works fine, the Android SDK provides a built-in search feature that helps future-proof your apps.

Android search

If you are following along with your own app, open it and keep going with it for this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **PodPlay** app under the **starter** folder. The first time you open the project, Android Studio takes a few minutes to set up your environment and update dependencies.

Android's search functionality provides part of the search interface. It can be displayed either as a **search dialog** at the top of an activity, or as a **search widget**, which can be placed within an activity or on the action bar.

The way it works is like this: Android handles the user input and then passes the search query to an activity. This makes it easy to add search capability to any activity within your app, while only using a single dedicated activity to display the results.

Some benefits to using Android search include:

- Displaying suggestions based on previous queries

- Displaying suggestions based on actual search data

- Having the ability to search by voice

- Adding search suggestions to the system-wide Quick Search Box

When running on Android 3.0 or later, Google suggests that you use a search widget instead of a search dialog, which is exactly what you'll do in PodPlay — you'll use the search widget and insert it as an **action view** in the app bar.

An action view is a standard feature of the support library Toolbar that allows for advanced functionality within the app bar. When the search widget is added as an action view, it displays a collapsable search view — located in the app bar — and handles all of the user input.

The following illustrates an active search widget, which gets activated when the user taps the search icon. It includes an `EditText` view with some hint text, along with a back arrow used to close out the search:
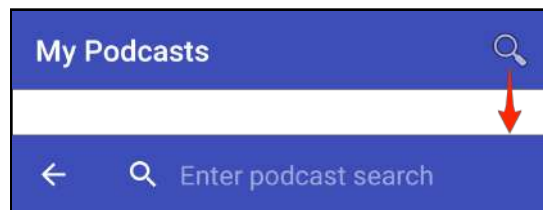
Implementing search

To implement search capabilities, several steps are required. You need to:

1. Create a search configuration XML file.

2. Declare a searchable activity.

3. Add an options menu.

4. Set the searchable configuration in `onCreateOptionsMenu`.

Search configuration file

The first step is to create a search configuration file. This file lets you define some details about the search behavior. It may contain several attributes such as:

- `label`: This should match the name of your application.

- `hint`: A hint thats displays in the search field before any text is entered.

- `inputType`: The type of data expected for the search field.

There are also multiple settings to control search suggestion behavior, voice search behavior, Quick Search box settings and more. The only required attribute is the label. Because you'll be implementing a basic search for PodPlay, you will only define the label and hint attributes.

> **Note:** The Android developer site has extensive documentation on the more advanced search options at https://developer.android.com/guide/topics/search/searchable-config.html

By convention, the search configuration file is named **searchable.xml**, and it must be stored in the **res\xml** folder.

To create this file in the proper location, right-click on the **res** folder and select **New ▸ Android resource file**. Set the values in the dialog as follows:

- File name: searchable

- Resource type: XML

- Root element: searchable

- Source set: main

- Directory name: xml

Click OK and the file — as well as the `xml` resource directory — will be created. Replace the contents of **searchable.xml** with this:

```xml
<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android=
    "http://schemas.android.com/apk/res/android"
            android:label="@string/app_name"
            android:hint="@string/search_hint" >
</searchable>
```

This will display an error for the missing `@string/search_hint` resource. To fix this, open **res\values\strings.xml** and add the following line:

```xml
<string name="search_hint">Enter podcast search</string>
```

Searchable activity

The next step is to designate a searchable Activity. The search widget will start this activity using an Intent that contains the users's search term. It's the activity's responsibility to take the search term, look it up and display the results to the user.

In some cases, you may want to have a separate activity display the search results. However, PodPlay is going to use a single Activity for the entire app, and you'll use fragments to display the different views. This makes adding the searchable activity straightforward — you'll designate `PodcastActivity` as the searchable activity.

The searchable activity is set on the `<activity>` element in your manifest file. There are two things you need to do to set up a searchable activity:

1. Add an intent filter for action **Intent.ACTION_SEARCH**. This is a static property in the **Intent** class, and is defined with the value "android.intent.action.SEARCH". The value is required in the manifest, but you'll use **Intent.ACTION_SEARCH** in code.

2. Specify the searchable configuration file that you defined earlier using a **meta-data** element.

Open **app\manifests\AndroidManifest.xml** and update the `PodcastActivity` element to match this:
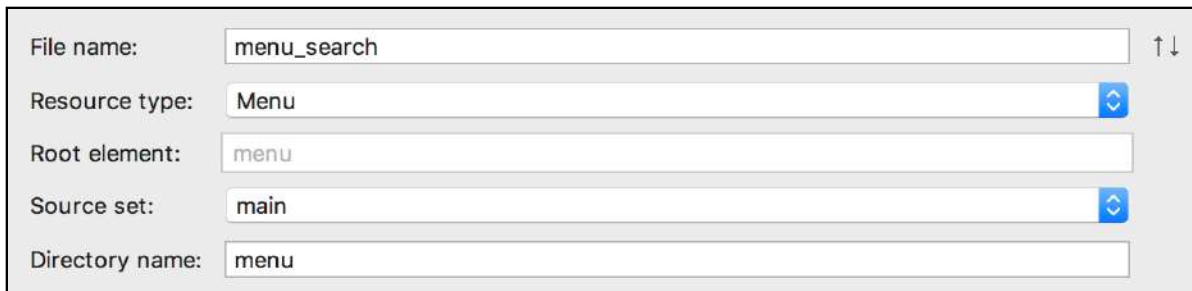
```xml
<activity android:name=".ui.PodcastActivity">
  <intent-filter>
    <action android:name="android.intent.action.MAIN"/>
    <action android:name="android.intent.action.SEARCH"/>
    <category android:name="android.intent.category.LAUNCHER"/>
  </intent-filter>
  <meta-data android:name="android.app.searchable"
      android:resource="@xml/searchable"/>
</activity>
```

Adding the options menu

Since the search widget will be shown as an action view in the app bar, you'll need to define an options menu with a single search button item.

To do this, right-click on the **res** folder, then select **New ▸ Android Resource File**. Set the resource type to **Menu**, which will automatically set the root element type to `menu` and the folder to `menu`. Name your file `menu_search`:

File name:	menu_search	↑↓
Resource type:	Menu	
Root element:	menu	
Source set:	main	
Directory name:	menu	

Click **OK**, then open **res\menu\menu_search.xml** and replace the existing text with the following:

```xml
<?xml version="1.0" encoding="utf-8"?>
<menu 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"
    tools:context=
        "com.raywenderlich.podplay.ui.PodcastActivity">

  <item android:id="@+id/search_item"
        android:title="@string/search"
        android:icon="@android:drawable/ic_menu_search"
        app:showAsAction=
          "collapseActionView|ifRoom"
        app:actionViewClass=
          "android.support.v7.widget.SearchView"/>
</menu>
```

This defines an options menu with a single `menu_search` item that is shown as an action view and uses the built-in `ic_menu_search` icon from the Android operating system.

The `showAsAction` pipe-separated options are set to collapse the action view by default and display in the app bar, if there's room. The `actionViewClass` must be set as `android.support.v7.widget.SearchView` since you want your shiny search bar to be backwards-compatible with older versions of Android.

Notice that you still need to define the value of the **search** resource string, which is indicated by the red text. You've already seen how to do that manually, but Android Studio offers another way to add a missing String resource directly from the code where you've tried to use it.

Place the cursor within the red `@string/search` text and press **Option+Return** on Mac or **Alt+Enter** on Windows to bring up the context menu, and select **Create string value resource 'search'**:
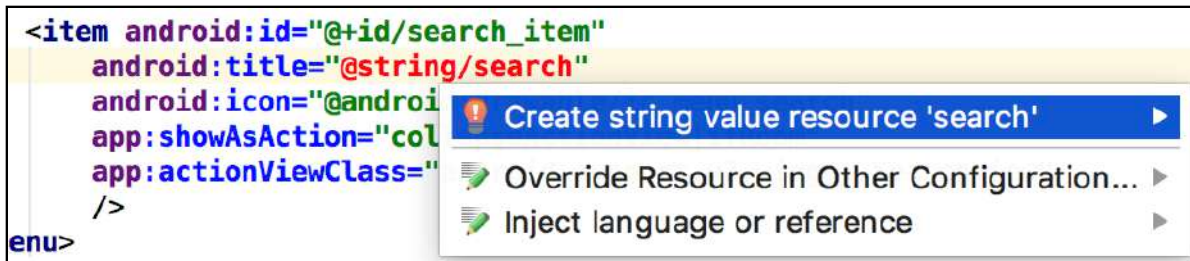
In the dialog that appears, type in `Search` for the **Resource value** field and click **OK**.
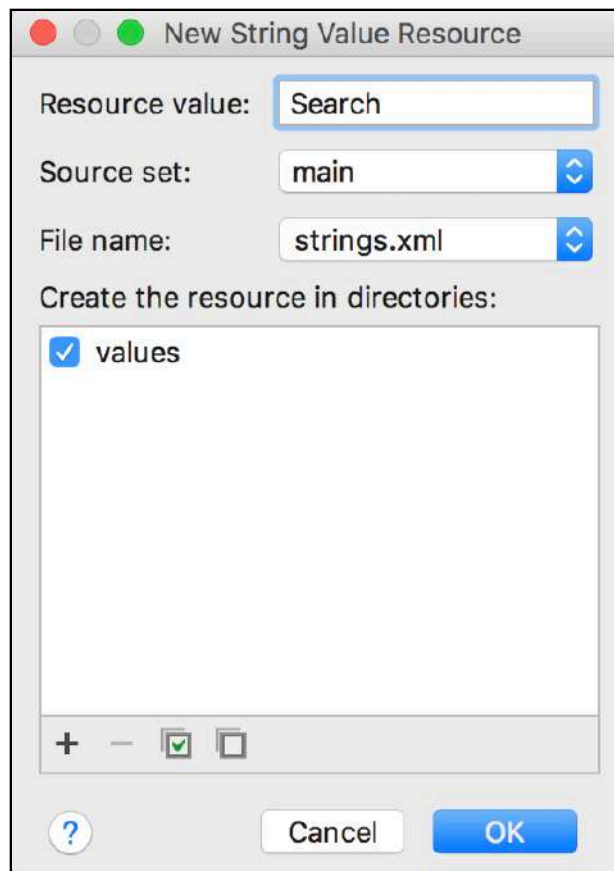
This will add the appropriate line to your **strings.xml** file, and the menu file will update so that all the text is a happy green, indicating that all of your resources exist.

Next, you need to load the options menu and configure it properly.

Loading the options menu

Open **PodcastActivity.kt** and override the `onCreateOptionsMenu` method; note that you do not need to call `super`:

```kotlin
override fun onCreateOptionsMenu(menu: Menu): Boolean {
  // 1
  val inflater = menuInflater
  inflater.inflate(R.menu.menu_search, menu)
  // 2
  val searchMenuItem = menu.findItem(R.id.search_item)
  val searchView = searchMenuItem?.actionView as SearchView
  // 3
  val searchManager = getSystemService(Context.SEARCH_SERVICE)
      as SearchManager
  // 4
  searchView.setSearchableInfo(
      searchManager.getSearchableInfo(componentName))

  return true
}
```

> **Note**: Be sure to import **android.support.v7.widget.SearchView** and NOT the non-support version to resolve the `SearchView` reference.

What's happening in this code?

1. First, the options menu is inflated. If you had only these two lines, you would have a basic search view which activates when the action button is tapped. The rest of the method is what makes it a fully functioning search widget.

2. The search action menu item is found within the options menu, and the search view is taken from the item's `actionView` property.

3. The system `SearchManager` object is loaded. `SearchManager` provides some key functionality when working with search services. It will be used later to load the searchable info xml file you created earlier.

4. `searchManager` is used to load the search configuration and assign it to the `searchView`.

Build and run the app. You'll see a search icon displayed in the app bar:

Tap the search icon and it will expand into the search view. Notice the features built into the search widget:
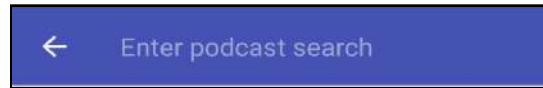
- A back arrow is displayed to cancel the search, hide the keyboard and return to the normal app bar.

- The hint you included in the search configuration is shown in the search view.

- A clear button is added to clear out the search text after at least one character has been entered.

Enter a search phrase and hit return. The search view goes away, and nothing else happens! The search widget is knocking on the activity's door, but no one is answering. It's now up to you to implement the actual search logic.

Implementing the search

By default, the search widget starts the searchable activity that you defined in the manifest, and it sends it an **Intent** with the search query as an extra data item on the intent. In this case, the searchable activity is already running, but you don't want two copies of it on the activity stack.

To get around this undesired behavior, you can set the **android:launchMode** of **PodcastActivity** to **singleTop**.

Open **manifests\AndroidManifest.xml** and update the PodcastActivity's activity element to add this attribute:

```
<activity android:name=".ui.PodcastActivity"
    android:launchMode="singleTop">
```

This tells the system to skip adding another PodcastActivity to the stack if it's already on top. Now instead of creating a new copy of PodcastActivity to receive the search intent, a call is made to **onNewIntent()** on the existing PodcastActivity.

Open **ui\PodcastActivity.kt** and add the following method:

```
private fun performSearch(term: String) {
    val itunesService = ItunesService.getServiceInstance()
    val itunesRepo = ItunesRepo(itunesService)
```

```
    itunesRepo.searchByTerm(term, {
      Log.i(TAG, "Results = $it")
    })
  }
```

This method contains the same code that you had in `onCreate()`, except that the search term is not hard-coded. If the search code is still in `onCreate()`, go ahead and remove it.

Next, add the following method to handle incoming intents:

```
  private fun handleIntent(intent: Intent) {
    if (Intent.ACTION_SEARCH == intent.action) {
      val query = intent.getStringExtra(SearchManager.QUERY)
      performSearch(query)
    }
  }
```

This method takes in an intent and checks to see if it's an `ACTION_SEARCH`. If so, it extracts the search query string and passes it to `performSearch()`.

Finally, override `onNewIntent` so it can receive the updated intent when a new search is performed:

```
  override fun onNewIntent(intent: Intent) {
    super.onNewIntent(intent)
    setIntent(intent)
    handleIntent(intent)
  }
```

This method will be called when the intent is sent from the search widget. It calls `setIntent()` to make sure the new intent is saved with the Activity. `handleIntent()` is called to perform the search.

Build and run the app. Tap the search icon, enter a search term and press return. The raw results of the search will be written to the Logcat window:

Now that you're getting the search results back from iTunes, you're finally ready to display those results to the user!

Displaying search results

You'll display results using a standard `RecyclerView`, with one podcast per row. iTunes includes a cover image for each podcast, which you'll display along with the podcast title and last updated date; this will give the user a quick overview of each podcast.

Start by doing some housekeeping to replace the standard action bar with the `appcompat` version. This is same technique used in the PlaceBook app; to save time, the dependencies are already set up, but there are still a few things which need to be done:

Appcompat app bar

Open the project **build.gradle** file and add the following to the **ext** section:

```
support_lib_version = '26.1.0'
```

Open the module **build.gradle** file and change the `appcompat-v7` dependency to the following:

```
implementation "com.android.support:appcompat-v7:$support_lib_version"
```

Add the following new line to the dependencies:

```
implementation "com.android.support:design:$support_lib_version"
```

Open **res****values****styles.xml** and add the following:

```
<style name="AppTheme.NoActionBar">
  <item name="windowActionBar">false</item>
  <item name="windowNoTitle">true</item>
</style>

<style name="AppTheme.AppBarOverlay"
    parent="ThemeOverlay.AppCompat.Dark.ActionBar"/>

<style name="AppTheme.PopupOverlay"
    parent="ThemeOverlay.AppCompat.Light"/>
```

Open **AndroidManifest.xml** and add the following attribute to the `PodcastActivity` activity element:

```
android:theme="@style/AppTheme.NoActionBar"
```

Open **res\layout\activity_podcast.xml** and replace the `<TextView>` with the following:

```
<android.support.design.widget.AppBarLayout
    android:id="@+id/app_bar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:fitsSystemWindows="true"
    android:theme="@style/AppTheme.AppBarOverlay">

  <android.support.v7.widget.Toolbar
      android:id="@+id/toolbar"
      android:layout_width="match_parent"
      android:layout_height="?attr/actionBarSize"
      app:popupTheme="@style/AppTheme.PopupOverlay"/>

</android.support.design.widget.AppBarLayout>
```

Open **PodcastActivity.kt** and make sure the activity can see the variables created in the layout by importing it with `KotlinExtensions`:

```
import kotlinx.android.synthetic.main.activity_podcast.*
```

Next, add the following method:

```
private fun setupToolbar() {
   setSupportActionBar(toolbar)
}
```

This is the same technique used in "Chapter 17: Detail Activity" to get ActionBar support for the activity. `setSupportActionBar()` is a built-in method that makes the Toolbar act as the `ActionBar` for this Activity.

Finally, call that method from the end of `onCreate()`:

```
setupToolbar()
```

SearchViewModel

To display the results in the activity, you'll need a view model first. Remember from previous architecture discussions that views using Architecture Components only get data from view models. You'll create a **SearchViewModel** and the `PodcastActivity` will use it to display the results.

`SearchViewModel` will inherit from `AndroidViewModel`, which is part of the lifecycle component of the Android architecture components.

Open the project **build.gradle** file and add the following to the **ext** section:

```
architecture_version = '1.1.0'
```

Open the module **build.gradle** file and add the following to the dependencies section:

```
implementation "android.arch.lifecycle:extensions:$architecture_version"
```

Right click on the **com.raywenderlich.podplay** folder, and create a new package named **viewmodel** to help keep your view models organized. Add new empty Kotlin file within the **viewmodel** package named **SearchViewModel.kt**.

Open it up, and set up the initial search view model class:

```
class SearchViewModel(application: Application) :
    AndroidViewModel(application) {
}
```

The `application` parameter is required by the `AndroidViewModel` superclass. In fact, you can't add additional parameters to this class's constructor because of how it is provided through the Architecture components, so you must set up any additional properties separately.

In this case, add a property for an `ItunesRepo` which will fetch the information:

```
var iTunesRepo: ItunesRepo? = null
```

This is optional and initialized to `null` since it's expected that the caller — in this case, the `PodcastActivity` — will pass this object in before calling any method to fetch the data.

Next, define a data class within your view model that has only the data which is actually necessary for the view, and that has default empty string values:

```
data class PodcastSummaryViewData(
  var name: String? = "",
  var lastUpdated: String? = "",
  var imageUrl: String? = "",
  var feedUrl: String? = "")
```

Next, add a helper method to convert from the raw model data to the view data:

```
private fun itunesPodcastToPodcastSummaryView(
   itunesPodcast: PodcastResponse.ItunesPodcast):
   PodcastSummaryViewData {
   return PodcastSummaryViewData(
     itunesPodcast.collectionCensoredName,
     itunesPodcast.releaseDate,
     itunesPodcast.artworkUrl30,
     itunesPodcast.feedUrl)
}
```

Finally, define a method to perform the search, which will eventually be called by
`PodcastActivity`:

```
// 1
fun searchPodcasts(term: String,
  callback: (List<PodcastSummaryViewData>) -> Unit) {
  // 2
  iTunesRepo?.searchByTerm(term, { results ->
    if (results == null) {
      // 3
      callback(emptyList())
    } else {
      // 4
      val searchViews = results.map { podcast ->
        itunesPodcastToPodcastSummaryView(podcast)
      }
      // 5
      searchViews.let { it -> callback(it) }
    }
  })
}
```

Let's go through the code of this method step-by-step:

1. The first parameter is the search term. The `callback` parameter is a method that's
    called with the results. Since the iTunes repo's search method runs asynchronously,
    this method needs a way to let its caller know when the work is done.

2. iTunesRepo is used to perform the search asynchronously.

3. If the results are `null`, then an empty list is passed to the `callback` method.

4. If the results are not `null`, then they're mapped to **PodcastSummaryViewData**
    objects. This follows the principle of providing the view with just enough data for
    presentation.

5. If the mapped results are valid (in this case, non-null) they are passed to the
    `callback` method so they can be displayed.

Next, you'll add the RecyclerView to display the search results.

Results RecyclerView

First, define the layout for a single search result item. Create a new file in the
res\layout folder named **search_item.xml** and set the contents to the following:

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:orientation="horizontal"
```

```xml
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:paddingTop="10dp"
        android:paddingBottom="10dp"
        android:paddingLeft="5dp"
        android:paddingRight="5dp">
    <ImageView
        android:id="@+id/podcastImage"
        android:layout_width="40dp"
        android:layout_height="40dp"
        android:layout_marginEnd="5dp"
        android:adjustViewBounds="true"
        android:scaleType="fitStart"/>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginRight="5dp"
        android:orientation="vertical">

      <TextView
          android:id="@+id/podcastNameTextView"
          android:layout_width="match_parent"
          android:layout_height="wrap_content"
          android:layout_gravity="top"
          android:layout_marginBottom="5dp"
          android:textStyle="bold"
          tools:text="Name"/>

      <TextView
          android:id="@+id/podcastLastUpdatedTextView"
          android:layout_width="match_parent"
          android:layout_height="wrap_content"
          android:layout_gravity="top"
          android:textSize="12sp"
          tools:text="Last updated"/>
    </LinearLayout>
  </LinearLayout>
```

This layout defines an image on the left, and a podcast name and last updated date on the right:
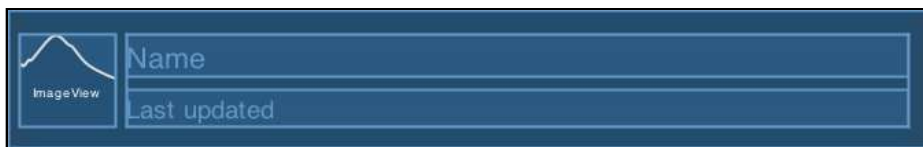
Next, open **xml/layout/activity_podcast.xml** and add the following below the closing tag of the `AppBarLayout`:

```xml
<android.support.v7.widget.RecyclerView
    android:id="@+id/podcastRecyclerView"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_marginEnd="0dp"
```

```
        android:layout_marginStart="0dp"
        android:scrollbars="vertical"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@id/app_bar"/>

    <ProgressBar
        android:id="@+id/progressBar"
        android:layout_width="40dp"
        android:layout_height="40dp"
        android:layout_gravity="center"
        android:visibility="gone"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:visibility="visible"/>
```

This defines a `RecyclerView` to hold the search results and a `ProgressBar` to display while the search is being performed.

Glide image loader

Before defining the adapter for the `RecyclerView`, you need to consider the best way to display the cover art efficiently. The user may do many searches in a row, and each one will return up to 50 results.

If you decided to pre-fetch the image for each one and store it locally or in memory, it wouldn't make for an enjoyable user experience; there could potentially be a large delay before any results would show up.

You may try to get a little smarter about it and only load the images as they're needed by the `RecyclerView` adapter, but this will result in clunky scrolling performance.

Your next step to image loading nirvana might be to load the images on-demand in the background, so the scrolling remains smooth. At about this point in the development process you're probably thinking, "This sounds like a lot of work. There has to be a better way!" and fortunately there is!

There are several third party libraries made to handle this exact situation. They perform on-demand loading in the background and do intelligent caching to keep the most recently loaded images ready for quick retrieval. One popular library recommended by Google is named **Glide**.

Glide was developed to make image scrolling as smooth as possible, but it can be used in any situation where you need to load images from a remote source.

Using Glide is as simple as making a single chain of calls that specifies a context, the remote image url and a view to place the image. Glide handles all of the details, including background loading and cancelling the image load when the parent view goes away.

To use Glide, add the following to the dependencies section in the module **build.gradle** file:

```
implementation "com.github.bumptech.glide:glide:4.2.0"
```

Create a new package under **com.raywenderlich.podplay** named **adapter**. Add a new Kotlin file to this package named **PodcastListAdapter.kt** and update it with the following contents:

```kotlin
class PodcastListAdapter(
    private var podcastSummaryViewList:
        List<PodcastSummaryViewData>?,
    private val podcastListAdapterListener:
        PodcastListAdapterListener,
    private val parentActivity: Activity) :
    RecyclerView.Adapter<PodcastListAdapter.ViewHolder>() {

  interface PodcastListAdapterListener {
    fun onShowDetails(podcastSummaryViewData:
        PodcastSummaryViewData)
  }

  inner class ViewHolder(v: View,
      private val podcastListAdapterListener:
          PodcastListAdapterListener) :
      RecyclerView.ViewHolder(v) {

    var podcastSummaryViewData: PodcastSummaryViewData? = null
    val nameTextView: TextView =
        v.findViewById(R.id.podcastNameTextView)
    val lastUpdatedTextView: TextView =
        v.findViewById(R.id.podcastLastUpdatedTextView)
    val podcastImageView: ImageView =
        v.findViewById(R.id.podcastImage)

    init {
      v.setOnClickListener {
        podcastSummaryViewData?.let {
          podcastListAdapterListener.onShowDetails(it)
        }
      }
    }
  }

  fun setSearchData(podcastSummaryViewData:
      List<PodcastSummaryViewData>) {
    podcastSummaryViewList = podcastSummaryViewData
    this.notifyDataSetChanged()
  }
```

```
    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int):
        PodcastListAdapter.ViewHolder {
      return ViewHolder(LayoutInflater.from(parent.context)
          .inflate(R.layout.search_item, parent, false),
          podcastListAdapterListener)
    }

    override fun onBindViewHolder(holder: ViewHolder,
        position: Int) {
      val searchViewList = podcastSummaryViewList ?: return
      val searchView = searchViewList[position]
      holder.podcastSummaryViewData = searchView
      holder.nameTextView.text = searchView.name
      holder.lastUpdatedTextView.text = searchView.lastUpdated
        //TODO: Use Glide to load image
    }

    override fun getItemCount(): Int {
      return podcastSummaryViewList?.size ?: 0
    }
  }
```

Most of this has already been covered in earlier chapters on `RecyclerViews`, so we won't
cover it again in detail here.

Now, replace the TODO in `onBindViewHolder` with the following:

```
  Glide.with(parentActivity)
    .load(searchView.imageUrl)
    .into(holder.podcastImageView)
```

This uses Glide's fluent API to efficiently load the podcast image into the image view.
The `with()` call can take an Activity, Fragment, View or Context. By providing Glide
with the `parentActivity` that was passed in with the constructor, it will be tied to the
Activity Lifecycle and properly clean up image usage.

The `load()` call specifies the remote URL of the image to be loaded. The `into()` call
specifies the `ImageView` to place the image into once it's loaded.

Glide also allows you to load images directly into Bitmap images instead of into a
specified `ImageView`. There are several other calls that can be added to the fluent API to
control options and do image manipulation such as transformations and animated
transitions.

You now have everything in place to display the data from the view model — it's time to
hook up the view model data to the RecyclerView.

Populating the RecyclerView

Open **PodcastActivity.kt** and add the following lines to the top of the class:

```
private lateinit var searchViewModel: SearchViewModel
private lateinit var podcastListAdapter: PodcastListAdapter
```

Add the following method to set up view models — for now, just the `SearchViewModel`:

```
private fun setupViewModels() {
  val service = ItunesService.instance
  searchViewModel = ViewModelProviders.of(this).get(
      SearchViewModel::class.java)
  searchViewModel.iTunesRepo = ItunesRepo(service)
}
```

This creates an instance of the `ItunesService`, and then uses the `ViewModelProviders` class to get an instance of the `SearchViewModel`. It then creates a new `ItunesRepo` object with the `ItunesService` and assigns it to the `SearchViewModel`.

Next, add the following method to set up the `RecyclerView` with a `PodcastListAdapter`:

```
private fun updateControls() {
  podcastRecyclerView.setHasFixedSize(true)

  val layoutManager = LinearLayoutManager(this)
  podcastRecyclerView.layoutManager = layoutManager

  val dividerItemDecoration =
      android.support.v7.widget.DividerItemDecoration(
      podcastRecyclerView.context, layoutManager.orientation)
  podcastRecyclerView.addItemDecoration(dividerItemDecoration)

  podcastListAdapter = PodcastListAdapter(null, this, this)
  podcastRecyclerView.adapter = podcastListAdapter
}
```

Add the following lines calling the setup methods you just made to the end of `onCreate()`:

```
setupViewModels()
updateControls()
```

Next, update the `PodcastActivity` class declaration to adopt the `PodcastListAdapterListener` interface for `PodcastActivity`:

```
class PodcastActivity : AppCompatActivity(),
    PodcastListAdapterListener {
```

This is required by the `PodcastListAdapter` created in `updateControls()`.

Now, add the following method to satisfy the `PodcastListAdapterListener` interface:

```
override fun onShowDetails(
    podcastSummaryViewData: PodcastSummaryViewData) {
  // Not implemented yet
}
```

This will be called when the user taps on a podcast in the `RecyclerView`. You'll complete the implementation in the next chapter.

Next, add the following helper methods to encapsulate showing and hiding the progress bar during searching:

```
private fun showProgressBar() {
  progressBar.visibility = View.VISIBLE
}

private fun hideProgressBar() {
  progressBar.visibility = View.INVISIBLE
}
```

The last thing you'll need to do in **PodcastActivity.kt** is update the `performSearch` method to use the view model you set up:

```
private fun performSearch(term: String) {
  showProgressBar()
  searchViewModel.searchPodcasts(term, { results ->
    hideProgressBar()
    toolbar.title = getString(R.string.search_results)
    podcastListAdapter.setSearchData(results)
  })
}
```

Finally, add the following line to your **res/values/strings.xml** file to satisfy the `@string/search_results` reference.

```
<string name="search_results">Search Results</string>
```

This uses `SearchViewModel` to find the podcasts based on the search `term`. It displays the progress bar before the search starts and hides it as soon as it's over. The `toolbar` title is updated and the `RecyclerView` adapter is updated with the `results`.

Build and run the app. Tap the search icon and enter a search term. The results are displayed and you'll see the cover art images load in after the main content is displayed. If your search returns enough results, scroll through the list as quickly as possible and the movement should remain smooth no matter how many results and images are loading.
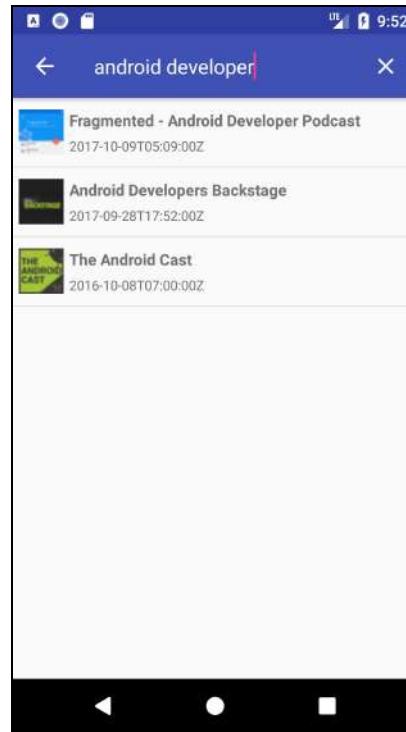
That doesn't look too bad, but the last updated date is formatted more for computers than for humans. Time to fix that!

Date formatting

Create a new package under **com.raywenderlich.podplay** named **util**. To it, add a new Kotlin file named **DateUtils.kt** with the following contents:

```kotlin
object DateUtils {
  fun jsonDateToShortDate(jsonDate: String?): String {
    //1
    if (jsonDate == null) {
      return "–"
    }

    // 2
    val inFormat = SimpleDateFormat("yyyy–MM–dd'T'HH:mm:ss")
    // 3
    val date = inFormat.parse(jsonDate)
    // 4
    val outputFormat = DateFormat.getDateInstance(DateFormat.SHORT,
        Locale.getDefault())
    // 6
    return outputFormat.format(date)
  }
}
```

> **Note**: Make sure you import **java.text.DateFormat** and
> **java.text.SimpleDateFormat** rather than their **android** counterparts!

This defines a method named `jsonDateToShortDate` that converts the date returned from iTunes into a simple month, date and year format using the user's current locale.

1. First, check that the `jsonDate` string coming in is not `null`. If it is, simply return a string, which doesn't need to be translated (to avoid calling in to Android Resources), indicating that no date was provided.

2. A `SimpleDateFormat` is defined to match the date format returned by iTunes.

3. The `jsonDate` string is parsed and placed into a `Date` object, named `date`.

4. The output format is defined as a short date to match the currently defined locale. By passing in the `Locale.getDefault()`, Android will honor the locale and date settings set by the user.

5. The `date` is formatted and returned.

Open **SearchViewModel.kt**, and in `itunesPodcastToPodcastSummaryView()`, replace the `itunesPodcast.releaseDate` line with the following:

```
DateUtils.jsonDateToShortDate(itunesPodcast.releaseDate),
```

You're calling `jsonDateToShortDate()` to convert the date before it's returned from the SearchViewModel — that way the View never has to know that the date has been formatted, but it will still look much nicer to the user.

Build and run the app. Search for podcasts again and notice the date is now shown in a shorter format and based on the device language settings.
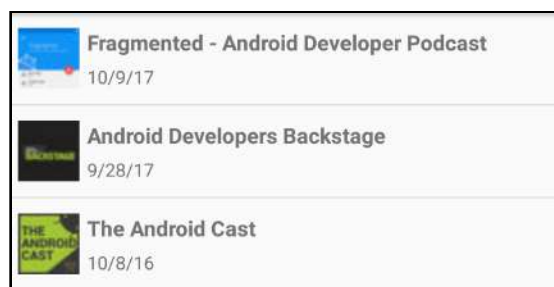
For instance, if you're in the US, the date will be formatted similar to the screenshots above, because `en-US` is most likely your default locale. If you're in a country that uses Day/Month/Year formatting, such as the UK, then the date will be formatted as 28/9/17 instead of 9/28/17.

Want to double check? Go to to Android's Settings app and drill down to **System** ‣ **Languages & Input** ‣ **Languages** and add a language that uses a different date format — for example, if you're from the US, add UK English, or if you're from the UK, add US English. Drag the language you just added to the top of the list:
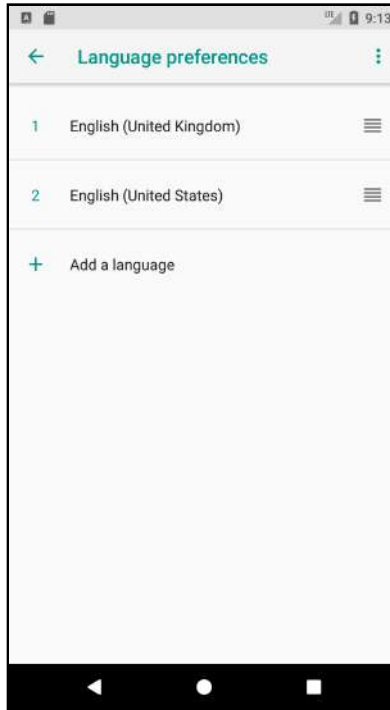
Now return to the app and…

What happened to the search results?! Turns out that when you changed the language settings, Android triggered a configuration change and restarted the `PodcastActivity`.

This is where saving the search intent in `newIntent()` method pays off. You can grab the saved intent when the activity is restarted and then redo the search.

Open **PodcastActivity.kt** and add the following line to the end of `onCreate()`:

```
handleIntent(intent)
```

This gets the saved intent and passes it to the existing `handleIntent()` method.

Build and run the app. Search for some podcasts, and then change language settings again by dragging your primary language back up to the top. This time, the changes are reflected immediately when you re-enter the application:
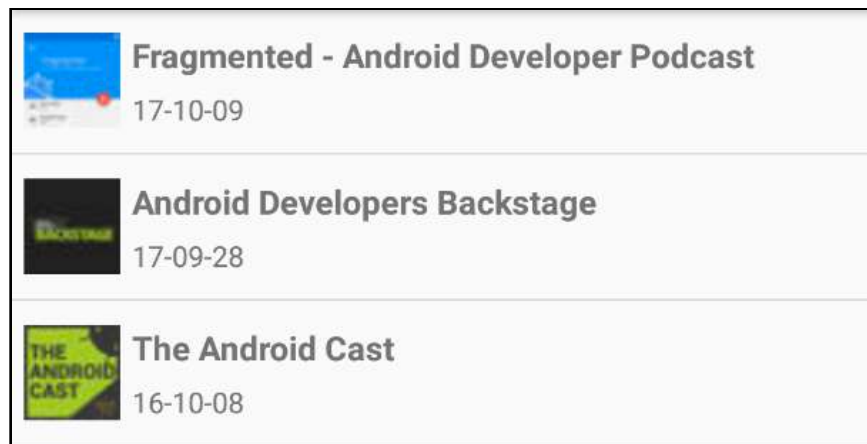
Any configuration change — including rotating the screen — will now be handled correctly.

Where to go from here?

In the next chapter, you'll build out a detailed display for a single podcast and all of its episodes. You'll also build out a data layer for subscribing to podcasts.

Chapter 22: Podcast Details

By Tom Blankenship

Now that the user can find their favorite podcasts, you're ready to add a podcast detail screen. In this chapter, you'll complete the following:

1. Design and build the podcast detail fragment.

2. Expand on the app architecture.

3. Add a podcast detail fragment.

Getting started

If you are following along with your own app, open it and keep going with it for this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **PodPlay** app under the **starter** folder. The first time you open the project, Android Studio takes a few minutes to set up your environment and update dependencies.

You'll start by designing a layout for the podcast detail screen. The purpose of the detail screen is to give the user a quick overview of the podcast, including the title, description, album art and a list of recent episodes. It will also provide a Subscribe action.

The layout will contain the album art and title at the top, a scrollable description below that and a list of episodes below the description. Each episode will contain the title, description, published date and length. The final layout will look like this:
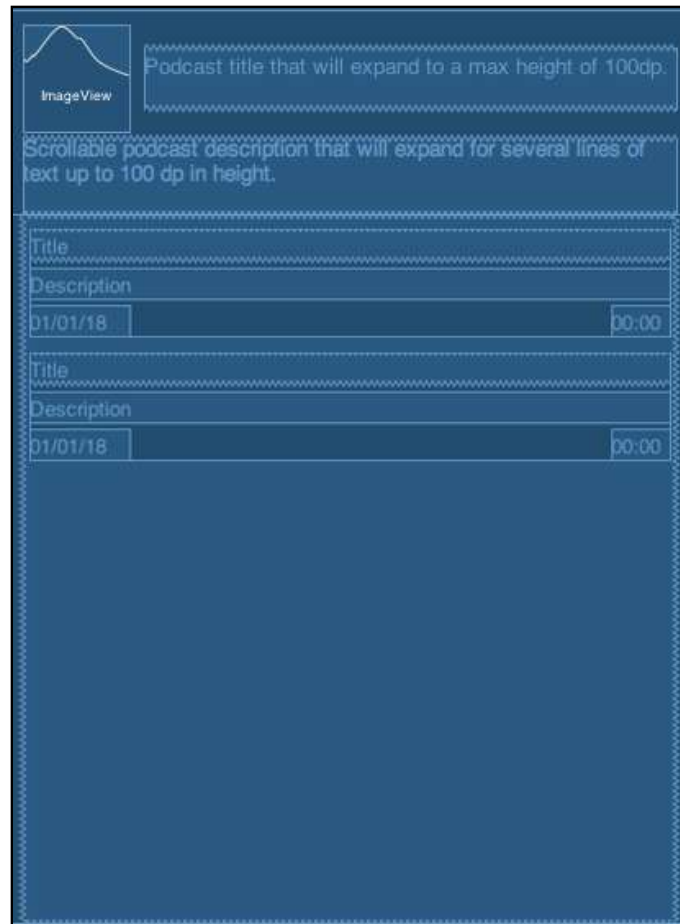
Rather than defining a new activity for the podcast detail, you'll use a fragment to swap out the main podcast listing view with the podcast detail view. The advantage of using fragments will become more clear as you build out the full user interface in later chapters.

Defining the layouts

Create a new layout named **fragment_podcast_details.xml** and replace the contents with the following:

```
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
```

```
            android:layout_height="match_parent">

    <android.support.constraint.ConstraintLayout
        android:id="@+id/headerView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="#eeeeee"
        android:maxHeight="300dp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent">

        <ImageView
            android:id="@+id/feedImageView"
            android:layout_width="60dp"
            android:layout_height="60dp"
            android:layout_marginStart="8dp"
            android:layout_marginTop="8dp"
            android:src="@android:drawable/ic_menu_report_image"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent"/>

        <TextView
            android:id="@+id/feedTitleTextView"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_marginEnd="8dp"
            android:layout_marginStart="8dp"
            android:maxHeight="100dp"
            android:text=""
            android:textSize="14sp"
            android:textStyle="bold"
            app:layout_constraintBottom_toBottomOf="@+id/feedImageView"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toEndOf="@+id/feedImageView"
            app:layout_constraintTop_toTopOf="@+id/feedImageView"/>

        <TextView
            android:id="@+id/feedDescTextView"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_marginEnd="8dp"
            android:layout_marginStart="8dp"
            android:layout_marginTop="4dp"
            android:maxHeight="100dp"
            android:paddingBottom="8dp"
            android:scrollbars="vertical"
            android:text=""
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/feedImageView"/>

    </android.support.constraint.ConstraintLayout>

    <android.support.v7.widget.RecyclerView
        android:id="@+id/episodeRecyclerView"
        android:layout_width="0dp"
        android:layout_height="0dp"
```

```
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/headerView"
        />

</android.support.constraint.ConstraintLayout>
```

This defines the main layout for the detail fragment.

Create a new layout named **episode_item.xml**, which will layout each episode in the RecyclerView in the podcast detail view, and replace the contents with the following:

```
<?xml version="1.0" encoding="utf-8"?>

<android.support.constraint.ConstraintLayout
    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:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    android:layout_marginTop="8dp"
    >

  <TextView
      android:id="@+id/titleView"
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_gravity="top"
      android:layout_marginEnd="0dp"
      android:textStyle="bold"
      app:layout_constraintEnd_toEndOf="parent"
      app:layout_constraintHorizontal_chainStyle="spread"
      app:layout_constraintStart_toStartOf="parent"
      app:layout_constraintTop_toTopOf="parent"
      tools:text="Title"/>

  <TextView
      android:id="@+id/releaseDateView"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="top"
      android:layout_marginTop="4dp"
      app:layout_constraintStart_toStartOf="parent"
      app:layout_constraintTop_toBottomOf="@+id/descView"
      tools:text="01/01/18"/>

  <TextView
      android:id="@+id/durationView"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="top"
      android:layout_marginTop="4dp"
```

```
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/descView"
        tools:text="00:00"/>

    <TextView
        android:id="@+id/descView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_gravity="top"
        android:layout_marginTop="4dp"
        android:maxLines="3"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/titleView"
        tools:text="Description"/>

</android.support.constraint.ConstraintLayout>
```

This defines the layout for a single episode detail item.

Open the **activity_podcast.xml** layout file and add the following before the
`RecyclerView` widget:

```
<FrameLayout
    android:id="@+id/podcastDetailsContainer"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/app_bar"/>
```

This is the container that will hold the podcast detail fragment. It's configured to cover
the entire activity view below the app bar. Nothing is displayed in the container until
you load the podcast detail fragment, after the user taps on a podcast row.

Basic architecture

As in previous chapters, you'll define the basic architecture components consisting of a
repository, a service and a view model in order to display the podcast detail. There's no
need for any database layer at this point.

Let's start with a basic implementation to get the navigation working.

Podcast models

Two models are required to store the podcast data. One defines the detail for a single
podcast episode, and the other is the podcast detail containing a list of episode models.

Create a new package under **com.raywenderlich.podplay** named **model**.

In the **model** package, create a new file named **Episode.kt** and replace the contents with the following:

```kotlin
data class Episode (
    var guid: String = "",
    var title: String = "",
    var description: String = "",
    var mediaUrl: String = "",
    var mimeType: String = "",
    var releaseDate: Date = Date(),
    var duration: String = ""
)
```

This defines the data for a single podcast episode. These properties are required for display, management or playback of an episode.

- `guid`: Unique identifier provided in the RSS feed for an episode.

- `title`: The name of the episode.

- `description`: A description of the episode.

- `mediaUrl`: The location of the episode media. This will be either an audio or video file.

- `mimeType`: Determines the type of file located at the `mediaUrl`.

- `releaseDate`: Date the episode was released.

- `duration`: Duration of the episode as provided in the RSS feed.

Again in the **model** package, create another new file named **Podcast.kt** and replace the contents with the following:

```kotlin
data class Podcast(
    var feedUrl: String = "",
    var feedTitle: String = "",
    var feedDesc: String = "",
    var imageUrl: String = "",
    var lastUpdated: Date = Date(),
    var episodes: List<Episode> = listOf()
)
```

This defines the data for a single podcast.

- `feedUrl`: Location of the RSS feed.

- `feedTitle`: Title of the podcast.

- `feedDesc`: Description of the podcast.

- `imageUrl`: Location of the podcast album art.

- `lastUpdated`: Date the podcast was last updated.

- `episodes`: List of episodes for the podcast.

Podcast repository

A repo will be responsible for retrieving the podcast details and returning it to the view model.

In the **repository** package, create a new file named **PodcastRepo.kt** and replace the contents with the following:

```kotlin
class PodcastRepo {
  fun getPodcast(feedUrl: String,
      callback: (Podcast?) -> Unit) {
    callback(
        Podcast(feedUrl, "No Name", "No description", "No image")
    )
  }
}
```

`PodcastRepo` defines a single method, `getPodcast()`. This method has parameters for a feed URL and a `callback` method. You'll eventually add code to retrieve the feed from the URL and parse it into a `Podcast` object. For now, a simple version of the `Podcast` object is created and passed to the callback method.

Podcast view model

In the **viewmodel** package, create a new file named **PodcastViewModel.kt** and replace the contents with the following:

```kotlin
class PodcastViewModel(application: Application) :
    AndroidViewModel (application) {

  var podcastRepo: PodcastRepo? = null
  var activePodcastViewData: PodcastViewData? = null

  data class PodcastViewData(
      var subscribed: Boolean = false,
      var feedTitle: String? = "",
      var feedUrl: String? = "",
      var feedDesc: String? = "",
      var imageUrl: String? = "",
      var episodes: List<EpisodeViewData>
  )

  data class EpisodeViewData (
      var guid: String? = "",
      var title: String? = "",
```

```
        var description: String? = "",
        var mediaUrl: String? = "",
        var releaseDate: Date? = null,
        var duration: String? = ""
    )
}
```

This defines the `PodcastViewModel` for the detail fragment. The property `podcastRepo` will be set by the caller. The property `activePodcastViewData` will hold the most recently loaded podcast view data. `PodcastViewData` contains everything needed to display the details of a podcast.

The repo returns a list of `Episode` models, so you'll need a method to convert these models into `EpisodeViewData` view models.

Add the following method to the class:

```
private fun episodesToEpisodesView(episodes: List<Episode>):
    List<EpisodeViewData> {
  return episodes.map {
    EpisodeViewData(it.guid, it.title, it.description,
        it.mediaUrl, it.releaseDate, it.duration)
  }
}
```

This method uses `map` to do the following:

• Iterate over a list of `Episode` models.

• Convert `Episode` models to `EpisodeViewData` objects.

• Collect everything back into a list.

You'll also need a method to convert the `Podcast` models from the repo into `PodcastViewData` view objects.

Add the following method:

```
private fun podcastToPodcastView(podcast: Podcast):
    PodcastViewData {
  return PodcastViewData(
      false,
      podcast.feedTitle,
      podcast.feedUrl,
      podcast.feedDesc,
      podcast.imageUrl,
      episodesToEpisodesView(podcast.episodes)
  )
}
```

This method converts a `Podcast` model to a `PodcastViewData` object.

All that's left to do is implement a method to retrieve a podcast from the repo.

Add the following method:

```
// 1
fun getPodcast(podcastSummaryViewData: PodcastSummaryViewData,
    callback: (PodcastViewData?) -> Unit) {
  // 2
  val repo = podcastRepo ?: return
  val feedUrl = podcastSummaryViewData.feedUrl ?: return
  // 3
  repo.getPodcast(feedUrl, {
    // 4
    it?.let {
      // 5
      it.feedTitle = podcastSummaryViewData.name ?: ""
      // 6
      it.imageUrl = podcastSummaryViewData.imageUrl ?: ""
      // 7
      activePodcastViewData = podcastToPodcastView(it)
      // 8
      callback(activePodcastViewData)
    }
  })
}
```

Let's take a closer look at what's happening:

1. `getPodcast()` takes a `PodcastSummaryViewData` object and a callback method.

2. Local variables are assigned to `podcastRepo` and `podcastSummaryViewData.feedUrl`. If either one is `null`, the method returns early.

3. `getPodcast()` from the podcast repo is called with the feed URL.

4. The podcast detail object is checked to make sure it's not `null`.

5. The podcast title is set to the podcast summary name. This line is required because you haven't built out the full implementation of `repo.getPodcast()`. In future chapters, `repo.getPodcast()` will fill in this item, and this line will be removed.

6. The podcast detail image is set to match the podcast summary image URL if it's not `null`.

7. The Podcast object is converted to a `PodcastViewData` object and assigned to the `activePodcastViewData` property.

8. The `callback` method is called and passed the podcast view data.

Details fragment

The detail fragment will be responsible for displaying the podcast details. It will get its data from the `PodcastViewModel`. This is also where the user will be able to Subscribe to a podcast. First, you'll add an action menu with a single Subscribe item.

Open the **strings.xml** resource file and add the following line:

```xml
<string name="subscribe">Subscribe</string>
```

Create a menu resource file named **menu_details.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"
      xmlns:app="http://schemas.android.com/apk/res-auto">
  <item
      android:id="@+id/menu_feed_action"
      android:title="@string/subscribe"
      app:showAsAction="ifRoom"
      />
</menu>
```

This defines the content of a menu that will display when the details fragment is active. It contains a single item with the label "Subscribe".

In the **ui** package, create a new file named **PodcastDetailsFragment.kt** and replace the contents with the following:

```kotlin
class PodcastDetailsFragment : Fragment() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // 1
    setHasOptionsMenu(true)
  }

  override fun onCreateView(inflater: LayoutInflater?,
      container: ViewGroup?, savedInstanceState: Bundle?):
      View? {

    return inflater!!.inflate(R.layout.fragment_podcast_details,
        container, false)
  }

  override fun onActivityCreated(savedInstanceState: Bundle?) {
    super.onActivityCreated(savedInstanceState)
  }

  // 2
  override fun onCreateOptionsMenu(menu: Menu?,
      inflater: MenuInflater?) {
```

```
    super.onCreateOptionsMenu(menu, inflater)
    inflater?.inflate(R.menu.menu_details, menu)
  }
}
```

> **Note**: Be sure to use `import android.support.v4.app.Fragment` when resolving the `Fragment` class.

This is standard procedure for setting up a fragment, except for a couple of important details:

1. The call to `setHasOptionsMenu()` tells Android that this fragment wants to add items to the options menu. This will cause the fragment to receive a call to `onCreateOptionsMenu()`.

2. `onCreateOptionsMenu()` inflates the `menu_details` options menu so its items will be added to the podcast activity menu.

Next, you'll give the fragment access to the main podcast view model.

Add the following property to the class:

```
private lateinit var podcastViewModel: PodcastViewModel
```

Add the following method:

```
private fun setupViewModel() {
  podcastViewModel = ViewModelProviders.of(activity)
      .get(PodcastViewModel::class.java)
}
```

This retrieves an instance of the `PodcastViewModel` from `ViewModelProviders`.

In previous chapters, you used different techniques to communicate between activities and fragments. By using `ViewModelProviders` to manage your view models, you can use a shared view model data as the communication mechanism between a fragment and its host activity.

By passing in the `activity` to `ViewModelProviders.of()`, you'll get the same instance of the `PodcastViewModel` that was created in the `PodcastActivity`. Because the instance was already created in the Activity and assigned the podcast repo, all you need to do is request the already existing instance.

> **Note**: The usage here illustrates a key benefit of using view models: View models

> can be seamlessly shared with any fragments managed by the activity. In addition, view models survive configuration changes, so they don't need to be created again if the screen rotates.

Add the following line to the end of `onCreate()`:

```
setupViewModel()
```

This calls `setupViewModel()` when the fragment is created.

That handles all of the initial setup. Now, it's time to fill out the user interface controls.

Still inside **PodcastDetailsFragment.kt**, add the following method:

```
private fun updateControls() {
  val viewData = podcastViewModel.activePodcastViewData ?:
      return
  feedTitleTextView.text = viewData.feedTitle
  feedDescTextView.text = viewData.feedDesc
  Glide.with(activity).load(viewData.imageUrl)
      .into(feedImageView)
}
```

> **Note**: If Android Studio complains about not being able to resolve `feedTitleTextView` and `feedDescTextView`, add `import kotlinx.android.synthetic.main.fragment_podcast_details.*` to the top of the file.

This first line checks to make sure there is view data available (that we have something in `activePodcastViewData` which we defined earlier to hold the most recently loaded podcast view data). It then uses the view data to populate the title and description `TextView` elements, as well as load the podcast image using Glide.

Add the following to the end of `onActivityCreated()`:

```
updateControls()
```

This calls `updateControls()` after the activity is created. By placing this call here, you ensure that the podcast view data has already been loaded by the main activity.

The last thing you need is a method that the activity can use to create an instance of the fragment.

Add the following method:

```
companion object {
  fun newInstance(): PodcastDetailsFragment {
    return PodcastDetailsFragment()
  }
}
```

This is a convenience method that returns an instance of the `PodcastDetailsFragment`. This may seem unnecessary, but by allowing the fragment to control its own creation, you're giving your code more future flexibility.

Displaying details

Now it's time to show the fragment. Jump over to the `PodcastActivity` and wire it up.

Much of the code you're about to write should look familiar from your previous experience with managing fragments. If you need a refresher, check out chapter 11, "Using Fragments".

Open **PodcastActivity.kt** and add the following:

```
companion object {
  private val TAG_DETAILS_FRAGMENT = "DetailsFragment"
}
```

This defines a tag to uniquely identify the details fragment in the fragment manager.

Add the following method to `PodcastActivity`:

```
private fun createPodcastDetailsFragment():
    PodcastDetailsFragment {
  // 1
  var podcastDetailsFragment = supportFragmentManager
      .findFragmentByTag(TAG_DETAILS_FRAGMENT) as
      PodcastDetailsFragment?

  // 2
  if (podcastDetailsFragment == null) {
    podcastDetailsFragment =
        PodcastDetailsFragment.newInstance()
  }

  return podcastDetailsFragment
}
```

This method will either create the details fragment or use an existing instance if it exists. Let's take a closer look at how it works:

1. `supportFragmentManager.findFragmentByTag()` is used to check if the fragment already exists.

2. If there is no existing fragment, then a new one is created using the `newInstance()` method on the fragment's companion object.

3. The fragment object is returned.

When the detail fragment is shown, it's a good idea to hide the search icon. But first, you need to save a reference to the search icon menu item to allow you to hide/show the icon.

Add the following property to the top of the class:

```
private lateinit var searchMenuItem: MenuItem
```

In `onCreateOptionsMenu()`, remove the `var` keyword from the line that assigns `searchMenuItem`:

```
searchMenuItem = menu.findItem(R.id.search_item)
```

Update the next line in `onCreateOptionsMenu()` to remove the `?` safe call operator:

```
val searchView = searchMenuItem.actionView as SearchView
```

Now you can add the method that displays the details fragment:

```
private fun showDetailsFragment() {
  // 1
  val podcastDetailsFragment = createPodcastDetailsFragment()
  // 2
  supportFragmentManager.beginTransaction().add(
      R.id.podcastDetailsContainer,
      podcastDetailsFragment, TAG_DETAILS_FRAGMENT)
        .addToBackStack("DetailsFragment").commit()
  // 3
  podcastRecyclerView.visibility = View.INVISIBLE
  // 4
  searchMenuItem.isVisible = false
}
```

Let's take a look at what's going on with that method:

1. The details fragment is created or retrieved from the fragment manager.

2. The fragment is added to the `supportFragmentManager`. The `TAG_DETAILS_FRAGMENT` constant you defined earlier is used to identify the fragment. `addToBackStack()` is used to make sure the back button works to close the fragment.

3. The main podcast `RecyclerView` is hidden so the only thing showing is the detail fragment.

4. The `searchMenuItem` is hidden so the the search icon is not shown on the details
    screen.

> **Note**: Adding the fragment to the back stack is important for proper app
> navigation. If you don't add the call to `addToBackStack()`, then pressing the back
> button, while the fragment is displayed, will close the app.

Add the following to the bottom of `onCreateOptionsMenu()` before the `return`:

```
if (podcastRecyclerView.visibility == View.INVISIBLE) {
  searchMenuItem.isVisible = false
}
```

This ensures that the `searchMenuItem` remains hidden if `podcastRecyclerView` is not
visible.

You may be asking, "Why is this added to `onCreateOptionsMenu()`?"

Great question! `onCreateOptionsMenu()` is called a second time when the fragment is
added. Even though you hid the `searchMenuItem` in `showDetailsFragment()`, it will get
shown again when the menu is recreated. This is because you requested that the
fragment add to the options menu, so Android recreates the menu from scratch when
adding the fragment.

The next thing to do is replace `onShowDetails()` with code that loads the
`PodcastViewModel` and calls `showDetailsFragment()`. Before you do that, define the
following helper method:

```
private fun showError(message: String) {
  AlertDialog.Builder(this)
      .setMessage(message)
      .setPositiveButton(getString(R.string.ok_button), null)
      .create()
      .show()
}
```

This displays a generic alert dialog with an error message. You'll show this dialog to
handle all error cases.

To define the `ok_button` string, add the following line to **strings.xml**:

```
<string name="ok_button">OK</string>
```

Next you'll create the `PodcastViewModel` that will be used to hold the podcast details
view data.

Add the following property to **PodcastActivity.kt**:

```
private lateinit var podcastViewModel: PodcastViewModel
```

Add the following to the bottom of `setupViewModels()`:

```
podcastViewModel = ViewModelProviders.of(this)
    .get(PodcastViewModel::class.java)
podcastViewModel.podcastRepo = PodcastRepo()
```

This initializes the `podcastViewModel` object when the activity is created. If the activity is being created for the first time, `ViewModelProviders` will create a new instance of the `PodcastViewModel` object. If it's just a configuration change, it will use an existing copy of the `PodcastViewModel` object instead.

Now the `podcastViewModel` object is ready to use when `onShowDetails()` is called in response to the user tapping on a podcast row. Let's code that now.

Replace `onShowDetails()` with the following:

```
override fun onShowDetails(podcastSummaryViewData:
    SearchViewModel.PodcastSummaryViewData) {
  // 1
  val feedUrl = podcastSummaryViewData.feedUrl ?: return
  // 2
  showProgressBar()
  // 3
  podcastViewModel.getPodcast(podcastSummaryViewData, {
    // 4
    hideProgressBar()
    if (it != null) {
      // 5
      showDetailsFragment()
    } else {
      // 6
      showError("Error loading feed $feedUrl")
    }
  })
}
```

This method will be called when the user taps on a podcast. Here's how it works:

1. The `feedUrl` is taken from the `podcastSummaryViewData` object if it's not `null`, otherwise the method returns without doing anything.

2. The progress bar is displayed to show the user that the app is busy loading the podcast data.

3. `podcastViewModel.getPodcast()` is called to load the podcast view data.

4. After the data is returned, the progress bar is hidden.

5. If the data is not `null`, then `showDetailsFragment()` is called to display the detail fragment.

6. If the data is `null`, then the error dialog is displayed.

Build and run the app. Search for a podcast and then tap on one. The detail screen will display showing the podcast image, title and the temporary description. The SUBSCRIBE menu option is shown but not yet functional.
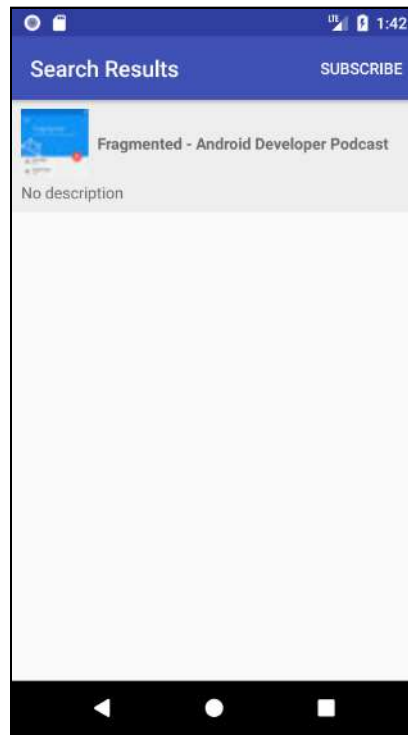
Tap the back button, and the detail fragment should go away. However, there's something wrong! The search icon is missing and the display is blank. Where did the list of podcasts go?

The problem exists because the `podcastRecyclerView` was hidden before the details fragment was displayed, but it was never made visible again!

You need to make the `podcastRecyclerView` visible again, but how do you know when the details fragment is closed?

One solution is to add a listener to `supportFragmentManager` so you're notified when the back stack changes.

Back in **PodcastActivity.kt**, add the following method:

```
private fun addBackStackListener()
{
  supportFragmentManager.addOnBackStackChangedListener {
    if (supportFragmentManager.backStackEntryCount == 0) {
      podcastRecyclerView.visibility = View.VISIBLE
    }
  }
}
```

This adds a lambda method that can respond to changes in the fragment back stack. This is called when items are added or removed from the stack. If the backStackEntryCount is 0, then all fragments have been removed and it's safe to make the podcast RecyclerView visible again.

Add the following line to the end of onCreate():

```
addBackStackListener()
```

This will add the back stack listener to the fragment manager when the activity is created.

Build and run the app. Bring up the detail screen and tap the back button. The screen will now look correct.
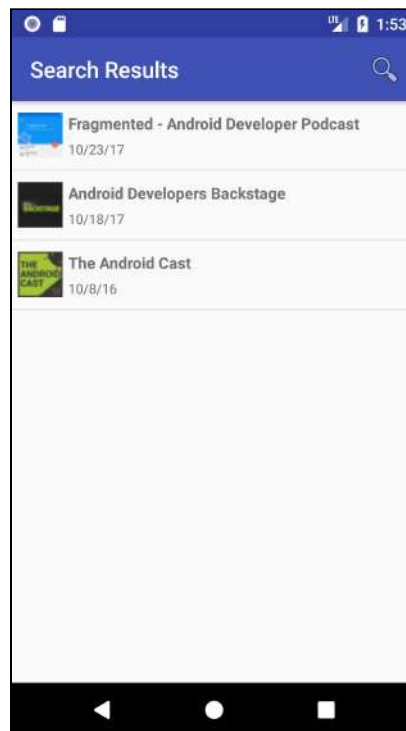
Before you call it a day, try and rotate the screen. You'll get an interesting mash up of the search results and the podcast details. Whoops!
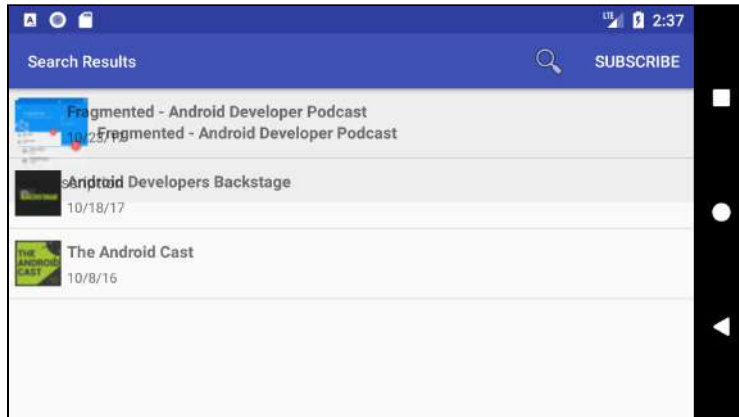
As this test demonstrates, your Android UI is not complete until you've tested it by rotating the screen. Fortunately, this is an easy fix: you need to hide the podcast RecyclerView after a configuration change.

Add the following in onCreateOptionsMenu() after the line that calls searchView.setSearchableInfo():

```
if (supportFragmentManager.backStackEntryCount > 0) {
  podcastRecyclerView.visibility = View.INVISIBLE
}
```

Now, when the device is rotated, the activity is created again. When onCreateOptionsMenu() is called — and if there are any fragments on the back stack — the podcastRecyclerView is hidden.

Build and run the app. For one last time in this chapter, bring up the detail screen for a podcast and rotate the device. The screen will now look as expected.
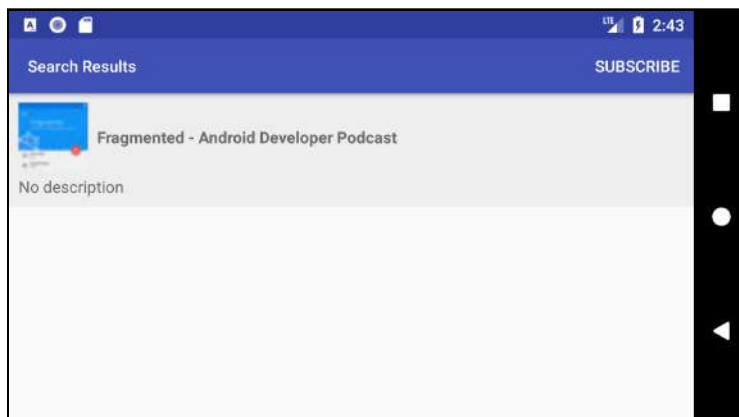
Where to go from here?

Congratulations, you made a lot of progress, but the detail screen is still missing some key information, including the list of podcast episodes and the ability to subscribe to the podcast.

But don't worry. You'll fix this in the next chapter by fetching the actual RSS feed and using it to add these missing pieces.

Chapter 23: Podcast Episodes

By Tom Blankenship

Until this point, you've only dealt with the top-level podcast details. Now it's time to dive deeper into the podcast episode details, and that involves loading and parsing the RSS feeds.

In this chapter, you'll accomplish the following:

1.   Use OkHttp to load an RSS feed from the internet.

2.   Parse the details in an RSS file.

3.   Display the podcast episodes.

If you are following along with your own app, open it and keep going with it for this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **PodPlay** app under the **starter** folder. The first time you open the project, Android Studio takes a few minutes to set up your environment and update dependencies.

Getting started

In previous chapters, you worked with the iTunes Search API, which is great for getting the basics about a podcast. But what if you need more information? What if you're looking for information about the individual episodes? That's where RSS feeds come into play!

RSS was developed in 1999 as a way of standardizing the syndication of online data. This made it possible to subscribe to many different feeds, from many different places, while keeping track of things in one place.

RSS feeds are formatted using XML 1.0, and they initially stored only textual data. But that all changed in 2000 when podcasting adopted RSS feeds and started adding media files. With the release of RSS 0.92, a new element was added: the enclosure element.

> **Note**: Although it's not necessary to fully understand how feeds are formatted, it's not a bad idea to read the full RSS specification, which can be found at http://www.rssboard.org/rss-specification.

Let's take a look at a sample RSS file for a fictitious podcast:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rss xmlns:itunes="http://www.itunes.com/dtds/podcast-1.0.dtd"
    version="2.0">
  <channel>
    <title>Android Apprentice Podcast</title>
    <link>http://rw.aa.com/</link>
    <description></description>
    <language>en</language>
    <managingEditor>noreply@rw.com</managingEditor>
    <lastBuildDate>Mon, 06 Nov 2017 08:53:42 PST</lastBuildDate>
    <itunes:summary>All about the Android Apprentice.</itunes:summary>
    <item>
      <title>Episode 999: Kotlin Basics</title>
      <link>http://rw.aa.com/episode-999.html</link>
      <author>developers@rw.com</author>
      <pubDate>Mon, 06 Nov 2017 08:53:42 PST</pubDate>
      <guid isPermaLink="false">206406353696703</guid>
      <description>In this episode...</description>
      <enclosure url="https://rw.aa.com/Kotlin.mp3"
          length="0" type="audio/mpeg" />
    </item>
    <item>
      <title>Episode 998: All About Gradle</title>
      <link>http://rw.aa.com/episode-998.html</link>
      <author>developers@rw.com</author>
      <pubDate>Tue, 31 Oct 2017 12:55:48 PDT</pubDate>
      <guid isPermaLink="false">1586082485159</guid>
      <description>In this episode...</description>
      <enclosure url="https://rw.aa.com/Gradle.mp3"
          length="0" type="audio/mpeg" />
    </item>
  </channel>
</rss>
```

Generally speaking, podcast feeds contain a lot more data than what is shown in the example; you also don't always need everything included in the feed. Regardless of the extras, they all share some common elements.

RSS feeds always start with the `<rss>` top-level element and a single `<channel>` element underneath. The `<channel>` element holds the main podcast details. For each episode, there's an `<item>` element.

Notice the `<enclosure>` element under each `<item>`. This is the element that holds the playback media.

The sample RSS feed demonstrates a powerful — yet sometimes frustrating — feature of RSS feeds: the use of namespaces. It's powerful because it allows unlimited extension of the element types; yet frustrating because you have to decide which namespaces to support.

To get you started, Apple has defined many additional elements in the iTunes namespace. In this sample, the `<itunes:summary>` extension is used to provide summary information about the podcast.

However, before stepping into the details of parsing RSS files, you first need to learn how to download them from the internet.

In Android, there are many choices for handling network requests. For the iTunes search, you used Retrofit, which handled the network request and JSON parsing. But parsing XML podcast feeds is slightly more challenging.

Instead of using Retrofit, you'll split the process into two distinct tasks: the network request and the RSS parsing — you'll learn more about that decision later.

Using OkHttp

You'll use OkHttp to pull down the RSS file, which is already included with the Retrofit library.

Start by creating a response model to hold the parsed RSS feed response.

In the **service** package, create a new file named **RssFeedResponse.kt** and add:

```
data class RssFeedResponse(
    var title: String = "",
    var description: String = "",
    var summary: String = "",
    var lastUpdated: Date = Date(),
    var episodes: MutableList<EpisodeResponse>? = null
) {

  data class EpisodeResponse(
      var title: String? = null,
      var link: String? = null,
      var description: String? = null,
      var guid: String? = null,
      var pubDate: String? = null,
      var duration: String? = null,
      var url: String? = null,
      var type: String? = null
  )
```

```
  }
```

This represents all of the data you'll retrieve from an RSS feed.

RssFeedResponse

- title: The podcast title.

- description: The podcast description.

- summary: The podcast summary.

- lastUpdated: The last update date for the podcast.

- episodes: The list of episodes for the podcast.

EpisodeResponse

- title: The episode title.

- link: URL link to the episode media file.

- description: The episode description.

- guid: Unique ID for the episode.

- pubDate: Publication date of the episode.

- duration: Episode duration.

- url: URL to the the episode landing page.

- type: Type of media for the episode ('audio' or 'video').

Next, create a new service to process the RSS feed.

In the **service** package, create a new file named **RssFeedService.kt** and add the following:

```kotlin
class RssFeedService: FeedService  {
  override fun getFeed(xmlFileURL: String,
      callBack: (RssFeedResponse?) -> Unit) {

  }
}

interface FeedService {
  // 1
  fun getFeed(xmlFileURL: String,
      callBack: (RssFeedResponse?) -> Unit)
  // 2
  companion object {
    val instance: FeedService by lazy {
```

```
      RssFeedService()
    }
  }
}
```

This is the basic outline of the RSS feed service. It provides a generic interface named FeedService, with a single method named getFeed(). It provides a FeedService implementation named RssFeedService that will eventually implement getFeed().

Looking a little deeper at the code:

1. getFeed() takes a URL pointing to an RSS file and a callback method. After the file is loaded and parsed, the callback method gets called with the final RSS feed response.

2. A companion object is used to provide a singleton instance of the FeedService.

Now you'll start implementing the getFeed() method. The first task is to download the RSS file.

Add the following code to getFeed() in RssFeedService:

```
// 1
val client = OkHttpClient()
// 2
val request = Request.Builder()
    .url(xmlFileURL)
    .build()
// 3
client.newCall(request).enqueue(object : Callback {
  // 4
  override fun onFailure(call: Call, e: IOException) {
    callBack(null)
  }
  // 5
  @Throws(IOException::class)
  override fun onResponse(call: Call, response: Response) {
    // 6
    if (response.isSuccessful) {
      // 7
      response.body()?.let { responseBody ->
        // 8
        println(responseBody.string());
        // Parse response and send to callback
        return
      }
    }
    // 9
    callBack(null)
  }
})
```

> **Note**: Be sure to select `okhttp3.Request`, `okhttp3.Callback`, `okhttp3.Call`, `okhttp3.Response` to satisfy the `Request`, `Callback`, `Call` and `Request` dependencies.

Let's break the code apart:

1. A new instance of `OkHttpClient` is created. You'll use the OkHttp client to asynchronously fetch the RSS file. This will ensure that the main thread is not blocked during the fetch.

2. To make a call with `OkHttpClient`, an HTTP `Request` object is required. In this case, you build the object using the URL of the RSS file. If you need to have fine-grained control of the HTTP Request, you can specifying headers, caching control and the request method type.

3. Once you have a `Request` object, pass it into the `client` through the `newCall()` method, which returns a `Call` object. The `Call` object's `enqueue` method synchronously executes the `Request`. A `Callback` object is passed to `enqueue()`. When the `Request` is complete, OkHttp will call either `onFailure()` or `onResponse()` on the callback object.

4. `onFailure()` is defined to handle the call from OkHttp if the `Request` fails. The main `callBack` method is called with `null` to indicate a failure.

5. If the `Request` succeeds, `onResponse()` is called by OkHttp. The `Response` object contains all of the details about the returned object, including the HTTP status code and the main `response` body.

6. The `response` is checked for success. Behind the scenes, this is checking to see if the server hosting the RSS file returned an HTTP status code in the 200s.

7. The response body is checked for `null`.

8. The `responseBody` is converted to a string and printed out. This is just a placeholder to check that everything is returned correctly. You'll implement the actual XML parsing method later.

> **Note**: The `responseBody` object is represented as a one shot stream and can be consumed only once. Anything that reads the full stream, such as calling `string()` or `bytes()`, will empty and close the stream. Try calling `println` twice with the `responseBody.string()` and you'll see how easy it is to crash the app with an `java.lang.IllegalStateException: closed` exception!

To test the `getFeed()` method, open **PodcastRepo.kt** and add the following to the top of `getPodcast()`:

```
val rssFeedService = RssFeedService()

rssFeedService.getFeed(feedUrl, {
})
```

Build and run the app. Now find a podcast, and tap on a single episode to display the details. Take a look at the Logcat window, and view the output of the RSS XML file.

XML to DOM

Even though Retrofit can be used to parse XML and comes with a built-in XML parser, there are too many edge cases to make Retrofit usable as-is; you need to properly handle namespaces and ignore duplicate elements. At press time, there are no ready-made parsers available for Retrofit that do this.

Fortunately, the **DOM** parser provided in the standard Android libraries can read the XML data. DOM stands for **Document Object Model** and represents HTML and XML data as a node-based tree structure. The object returned from the DOM parser is a single top-level **Document** object with child **Nodes** underneath. Each node contains a node type, a list of child nodes, a name, text content and optional attributes.

Here's a simple XML file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
  <channel>
    <title>Android Apprentice Podcast</title>
    <link>http://rw.aa.com/</link>
    <item>
      <title>Episode 999: Kotlin Basics</title>
      <link>http://rw.aa.com/episode-999.html</link>
      <enclosure url="https://rw.aa.com/Kotlin.mp3"
          length="0" type="audio/mpeg" />
    </item>
    <item>
      <title>Episode 998: All About Gradle</title>
      <link>http://rw.aa.com/episode-998.html</link>
      <enclosure url="https://rw.aa.com/Gradle.mp3"
          length="0" type="audio/mpeg" />
    </item>
  </channel>
</rss>
```

Parsing this file results in the following tree structure:

```
rss
+--channel
    |--title
    |--link
    |--item
    |   |--title
    |   |--link
    |   +--enclosure
    +--item
        |--title
        |--link
        +--enclosure
```

The names shown in the tree are taken from the node name property. If an XML element contains attributes, such as url in `<enclosure>`, the node will store those in an attributes array. All of the data within a node is stored in the `textContent` property. The key to parsing nodes into your data model structure is recognizing the correct node types, and then identifying the node's location within the tree.

Before writing the parser, you first need to read the RSS file into a **Document** object. The `Document` object represents the top-level node in the XML tree and derives from the `Node` class.

In `getFeed()`, replace the call to `println`, and the comment underneath it, with the following:

```
val dbFactory = DocumentBuilderFactory.newInstance()
val dBuilder = dbFactory.newDocumentBuilder()
val doc = dBuilder.parse(responseBody.byteStream())
```

`DocumentBuilderFactory` provides a factory that can be used to obtain a parser for XML documents. `DocumentBuilderFactory.newInstance()` creates a new document builder named `dBuilder`. `dBuilder.parse()` is called with the RSS file content stream and the resulting top level XML `Document` is assigned to `doc`.

That's all there is to parsing the XML file into a DOM.

DOM parsing

Now you need to turn the **Document** object into an **RssFeedResponse**.

First, add a helper method to convert from an XML date string to a `Date` object.

Open **DateUtils.kt** and add the following method:

```
fun xmlDateToDate(date: String?): Date {
  val date = date ?: return Date()
  val inFormat = SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss z")
  return inFormat.parse(date)
}
```

This converts a date string found in the RSS XML feed to a `Date` object.

Open **RssFeedService.kt** and add the following method to the `RssFeedService` class:

```
private fun domToRssFeedResponse(node: Node,
    rssFeedResponse: RssFeedResponse) {
  // 1
  if (node.nodeType == Node.ELEMENT_NODE) {
    // 2
    val nodeName = node.nodeName
    val parentName = node.parentNode.nodeName
    // 3
    if (parentName == "channel") {
      // 4
      when (nodeName) {
        "title" -> rssFeedResponse.title = node.textContent
        "description" -> rssFeedResponse.description = node.textContent
        "itunes:summary" -> rssFeedResponse.summary = node.textContent
        "item" -> rssFeedResponse.episodes?.
            add(RssFeedResponse.EpisodeResponse())
        "pubDate" -> rssFeedResponse.lastUpdated =
            DateUtils.xmlDateToDate(node.textContent)
      }
    }
```

```
  }
  // 5
  val nodeList = node.childNodes
  for (i in 0 until nodeList.length) {
    val childNode = nodeList.item(i)
    // 6
    domToRssFeedResponse(childNode, rssFeedResponse)
  }
}
```

This is a simplified version of the final parser. It only parses the top-level RSS feed info. You'll add item parsing next.

This method is designed to be recursive. It operates on a single node at a time, and then calls itself to process each child node of the current node.

Don't worry if this block seems a little confusing at this point. It will become more clear when you add episode item parsing next.

Here's what's going on with this code:

1. First the nodeType is checked to make sure it's an XML element.

2. You store the node's name and parent name. Each node, except the top-level one, contains a parent node. You use the name of the parent node to determine where the current node resides in the tree.

3. If the current node is a child of the channel node, extract the top level RSS feed information from this node.

4. You use the when expression to switch on the nodeName. Depending on the name, you fill in top level rssFeedResponse data with the textContent of the node. If the node is an episode item, a new empty EpisodeResponse object is added to the episodes list.

5. nodeList is assigned to the list of child nodes for the current node.

6. For each child node, you call domToRssFeedResponse(), passing in the existing rssFeedResponse object. This allows domToRssFeedResponse() to keep building out the rssFeedResponse object in a recursive fashion.

Now you just need to call domToRssFeedResponse(), and pass in the Document XML object and a new RssFeedResponse object.

Add the following after the assignment of the doc variable in `rssFeed()`:

```
val rssFeedResponse = RssFeedResponse(episodes = mutableListOf())
domToRssFeedResponse(doc, rssFeedResponse)
callBack(rssFeedResponse)
println(rssFeedResponse)
```

This creates a new empty `RssFeedResponse` and then calls `domToRssFeedResponse()` to parse the RSS document into the `rssFeedResponse` object. It then passes the `rssFeedResponse` to the `callBack` method and prints out the results.

Build and run the app. Once again, locate and display a podcast episode.

Look at the Logcat window and you'll see that the `RssFeedResponse` top-level information has been populated, along with a series of blank episode items.

```
11-12 09:38:58.270 10101-10254/com.raywenderlich.podplay I/System.out: RssFeedResponse(title=Fragmented — Android
Developer Podcast, description=The Fragmented Podcast is a podcast for Android Developers hosted by Donn Felker and
Kaushik Gopal. Our goal is to help you become a better Android Developer. We chat about topics such as Testing,
Dependency Injection, Android Patterns and Practices, useful libraries and much more. We will also be interviewing
some of the top Android Developers out there. Subscribe now and join us on the journey of being an Android Developer.,
 summary=The Fragmented Podcast is a podcast for Android Developers hosted by Donn Felker and Kaushik Gopal. Our goal
is to help you become a better Android Developer. We chat about topics such as Testing, Dependency Injection, Android
Patterns and Practices, useful libraries and much more. We will also be interviewing some of the top Android
Developers out there. Subscribe now and join us on the journey of being an Android Developer., lastUpdated=Sun Nov 12
09:38:58 EST 2017, episodes=[EpisodeResponse(title=null, link=null, description=null, guid=null, pubDate=null,
duration=null, url=null, type=null), EpisodeResponse(title=null, link=null, description=null, guid=null, pubDate=null,
 duration=null, url=null, type=null), EpisodeResponse(title=null, link=null, description=null, guid=null,
pubDate=null, duration=null, url=null, type=null), EpisodeResponse(title=null, link=null, description=null, guid=null,
 pubDate=null, duration=null, url=null, type=null), EpisodeResponse(title=null, link=null, description=null,
guid=null, pubDate=null, duration=null, url=null, type=null), EpisodeResponse(title=null, link=null, description=null,
 guid=null, pubDate=null, duration=null, url=null, type=null), EpisodeResponse(title=null, link=null,
description=null, guid=null, pubDate=null, duration=null, url=null, type=null), EpisodeResponse(title=null, link=null,
```

You're now ready to finish out the `domToRssFeedResponse()` by adding the episode item parsing.

In `domToRssFeedResponse()`, add the following below the assignment of `parentName`:

```
// 1
val grandParentName = node.parentNode.parentNode?.nodeName ?: ""
// 2
if (parentName == "item" && grandParentName == "channel") {
  // 3
  val currentItem = rssFeedResponse.episodes?.last()
  if (currentItem != null) {
    // 4
    when (nodeName) {
      "title" -> currentItem.title = node.textContent
      "description" -> currentItem.description = node.textContent
      "itunes:duration" -> currentItem.duration = node.textContent
      "guid" -> currentItem.guid = node.textContent
      "pubDate" -> currentItem.pubDate = node.textContent
      "link" -> currentItem.link = node.textContent
      "enclosure" -> {
        currentItem.url = node.attributes.getNamedItem("url")
            .textContent
        currentItem.type = node.attributes.getNamedItem("type")
```

```
                   .textContent
        }
      }
    }
  }
```

Let's take a look at what's going on with this code:

1. In addition to the name of the parent node, you also need to know the name of the parent of the parent; in other words, the grandparent node.

2. If this node is a child of an `item` node, and the `item` node is a child of a `channel` node, then you know it is an episode element.

3. Because the parsing is recursive, you know that the parent `item` was parsed already and an empty episode object was added to `episodes` list in the `rssFeedResponse` object. The `currentItem` variable is assigned to the last episode in the `episodes` list.

4. The `when` expression is used to switch on the current node's name. Based on the node name, the current episode item's details are populated from the node's `textContent` property. If the node is an enclosure, the `url` and `type` are extracted from the node's attributes and set on the `currentItem`.

Build and run the app. Just as before, locate and display a podcast episode.

Look at the Logcat window and you'll see that the `RssFeedResponse` is now fully populated with podcasts and episode details.

```
11-12 10:30:06.644 10482-10578/com.raywenderlich.podplay I/System.out: , guid=https://www.signalleaf
.com/podcasts/Fragmented/554ae00f33b8570300079b47, pubDate=Wed, 06 May 2015 13:44:00 +0000,
duration=01:19:26, url=https://audio.simplecast.com/ef2c9510.mp3, type=audio/mpeg), EpisodeResponse
(title=006: Jake Wharton on Becoming a Better Developer and Creating Successful Open Source Projects
(Part 1), link=null, description=In part one of this two-part segment, we talk to the one and only
Jake Wharton. He gives us the scoop on how he operates day to day, what he looks for in a good
Android developer and how to become a better Android developer. He also touches upon the various
sources and non-Java platforms that he draws inspiration from. Finally, he talks about open source
and gives tips on leading an open source project.
11-12 10:30:06.644 10482-10578/com.raywenderlich.podplay I/System.out: , guid=https://www.signalleaf
.com/podcasts/Fragmented/5541ac620374a203003d7438, pubDate=Wed, 29 Apr 2015 14:13:00 +0000,
duration=00:54:14, url=https://audio.simplecast.com/017d8790.mp3, type=audio/mpeg), EpisodeResponse
(title=005: Image libraries for Android, link=null, description=In this episode of Fragmented, Donn
and Kaushik start off by discussing the tips and tricks available for efficiently loading images in
an Android app. Good image libraries make use of these techniques and perform all the heavy lifting
in the background. So they then discuss the different image library options available for Android
developers.
```

Congratulations, you created an RSS feed service that returns an RSS response object for any feed you throw at it!

Now you can use the new `RssFeedService` to revisit the `PodcastRepo` class and add in the missing podcast details from earlier.

Updating the podcast repo

Open **PodcastRepo.kt** and update the class declaration to the following:

```
class PodcastRepo(private var feedService: FeedService) {
```

This declares a new `feedService` property that will be passed into the constructor.

Now you need need a helper method to convert the `RssResponse` data into `Episode` and `Podcast` objects.

Add the following method:

```
private fun rssItemsToEpisodes(episodeResponses:
List<RssFeedResponse.EpisodeResponse>): List<Episode> {
  return episodeResponses.map {
    Episode(
        it.guid ?: "",
        it.title ?: "",
        it.description ?: "",
        it.url ?: "",
        it.type ?: "",
        DateUtils.xmlDateToDate(it.pubDate),
        it.duration ?: ""
    )
  }
}
```

This uses the `map` method to convert a list of `EpisodeResponse` objects into a list of `Episode` objects. The `pubDate` string is converted to a `Date` object using the new `xmlDateToDate` method.

With this method in place, you can convert the full `RssFeedResponse` to a `Podcast` object.

Add the following new method:

```
private fun rssResponseToPodcast(feedUrl: String, imageUrl:
    String, rssResponse: RssFeedResponse): Podcast? {
  // 1
  val items = rssResponse.episodes ?: return null
  // 2
  val description = if (rssResponse.description == "")
      rssResponse.summary else rssResponse.description
  // 3
  return Podcast(feedUrl, rssResponse.title, description, imageUrl,
      rssResponse.lastUpdated, episodes = rssItemsToEpisodes(items))
}
```

Here's what's happening with the code:

1. The list of episodes is assigned to the `items` variable provided it's not `null`; otherwise the method returns `null`.

2. If the `description` is empty, the `description` property is set to the response `summary`; otherwise it's set to the response `description`.

3. A new `Podcast` object is created using the response data and it's returned to the caller.

Now you can update the `getPodcast()` method to use the new capabilities.

Since `feedService.getFeed()` is using the OkHttp client to retrieve the podcast feed asynchronously, it will execute the `callBack` method in a background thread.

To prevent problems with updating UI elements, you'll use coroutines to jump back to the main thread before returning the podcast details from the podcast repo.

Open the project **build.gradle** file and add the following to the `ext` element:

```
coroutines_version = '0.19.3'
```

Open the application **build.gradle** file and add the following to the `dependencies` element:

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:
$coroutines_version"
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:
$coroutines_version"
```

Now, sync the project.

Back in **PodcastRepo.kt**, replace the contents of `getPodcast()` with the following:

```
feedService.getFeed(feedUrl, { feedResponse ->
  var podcast: Podcast? = null
  if (feedResponse != null) {
    podcast = rssResponseToPodcast(feedUrl, "", feedResponse)
  }

  launch(UI) {
    callback(podcast)
  }
})
```

If the `feedResponse` is `null`, then a `null` is passed to the `callBack` method. If `feedResponse` is valid, then it's converted to a `Podcast` object and passed to the `callback` method.

Note that the calls to the callback method are surrounded with launch(UI). This is a feature of coroutines that forces the enclosing code to run on the main thread.

Episode list adapter

In previous chapters, you defined a RecyclerView in the podcast detail layout and created a layout for the podcast episode items for the rows. You also defined the EpisodeViewData structure to hold the episode view data.

Now, you just need to add a list adapter to populate the RecyclerView using EpisodeViewData items.

In the **adapter** package, create a new file named **EpisodeListAdapter.kt** and replace the contents with the following:

```kotlin
class EpisodeListAdapter(
    private var episodeViewList: List<EpisodeViewData>?) :
    RecyclerView.Adapter<EpisodeListAdapter.ViewHolder>() {

  class ViewHolder(v: View) : RecyclerView.ViewHolder(v) {
    var episodeViewData: EpisodeViewData? = null
    val titleTextView: TextView = v.findViewById(R.id.titleView)
    val descTextView: TextView = v.findViewById(R.id.descView)
    val durationTextView: TextView = v.findViewById(R.id.durationView)
    val releaseDateTextView: TextView =
        v.findViewById(R.id.releaseDateView)
  }

  fun setViewData(episodeList: List<EpisodeViewData>) {
    episodeViewList = episodeList
    this.notifyDataSetChanged()
  }

  override fun onCreateViewHolder(parent: ViewGroup,
                                  viewType: Int):
 EpisodeListAdapter.ViewHolder {
    return ViewHolder(LayoutInflater.from(parent.context)
        .inflate(R.layout.episode_item, parent, false))
  }

  override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    val episodeViewList = episodeViewList ?: return
    val episodeView = episodeViewList[position]

    holder.episodeViewData = episodeView
    holder.titleTextView.text = episodeView.title
    holder.descTextView.text =  episodeView.description
    holder.durationTextView.text = episodeView.duration
    holder.releaseDateTextView.text = episodeView.releaseDate.toString()
  }
```

```
    override fun getItemCount(): Int {
      return episodeViewList?.size ?: 0
    }
  }
```

This is a standard list adapter that creates `RecyclerView` items from a list of `EpisodeViewData` objects. You've seen this pattern several times in previous chapters, so we'll skip the detailed explanation and move on to hooking up the adapter in the podcast detail fragment.

Updating the view model

Now that `PodcastRepo` uses the `RssFeedService` to retrieve the podcast details, the view model set up in `PodcastActivity` needs to be updated to match.

Open **PodcastActivity.kt** and replace the assignment of `podcastViewModel.podcastRepo` in **setupViewModels()** with the following:

```
val rssService = FeedService.instance
podcastViewModel.podcastRepo = PodcastRepo(rssService)
```

This creates a new instance of the `FeedService` and uses it to create a new `PodcastRepo` object. The `PodcastRepo` object is assigned to the `podcastViewModel.podcastRepo` property.

All that's left to do now is to set up the `RecyclerView` with the `EpisodeListAdapter`.

RecyclerView set up

Open **PodcastDetailsFragment.kt** and add the following property to the class:

```
private lateinit var episodeListAdapter: EpisodeListAdapter
```

Add the following new method:

```
private fun setupControls() {
  // 1
  feedDescTextView.movementMethod = ScrollingMovementMethod()
  // 2
  episodeRecyclerView.setHasFixedSize(true)

  val layoutManager = LinearLayoutManager(activity)
  episodeRecyclerView.layoutManager = layoutManager

  val dividerItemDecoration =
      android.support.v7.widget.DividerItemDecoration(
          episodeRecyclerView.context, layoutManager.orientation)
  episodeRecyclerView.addItemDecoration(dividerItemDecoration)
  // 3
```

```
    episodeListAdapter = EpisodeListAdapter(
        podcastViewModel.activePodcastViewData?.episodes)
    episodeRecyclerView.adapter = episodeListAdapter
  }
```

Here's what's going on:

1. This allows the feed title to scroll if it gets too long for its container.

2. This section is standard set up code for the episode list `RecyclerView`.

3. The `EpisodelistAdapter` is created with the list of episodes in the `activePodcastViewData` object and assigned to the `episodeRecyclerView`.

In `onActivityCreated()`, add the call to `setupControls()`, before the call to `updateControls()`:

```
  setupControls()
```

Build and run the app. Once again, find a podcast and display the details for an episode.
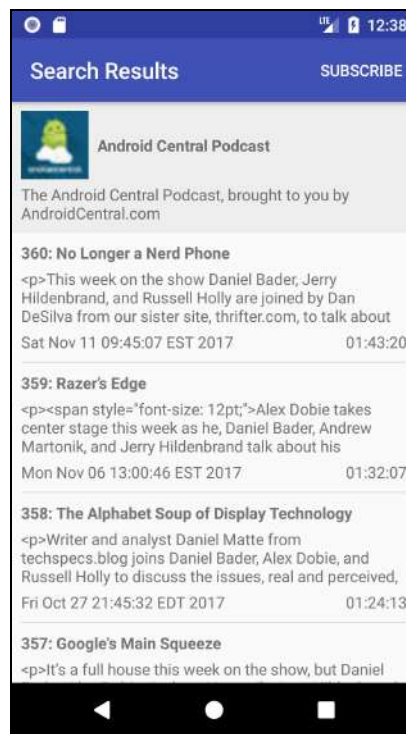
Podcast details cleanup

That's not too shabby, but a couple of items need a little cleanup. For some podcasts, the episode text may contain HTML formatting which will need some extra processing. You also need to format the dates on the episodes.

To fix the HTML formatting, create a utility method that uses an Android built-in method for converting HTML text into a series of character sequences which can be rendered properly in a standard `TextView`.

In the **util** package, create a new file named **HtmlUtils.kt** and replace the contents with the following:

```
object HtmlUtils {
  fun htmlToSpannable(htmlDesc: String): Spanned {
    // 1
    var newHtmlDesc = htmlDesc.replace("\n".toRegex(), "")
    newHtmlDesc = newHtmlDesc.replace("(<(/)img>)|(<img.+?>)".
        toRegex(), "")

    // 2
    val descSpan: Spanned
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.N) {
      descSpan = Html.fromHtml(newHtmlDesc, Html.FROM_HTML_MODE_LEGACY)
    } else {
      @Suppress("DEPRECATION")
      descSpan = Html.fromHtml(newHtmlDesc)
    }
    return descSpan
  }
}
```

A single `htmlToSpannable` method is defined to convert an HTML string into a spanned character sequence. Here's how it works:

1. Before converting the text to a `Spanned` object, some initial cleanup is required. These two lines strip out all `\n` characters and `<img>` elements from the text.

2. Android's `Html.fromHtml` method is used to convert the text to a `Spanned` object. This breaks the text down into multiple sections that Android will render with different styles.

> **Note**: The second parameter to `fromHtml()` is a flag added in Android N. This version of the call is only made if the app is running on Android N or higher. The flag can be set to either `Html.FROM_HTML_MODE_LEGACY` or `Html.FROM_HTML_MODE_COMPACT`, and controls how much space is added between block-level elements. The earlier version of `fromHtml()` has been deprecated, but is still required when running on Android M or lower. `@Suppress("DEPRECATION")` is used to allow the code to compile even though it is deprecated.

Now you'll update the list adapter to fix the text formatting as it populates the `TextView` widgets.

Open **EpisodeListAdapter.kt**. In `onBindViewHolder()`, replace the line that assigns **holder.descTextView.text** with the following:

```
holder.descTextView.text =
    HtmlUtils.htmlToSpannable(episodeView.description ?: "")
```

That takes care of the episode descriptions. You're ready to clean up the episode date display. First, add a new helper method to convert a `Date` object to a short date formatted string. Open **DateUtils.kt** and add the following method:

```
fun dateToShortDate(date: Date): String {
  val outputFormat = DateFormat.getDateInstance(
      DateFormat.SHORT, Locale.getDefault())
  return outputFormat.format(date)
}
```

This is the same code you used in `jsonDateToShortDate()` to create a locale-aware short date string.

Go back to **EpisodeListAdapter.kt**. In `onBindViewHolder()`, replace the line that assigns **holder.releaseDateTextView.text** with the following:

```
holder.releaseDateTextView.text = episodeView.releaseDate?.let {
  DateUtils.dateToShortDate(it)
}
```

If the `releaseDate` is not `null`, then it's converted to a short date string and assigned to the episode date text view. Build and run the app, and display the details for a podcast. The episode text and date formatting look much better now!
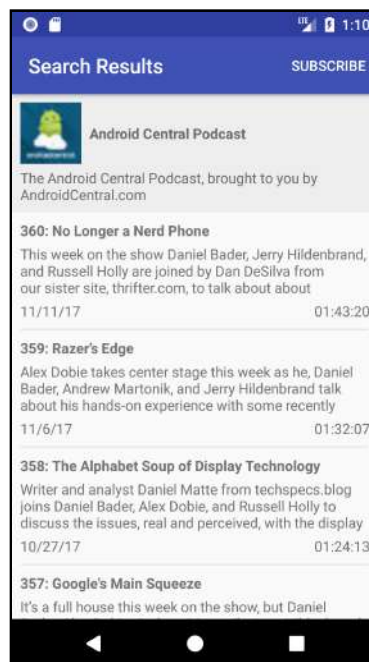
Where to go from here?

In the next chapter, you'll finally hook up the SUBSCRIBE button and build out the persistence layer, which will let users store podcast data offline.

Chapter 24: Podcast Subscriptions Part One

By Tom Blankenship

By giving users the ability to search for podcasts and displaying the podcast episodes, you made great progress in the development of the podcast app. In this section, you'll add the ability to subscribe to favorite podcasts.

Over the next two chapters, you'll add the following features to the app:

1. Storing the podcast details and episode lists locally for quick access. (this chapter)

2. Displaying the list of subscribed podcasts by default. (this chapter)

3. Notifying the user when new episodes are available. (next chapter)

You'll cover several new topics over the span of these two chapters including:

1. Using Room to store multiple related database tables.

2. Using `JobScheduler` services to periodically check for new episodes.

3. Using local notifications to alert users when new episodes are available.

Getting started

If you are following along with your own app, open it and keep going with it for this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **PodPlay** app under the **starter** folder. The first time you open the project, Android Studio takes a few minutes to set up your environment and update dependencies.

Saving podcasts

The first new feature you'll implement is the ability to track podcast subscriptions. You'll take the models already created and make them persistent entities by adding Room attributes. The database will only contain podcasts the user subscribes to.

Your first goal is to hook up the subscribe menu item so it will save the current podcast.

Setting up the database code follows the same general approach used in the **MapBook** app:

1. Annotate the podcast and episode models with Room attributes.

2. Create a database access object (DAO) used by the repositories in the app.

3. Create a `RoomDatabase` object to manage the models and provide the DAO.

Things will be slightly more difficult this time because you have two models — podcast and episode — to manage instead of just one. You'll also have to manage the relationship between these two models. For example, if a podcast is deleted from the database, all associated episodes should also be deleted from the database. However, don't fret! This is only slightly more difficult; Room will do all the heavy lifting for you.

The database diagram will look like this:
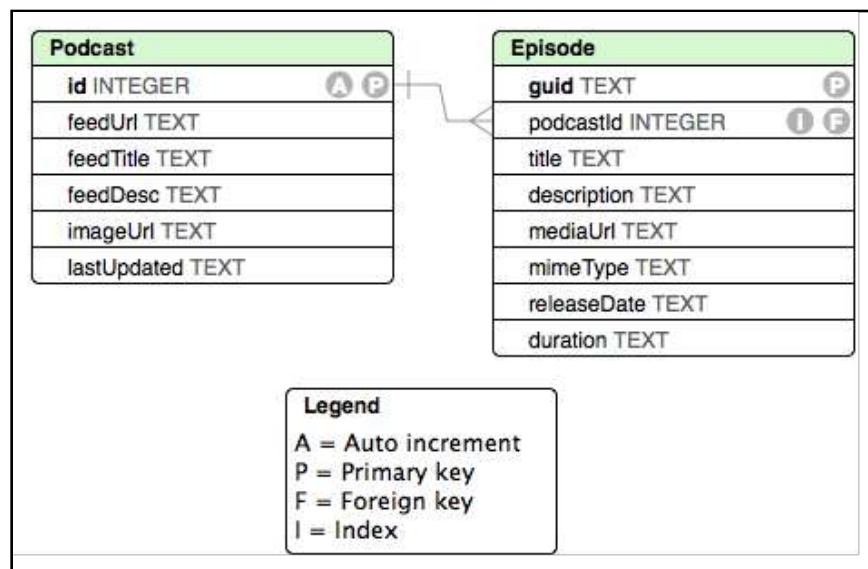
If you recall, the `Episode` table is nearly a one-to-one match with the `Episode` model. The only difference here is in the table — we add a foreign ID (`podcastId`) pointing the model back to a `Podcast` model. We'll dive more deeply into this relationship later in the chapter.

Adding Room support

Before getting into the code, you need to bring in the **Room** libraries.

Open the project **build.gradle** file and add the following to the `buildscript.ext` section:

```
room_version = '1.0.0'
```

Open the application module **build.gradle** file and add the following to the dependencies:

```
implementation "android.arch.persistence.room:runtime:$room_version"
annotationProcessor "android.arch.persistence.room:compiler:
$room_version"
kapt "android.arch.persistence.room:compiler:$room_version"
```

These are the same libraries you used in the **PlaceBook** app. See "Chapter 16: Saving bookmarks with Room" for details about these dependencies.

Sync the gradle file.

Annotating the models

Your first task is to properly annotate the existing models so Room knows how to store the data. Start by getting the `Podcast` class into shape.

Open **Podcast.kt** and update the class declaration with the `@Entity` annotation, like so:

```
@Entity
data class Podcast(...)
```

The `@Entity` annotation is the basic requirement for a class managed by Room.

Next, you'll need to add a primary key for the `Podcast` table.

Add the following as the first property declaration to the `Podcast` class:

```
@PrimaryKey(autoGenerate = true) var id: Long? = null,
```

This defines an `id` property that will auto generate as new items are added to the `Podcast` table.

By adding a new property to the class constructor, you broke the code that constructs a `Podcast` object. Fortunately, this only occurs in one place in the app.

Open **PodcastRepo.kt** and update the return call in `rssResponseToPodcast()` to match the following:

```
return Podcast(null, feedUrl, rssResponse.title, description, imageUrl,
    rssResponse.lastUpdated, episodes = rssItemsToEpisodes(items))
```

The only change is to pass in `null` for the `id` argument.

Now, bring the `Episode` class up-to-speed and make it an official database entity.

Open **Episode.kt** and update the class declaration with the `@Entity` annotation:

```
@Entity(
    foreignKeys = [
      ForeignKey(
          entity = Podcast::class,
          parentColumns = ["id"],
          childColumns = ["podcastId"],
          onDelete = ForeignKey.CASCADE
      )
    ],
    indices = [Index("podcastId")]
)
data class Episode (
```

> **Note**: If given a choice for importing `ForeignKey` and `Index`, make sure to select the following versions respectively:
>
> import android.arch.persistence.room.ForeignKeyimport
> android.arch.persistence.room.Index

Here, you're just adding a couple of new attributes to define a foreign key and an index on the database.

When you have multiple entities or models that are related, it's helpful to let Room know about these relationships. The `foreignKeys` attribute lets you define these relationships and add constraints on them. This helps maintain the database integrity without any extra work on your part.

In this case, you define a single `ForeignKey` that relates the `podcastId` property in the `Episode` entity to the property `id` in the `Podcast` entity. There are four fields defined on the `ForeignKey` attribute:

1. `entity`: Defines the parent entity.

2. `parentColumns`: Defines the column names on the parent entity (the `Podcast` class).

3. `childColumns`: Defines the column names in the child entity (the `Episode` class).

4. `onDelete`: Defines the behavior when the parent entity is deleted. `CASCADE` indicates that any time a podcast is deleted, all related child episodes will be deleted automatically.

Room recommends creating an index on the child table. This prevents a full scan of the database when performing cascading operations. In this case, the `indices` attributes defines the `podcastId` property as the index.

There is no need to add a new property for the `PrimaryKey` attribute on the `Episode` entity. Instead, you'll use the existing `guid` property. In database terminology this is known as a natural key, where the `id` you added to the `Podcast` class acts as a surrogate key.

The purpose of a primary key is to provide a unique value for each row in the database, and the `guid` value naturally meets this criteria.

Update the `guid` property with the `@PrimaryKey` annotation as follows:

```
@PrimaryKey var guid: String = "",
```

Now you need to add the `podcastId` property that defines the foreign key to the `Podcast` entity. Add the following property below the `guid` and above the `title` property in `Episode`:

```
var podcastId: Long? = null,
```

Now that you've add a new property to the constructor, you need to fix any places in the code that create a new `Episode`. Open **PodcastRepo.kt** and update the return call in `rssItemsToEpisodes()` with the following:

```
return episodeResponses.map {
  Episode(
      it.guid ?: "",
      null,
      it.title ?: "",
      it.description ?: "",
      it.url ?: "",
      it.type ?: "",
      DateUtils.xmlDateToDate(it.pubDate),
      it.duration ?: ""
  )
}
```

For the second argument, you pass in `null` for the `podcastId`. You'll fill in this value after inserting the parent `Podcast` into the database.

Data access object

Before you can define the main Room database object, you'll create the DAO to read and write to the database. This is where you define all of the SQL statements for the basic database operations.

You'll add additional methods later, but for now, all you need is the ability to save and load podcasts and their corresponding episodes.

Under **com.raywenderlich.podplay**, create a new package named **db**.

Now, under the **db** package, create a new file named **PodcastDao.kt** and replace the contents with the following:

```
// 1
@Dao
interface PodcastDao {
  // 2
  @Query("SELECT * FROM Podcast ORDER BY FeedTitle")
  fun loadPodcasts(): LiveData<List<Podcast>>
  // 3
  @Query("SELECT * FROM Episode WHERE podcastId = :arg0
      ORDER BY releaseDate DESC")
  fun loadEpisodes(podcastId: Long): List<Episode>
  // 4
  @Insert(onConflict = REPLACE)
  fun insertPodcast(podcast: Podcast): Long
  // 5
  @Insert(onConflict = REPLACE)
  fun insertEpisode(episode: Episode): Long
}
```

> **Note**: If given a choice for importing `Query` and `REPLACE`, make sure to select the following versions respectively:
>
> ```
> import android.arch.persistence.room.Queryimport
> android.arch.persistence.room.OnConflictStrategy.REPLACE
> ```

Let's break the code down a bit:

1. The `PodcastDao` interface is defined with the `@Dao` annotation. This indicates to the Room library that this is a managed DAO class.

2. `loadPodcasts()` loads all of the podcasts from the database and returns a `LiveData` object. The `@Query` annotation is defined to select all podcasts and sort them by their title in ascending order.

3. `loadEpisodes()` loads all of the episodes from the database. The `@Query` annotation is defined to select all episodes that match a single `podcastId` and sort them by the release date in descending order.

4. `insertPodcast()` inserts a single podcast into the database. No SQL statement is required on the `@Insert` annotation. `onConflict` is set to `REPLACE` to tell Room to replace the old record if a record with the same primary key already exists in the database.

5. `insertEpisode()` inserts a single episode into the database.

Define the Room database

All that's left to do is define the Room database object and have it instantiate the `PodcastDao` object.

Create a new file named **PodPlayDatabase.kt** in the **db** package and replace the contents with the following:

```kotlin
// 1
@Database(entities = arrayOf(Podcast::class, Episode::class),
    version = 1)
abstract class PodPlayDatabase : RoomDatabase() {
  // 2
  abstract fun podcastDao(): PodcastDao
  // 3
  companion object {
    // 4
    private var instance: PodPlayDatabase? = null
    // 5
    fun getInstance(context: Context): PodPlayDatabase {
      if (instance == null) {
        // 6
        instance = Room.databaseBuilder(context.applicationContext,
            PodPlayDatabase::class.java, "PodPlayer").build()
      }
      // 7
      return instance as PodPlayDatabase
    }
  }
}
```

Let's take a look at what the code is doing:

1. `PodPlayDatabase` is defined as an abstract class that implements the `RoomDatabase` interface. The `@Database` annotation is used to define this as a Room database with two tables: `Podcast` and `Episode`.

2. The abstract method `podcastDao` is defined to return a `PodcastDao` object. Room will take care of creating the final implementation of the `PodcastDao` class.

3. A companion object is defined to hold the single instance of the `PodPlayDatabase`.

4. The single instance of the `PodPlayDatabase` is defined and set to `null`.

5. `getInstance()` will return a single application-wide instance of the `PodPlayDatabase`.

6. If an instance of `PodPlayDatabase` hasn't been created before, it will be created now. `Room.databaseBuilder()` is used to instantiate the `PodPlayDatabase` object.

7. The `PodPlayDatabase` object is returned to the caller.

Go ahead and build the project using Command-F9 (Control-F9 on Windows) and you will get the following errors from the compiler:

```
– Cannot figure out how to save this field into database. You can
consider adding a type converter for it.
– Cannot figure out how to read this field from a cursor.
```

Unfortunately, Android Studio may not point you to the location of the errors.

The error message is telling you that Room doesn't know how to handle one or more of the fields in your models. Why is that? Because Room only knows how to deal with basic and boxed basic types, not complex types.

> **Note**: A boxed basic type is one that has been wrapped in an object so it can be made nullable. For example, `Integer` is the boxed type for the basic type `int`.

Looking at the `Podcast` and `Episode` models, there are three complex properties:

In `Podcast`:

```
var lastUpdated: Date = Date()
var episodes: List<Episode> = listOf()
```

In `Episode`:

```
var releaseDate: Date = Date()
```

To handle the `Date` and `List<Episode>` complex types, you'll use something called **TypeConverters**.

Room type converters

Although Room can't handle complex types directly, it provides a concept known as **TypeConverters** that let you define how to convert them to-and-from basic types. This is the perfect solution for the `Date` properties.

The `List<Episode>` property is another matter. In this case, you're not trying to store episodes in the `Podcast` table, instead you are defining a relationship to `Episode` objects stored in the `Episode` table.

Let's take care of the `Date` properties first and then address the episodes reference.

All you need to do is let Room know how to convert a date to a basic type and then back again. Using type converters, you can easily convert the `Date` object to a `Long`, and a `Long` back to a `Date`.

Open **PodPlayDatabase.kt** and add the following class before the `PodPlayDatabase` class definition:

```kotlin
class Converters {
  @TypeConverter
  fun fromTimestamp(value: Long?): Date? {
    return if (value == null) null else Date(value)
  }

  @TypeConverter
  fun toTimestamp(date: Date?): Long? {
    return (date?.time)
  }
}
```

> **Note**: If given a choice of imports for `Date`, make sure to use `java.util.Date`

The `Converters` class is a holder for the two `TypeConverter` methods. `fromTimestamp()` converts a `Long` to a `Date`, and `toTimestamp()` converts a `Date` to a `Long`. The `@TypeConverter` annotation is required on all type converters.

To let Room know to use these type converters, you need to add a new annotation to the `PodPlayDatabase` class.

In the `PodPlayDatabase` class, sandwich a `@TypeConverters` annotation between the `@Database` annotation and the class declaration, so it looks like this:

```kotlin
@Database(entities = arrayOf(Podcast::class, Episode::class),
    version = 1)
@TypeConverters(Converters::class)
abstract class PodPlayDatabase : RoomDatabase() {...}
```

This tells Room to look in the `Converters` class to find all methods annotated by `@TypeConverter`. Room will recognize the two methods for handling `Dates`, and it will call them when reading and writing the `releaseDate` and `lastUpdated` fields to the database.

Room object references

Now back to the episodes list in the `Podcast` model. Since Room does not support defining object references in `Entity` classes, you need to tell it to ignore the `episodes` property.

> **Note**: You may be wondering why Room doesn't allow object references. That's a valid question, and the Room designers have some good reasons why this isn't allowed. If you're curious about the reasons, the following page gives a good explanation: https://developer.android.com/training/data-storage/room/referencing-data.html#understand-no-object-references.

Open **Podcast.kt** and update the `episodes` property to match the following:

```
@Ignore
var episodes: List<Episode> = listOf()
```

With this field ignored, Room will not attempt to populate it when loading a `Podcast` from the database.

Build the app again to verify the errors are gone.

That handles all of the database access layer, now you need to define some methods in the podcast repo to read and write podcasts and episodes.

Update the podcast repo

The podcast repo currently uses only the `RssFeedService` to retrieve podcast data. One benefit of using the repository pattern is that a single repository can access data from multiple sources or services.

Now it's time to add the ability for the podcast repo to access the podcast DAO in addition to the feed service.

Open **PodcastRepo.kt** and update the constructor from this:

```
class PodcastRepo(private var feedService: FeedService) {
```

To this:

```
class PodcastRepo(private var feedService: FeedService,
                  private var podcastDao: PodcastDao) {
```

This adds a new property to hold the `PodcastDao` object.

Next you need to update the podcast activity to properly instantiate the `PodcastRepo` class with the new `podcastDao` property.

Open **PodcastActivity.kt** and replace the following line in `setupViewModels()`:

```
podcastViewModel.podcastRepo = PodcastRepo(rssService)
```

with this:

```
val db = PodPlayDatabase.getInstance(this)
val podcastDao = db.podcastDao()
podcastViewModel.podcastRepo = PodcastRepo(rssService, podcastDao)
```

An instance of the `PodPlayDatabase` is created and the `PodcastDao` object is retrieved from it. The `PodcastRepo` is updated to pass in the podcast DAO object in addition to the RSS service.

Great! Now you can go back to the podcast repo and update it with the database access methods.

Open **PodcastRepo.kt** and add the following method:

```
fun save(podcast: Podcast) {
  launch(CommonPool) {
    // 1
    val podcastId = podcastDao.insertPodcast(podcast)
    // 2
    for (episode in podcast.episodes) {
      // 3
      episode.podcastId = podcastId
      podcastDao.insertEpisode(episode)
    }
  }
}
```

This method uses the `podcastDao` object to insert a `Podcast` and its associated `Episodes` into the database.

Let's take a closer look at how this works:

1. First, the `Podcast` is inserted into the database. `insertPodcast()` returns the new primary key assigned to the podcast.

2. The `for` loop is used to walk through each episode belonging to the podcast.

3.   The episode's `podcastId` is assigned to the `id` of the inserted `Podcast` to create a relationship between the two.

4.   Finally, the `episode` is inserted in the database.

Now that the episode is *in* the database, you need a method to load it *from* the database.

Add the following new method:

```
fun getAll(): LiveData<List<Podcast>>
{
  return podcastDao.loadPodcasts()
}
```

This just passes the `LiveData` object from the DAO through to the caller.

Updating the view model

One more step is needed before you can connect the subscribe menu item. Since the view only talks to the view model, you need to update the podcast view model to use the new repository methods.

First, you need a method to save a podcast. To make it easy to save the currently loaded podcast, add a new property to store the active podcast. This will be updated any time the view loads a new podcast.

Open **PodcastViewModel.kt** and add the following property to the top of the class:

```
private var activePodcast: Podcast? = null
```

In `getPodcast()`, after the line that reads `activePodcastViewData = podcastToPodcastView(it)`, add the following:

```
activePodcast = it
```

This will assign the `activePodcast` to the podcast loaded by the `getPodcast()` method. This allows the podcast view model to keep track of the most recently loaded podcast.

Now you can add a method to save the active podcast. Add the following method:

```
fun saveActivePodcast() {
  val repo = podcastRepo ?: return
  activePodcast?.let {
    repo.save(it)
  }
}
```

This method first checks to make sure the `podcastRepo` and the `activePodcast` are not `null`. If they're both not `null`, then the `activatePodcast` is saved to the repo.

The final addition to the view model is a method to return a view of all the subscribed podcasts.

You'll return a `LiveData` version of the podcasts formatted for the summary view.

When you built out the search feature, the `SearchViewModel` class used a summary view model to return data for the search results. You can reuse this model to format the list of subscribed podcasts.

First, add the following method that converts from a podcast model to a summary view model.

```
private fun podcastToSummaryView(podcast: Podcast):
    PodcastSummaryViewData {
  return PodcastSummaryViewData(
      podcast.feedTitle,
      DateUtils.dateToShortDate(podcast.lastUpdated),
      podcast.imageUrl,
      podcast.feedUrl)
}
```

Next, create a method that returns the `LiveData` list of podcast summary view objects. It's designed to be invoked multiple times, yet only create the `LiveData` object once.

Add the following property to the top of the class:

```
var livePodcastData: LiveData<List<PodcastSummaryViewData>>? = null
```

This is used to hold the `LiveData` list of podcast view objects.

Add the following new method:

```
fun getPodcasts(): LiveData<List<PodcastSummaryViewData>>? {
  val repo = podcastRepo ?: return null
    // 1
  if (livePodcastData == null) {
    // 2
    val liveData = repo.getAll()
    // 3
    livePodcastData = Transformations.map(liveData) { podcastList ->
      podcastList.map { podcast ->
        podcastToSummaryView(podcast)
      }
    }
  }

  // 4
  return livePodcastData
}
```

Let's take a closer look:

1. If `livePodcastData` is `null`, create it.

2. The `LiveData` object is retrieved from the podcast repo. This is the list of `Podcast` data objects that now needs to be converted to versions formatted for the view.

3. The list of `LiveData` podcast objects are converted to a list of `LiveData` `PodcastSummaryViewData` objects.

4. The `livePodcastData` object is returned to the caller.

Connecting the subscribe menu item

Everything is now in place to hook-up the subscribe menu item on the podcast detail screen.

The activity is the best place to determine what action should take place and then update the view accordingly. Therefore, the detail fragment will listen for the tap on the menu item, and the podcast activity will handle the action.

Open **PodcastDetailsFragment.kt** and add the following to the end of the class.

```
interface OnPodcastDetailsListener {
  fun onSubscribe()
}
```

`PodcastDetailsFragment` will require its parent activity — in this case, the `PodcastActivity` — to implement the interface and will call the `onSubscribe()` method when the user activates the menu item.

You might be wondering why you should bother adding this level of abstraction? Why not just use `PodcastActivity` directly? Because doing it this way is considered good practice if you plan on using `PodcastDetailsFragment` in other activities.

Add the following property and method to `PodcastDetailsFragment`:

```
private var listener: OnPodcastDetailsListener? = null
```

```
override fun onAttach(context: Context?) {
  super.onAttach(context)
  if (context is OnPodcastDetailsListener) {
    listener = context
  } else {
    throw RuntimeException(context!!.toString() +
        " must implement OnPodcastDetailsListener")
  }
}
```

The property holds a reference to the listener. `onAttach()` is called by the fragment manager when the fragment is attached to its parent activity. The `context` argument is a reference to the parent activity. If the activity implements the `OnPodcastDetailsListener` interface, then you assign the listener property to it. If it doesn't implement the interface, then an exception is thrown.

Now you just need to listen for the user tapping on the subscribe menu item, and call the `onSubscribe` method on the listener.

Add the following override method:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
  when (item.itemId) {
    R.id.menu_feed_action -> {
      podcastViewModel.activePodcastViewData?.feedUrl?.let {
        listener?.onSubscribe()
      }
      return true
    }
    else ->
      return super.onOptionsItemSelected(item)
  }
}
```

`onOptionsItemSelected()` is called when the user selects a menu item. If the menu `itemId` matches the `menu_feed_action` (subscribe) item, and the active podcast is not `null`, then `onSubscribe()` is called on the `listener`.

Perfect! Now you need to jump back to the activity to handle the `onSubscribe()` call.

Open **PodcastActivity.kt** and update the class declaration as follows:

```
class PodcastActivity : AppCompatActivity(), PodcastListAdapterListener,
    OnPodcastDetailsListener {
```

To implement the `OnPodcastDetailsListener` interface add the following method:

```
override fun onSubscribe() {
  podcastViewModel.saveActivePodcast()
  supportFragmentManager.popBackStack()
}
```

Here, you're using the view model to save the active podcast and then you remove the `PodcastDetailsFragment` by calling `popBackStack()` on the fragment manager.

Displaying subscribed podcasts

That completes the code to subscribe to a podcast. Of course, subscribing to a podcast is not very useful if you don't let the user see their subscriptions!

The main podcast activity already contains a recycler view that displays a list of podcasts generated from search results. You can reuse the same recycler view to display a list of subscribed podcasts.

The idea is that the app will initially display the subscribed podcasts; when the user performs a search, those will be replaced with the search results.

You'll start by updating the podcast activity to load all of the podcasts and display them in the recycler view when the view is first created.

Open **PodcastActivity.kt** and add the following method:

```kotlin
private fun showSubscribedPodcasts()
{
  // 1
  val podcasts = podcastViewModel.getPodcasts()?.value
  // 2
  if (podcasts != null) {
    toolbar.title = getString(R.string.subscribed_podcasts)
    podcastListAdapter.setSearchData(podcasts)
  }
}
```

Let's take a look at what's going on with this code:

1. `getPodcasts()` is called on the view model to get the podcasts `LiveData` object. The `value` is the most recently returned object of the `LiveData` instance. This value may be `null` if the `LiveData` object does not have any observers attached yet, but you'll observe the `LiveData` object when the activity is created.

2. If `podcasts` is not `null`, then the podcast list adapter is updated with the `podcasts` object.

Add the following line to **strings.xml** to satisfy the `subscribed_podcasts` resource reference.

```xml
<string name="subscribed_podcasts">Subscribed</string>
```

Add the following method back in **PodcastActivity.kt**:

```kotlin
private fun setupPodcastListView() {
  podcastViewModel.getPodcasts()?.observe(this, Observer {
    if (it != null) {
      showSubscribedPodcasts()
    }
  })
}
```

> **Note**: If given import options on `Observer` make sure to choose `import`
> `android.arch.lifecycle.Observer`.

You'll call this method when the activity is created. It calls `getPodcasts()` on the view model and observes the changes to the data. When the data changes, `showSubscribedPodcasts()` is called and the podcast list adapter is updated with the latest list of podcasts.

Now you just need to call `setupPodcastListView()` when the view is created.

In `onCreate()`, add the following line after the call to `updateControls()`:

```
setupPodcastListView()
```

Build and run the app.

Search and display the details for a podcast. Tap the subscribe button, and the app will return to the search results.

Behind the scenes, the `Observer` you created in `setupPodcastListView()` will be called when the database is updated with the subscribed podcast. This will, in turn, update the `RecyclerView` and display the podcast in the list.
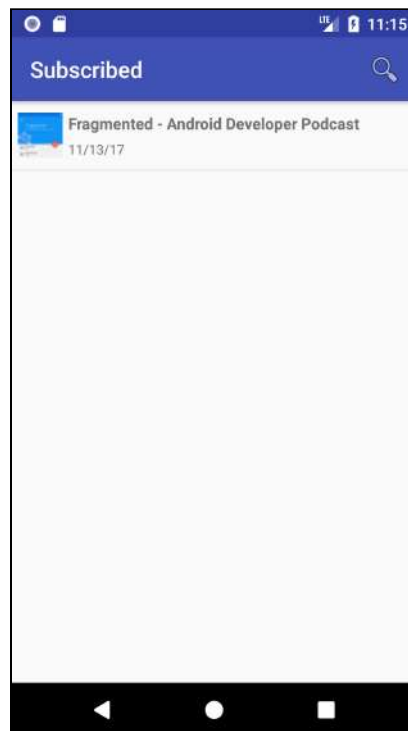
This is working fairly well, but there are a few things that need to be cleaned up:

1. When you tap on a subscribed podcast, it loads the episodes from the feed URL instead of using what you already have stored in the database. This may not be obvious at first, but if you disable your internet connection, the issue will become clear!

2. You can subscribe to a podcast more than once and it will keep adding to the list.

3. You can't unsubscribe to a podcast.

4. There is no way to get back to the subscribed podcast lists once you do a search.

You can fix the first issue by updating the podcast repo to check the database before it fetches a feed from the internet. First, you need a new method in the DAO that loads a podcast from the database based on the feed URL. Open **PodcastDao.kt** and add the following method:

```
@Query("SELECT * FROM Podcast WHERE feedUrl = :arg0")
fun loadPodcast(url: String): Podcast?
```

Next, you need to update the repo logic so it attempts to load from the database first.

Open **PodcastRepo.kt** and add the following to the beginning of `getPodcast()`:

```
launch(CommonPool) {

  val podcast = podcastDao.loadPodcast(feedUrl)

  if (podcast != null) {
    podcast.id?.let {
      podcast.episodes = podcastDao.loadEpisodes(it)
      launch(UI){
        callback(podcast)
      }
    }
  } else {
```

Also, add a closing brace to the end of `getPodcast()`:

```
  }
}
```

This attempts to load the podcast from the database. If the podcast is not `null`, then it loads in the matching episodes from the database and passes the podcast to the callback method.

If the podcast is `null`, then the existing code block executes and loads the podcast from the internet.

To fix the second and third problems, you need to make the detail fragment a little smarter. That means it needs to recognize the subscription status of a podcast; if already subscribed, the menu item shows as "Unsubscribe"; if not, the menu item shows as "Subscribe".

First, you need the view to determine if a podcast is subscribed to or not.

The `PodcastViewData` object already has a `subscribed` property, but it's not being used yet. So it's time to update the view model to set the subscribed property.

Open **PodcastViewModel.kt** and update the return call in `podcastToPodcastView()`:

```
return PodcastViewData(
    podcast.id != null,
    podcast.feedTitle,
    podcast.feedUrl,
    podcast.feedDesc,
    podcast.imageUrl,
    episodesToEpisodesView(podcast.episodes)
)
```

The only change is to the first parameter passed into `PodcastViewData`, which is the `subscribed` flag. If a podcast contains a non-null `id` value, that means it was loaded from the database. You can use that to determine how to set the `subscribed` property on `PodcastViewData`. Set it to `true` if the podcast `id` is not equal to `null`; or `false` if it is.

Now you can update the detail fragment so it sets the state of the subscribe menu item based on the value stored in the `subscribed` property. You can also update the details listener interface to support an unsubscribe action.

Open **PodcastDetailsFragment.kt** and add the following line to the `OnPodcastDetailsListener` interface declaration:

```
fun onUnsubscribe()
```

In order to update the menu item text to dynamically display either "Subscribe" or "Unsubscribe", you need to save the `MenuItem` in a local property.

Add the following property to the `PodcastDetailsFragment` class:

```
private var menuItem: MenuItem? = null
```

A new method is needed in order to update the menu item title based on the subscribed state of the podcast.

Add the following method:

```
private fun updateMenuItem() {
  // 1
  val viewData = podcastViewModel.activePodcastViewData ?: return
  // 2
  menuItem?.title = if (viewData.subscribed)
      getString(R.string.unsubscribe) else getString(R.string.subscribe)
}
```

The code you just added:

1. Verifies that there is an active podcast on the view model.

2. Sets the menu item title based on the `subscribed` property. If the user already subscribed to the podcast, the title is set to "Unsubscribe"; if not, the title is set to "Subscribe".

Add the following line to **strings.xml** to define the `R.string.unsubscribe` string resource.

```
<string name="unsubscribe">Unsubscribe</string>
```

Now you can assign the `menuItem` property to the menu action item and call `updateMenuItem()`.

Back in **PodcastDetailsFragment.kt**, add the following to the end of `onCreateOptionsMenu()`:

```
menuItem = menu?.findItem(R.id.menu_feed_action)
updateMenuItem()
```

This assigns the `menuItem` property to the menu item widget and then calls `updateMenuItem()`.

That's enough to set the correct menu item title, now you need to update the menu action handling code to subscribe or unsubscribe based on the current state.

Update the line in `onOptionsItemSelected()` from this:

```
    listener?.onSubscribe()
```

To this:

```
if (podcastViewModel.activePodcastViewData?.subscribed) {
  listener?.onUnsubscribe()
} else {
  listener?.onSubscribe()
}
```

If the podcast is already subscribed to, then call `onUnsubscribe()` on the listener. If the podcast is not subscribed to, then call `onSubscribe()` on the listener.

To complete this feature, you just need to define the `onUnsubscribe()` method in the podcast activity. Unsubscribing requires removing the podcast from the database, so you'll need some additional database code first.

Open **PodcastDao.kt** and add the following method:

```
@Delete
fun deletePodcast(podcast: Podcast)
```

Note: Deleting the podcast will automatically delete all related episodes thanks to the foreign key defined in the `@Entity` annotation on the `Episode` model.

Open **PodcastRepo.kt** and add the following method:

```
fun delete(podcast: Podcast) {
  launch(CommonPool) {
    podcastDao.deletePodcast(podcast)
  }
}
```

This calls the `deletePodcast` method in the background.

Open **PodcastViewModel.kt** and add the following method:

```
fun deleteActivePodcast() {
  val repo = podcastRepo ?: return
  activePodcast?.let {
    repo.delete(it)
  }
}
```

This method first checks to make sure the `podcastRepo` and the `activePodcast` are not `null`. If both are not `null`, then the `activatePodcast` is deleted from the repo.

Open **PodcastActivity.kt** and add the following method:

```
override fun onUnsubscribe() {
  podcastViewModel.deleteActivePodcast()
  supportFragmentManager.popBackStack()
}
```

This uses the view model to delete the active podcast and then removes the podcast details fragment.

> **Note**: If you created duplicate podcast entries by subscribing to the same one multiple times, you'll need to delete the app before running it again. If you don't do this, the database will not load in the existing episodes correctly.

Build and run the app.

Tap on a previously subscribed podcast to display the details screen. The menu action will now show "UNSUBSCRIBE".

Tap on "UNSUBSCRIBE". The app will return to the main activity, and the podcast will be gone.
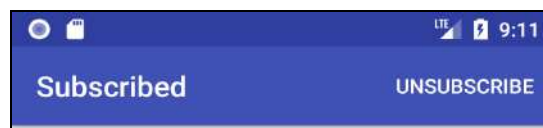
The final issue you'll address is getting back to the subscribed podcast list after performing a search.

This is easy enough to correct by listening for the search menu item to close, and then reloading the subscribed podcast list.

Menu items in Android allow you to assign a listener object that responds to the menu expanding and collapsing. You'll assign the listener, and listen for the collapse action to indicate when the subscribed podcast should be shown again.

Open **PodcastActivity.kt**. In `onCreateOptionsMenu()`, after the assignment of the `searchMenuItem`, add the following:

```
searchMenuItem.setOnActionExpandListener(object:
MenuItem.OnActionExpandListener {
  override fun onMenuItemActionExpand(p0: MenuItem?): Boolean {
    return true
  }
  override fun onMenuItemActionCollapse(p0: MenuItem?): Boolean {
    showSubscribedPodcasts()
    return true
  }
})
```

An `OnActionExpandListener` object is defined with two required overrides and assigned using `setOnActionExpandListener()`. You're not interested in the menu item expanding, so the `onMenuItemActionExpand()` method is empty.

`onMenuItemActionCollapse()` is called when the user closes the search widget. In response, `showSubscribedPodcasts()` is called to display the subscribed podcast items in place of the search results.

Build and run the app.

Search for a podcast, and then press the back arrow to close out the search widget. The display will return back to the list of subscribed podcasts.

Where to go from here?

Good job! You made it through the first part of podcast subscriptions. Take a breather, and pick up with part two when you're ready to finish!

Chapter 25: Podcast Subscriptions Part Two

By Tom Blankenship

Now that the user can subscribe to podcasts, it's helpful to notify them when new episodes are available. In this chapter, you'll update the app to periodically check for new episodes in the background and post a notification if any are found.

Getting started

If you are following along with your own app, the starter project for this chapter includes an additional icon that you'll need to complete the section. You can either begin this chapter with the **starter** project or copy the following resources from the starter project into yours:

- `src/main/res/drawable-hdpi/ic_episode_icon.png`
- `src/main/res/drawable-mdpi/ic_episode_icon.png`
- `src/main/res/drawable-xhdpi/ic_episode_icon.png`
- `src/main/res/drawable-xxhdpi/ic_episode_icon.png`

When you're done, the **res\drawable** folder in Android Studio should look like this:
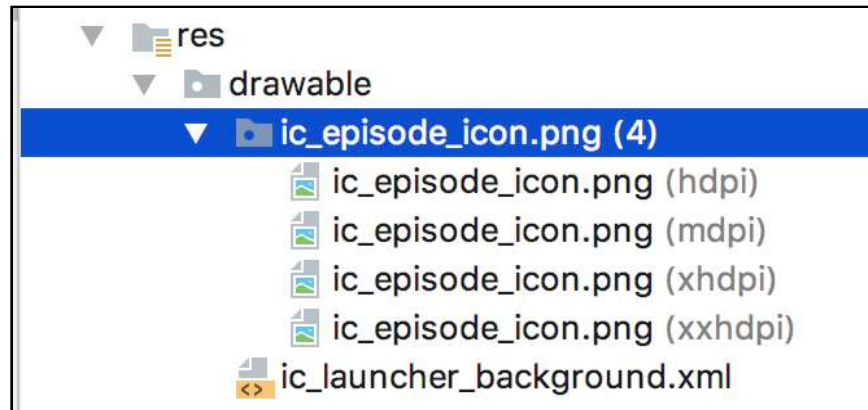
Background methods

Checking for new episodes should happen automatically at regular intervals whether the app is running or not. There are several methods available for an application to perform tasks when it's not running. It's important to choose the correct one so that it doesn't affect the performance of other running applications.

There are four primary methods to run tasks in the background:

Alarms

You can use the `AlarmManager` class to wake up the app at a specified time so it can perform operations. An `Intent` is sent to the application to wake it up, and then it can perform the work.

This is not intended for doing tasks at regular intervals, and therefore not a good solution for this app.

Broadcasts

You can register to receive broadcasts from the system for certain events and then perform tasks. This option is highly restricted to a limited number of broadcasts in apps that target API level 26 or higher.

This is obviously not an option for running a task at regular intervals.

Services

Android provides foreground and background services.

Foreground services are intended to perform work that is visible to the user. For example, in the next chapter, you'll use a foreground service to play podcasts that will keep playing when the app does not have focus.

Background services are intended for operations that are not visible to the user. Due to concerns with the performance of multiple application running background services at the same time, Android does not allow them for apps targeting API level 26 or higher.

This option is also not a good fit for the PodPlay app.

Scheduled jobs

This is the approach Google recommends for most background operations. You can specify detailed criteria about when the job will run. Android intelligently determines the best time and takes advantage of system idle time.

This sounds like the perfect choice for periodically checking for new episodes, but there's one issue. The platform supported API for scheduling jobs is through the `JobScheduler` class. The `JobScheduler` was introduced with API level 21, and at the time of this writing, Google has not released a backward compatible version.

This means you can only use `JobScheduler` if you're targeting API 21 or higher. To support back to API 19, a nice alternative is the **Firebase JobDispatcher**.

Firebase JobDispatcher

Firebase JobDispatcher is an open source library created by Google that has a similar API to the `JobScheduler`, but works all the way back to API 9. The only additional requirement is that the user's device must have Google Play services installed.

Before getting into the details of the `JobDispatcher`, you'll build out the underlying logic to update podcast episodes.

Here's a diagram showing how it will all fit together:
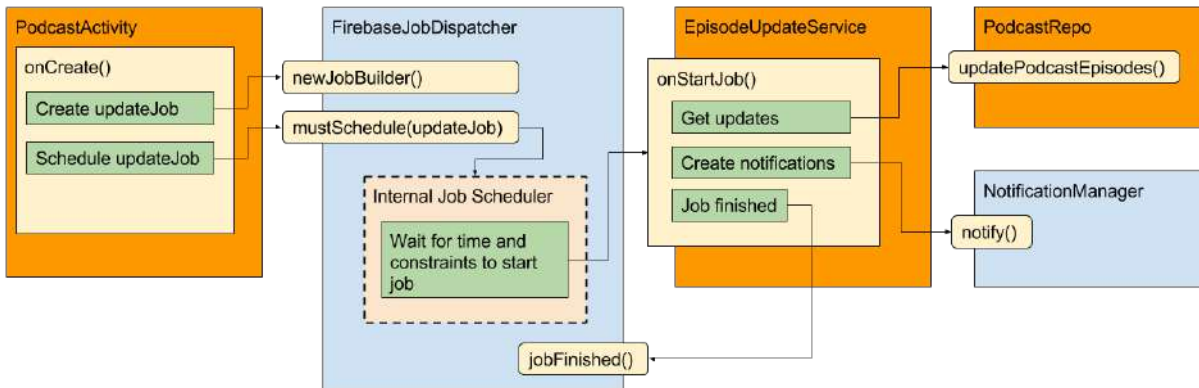
Episode update logic

To keep with the current architecture of using the repo for updating podcast data, you'll add a new method in the repo to handle the episode update logic.

The update logic will work as follows:

1. Walk through all subscribed podcasts.

2. Download the latest podcast feed.

3. Determine which episodes are new.

4. Add the new episodes to the database.

5. Notify the user when new episodes are available.

Because LiveData doesn't do much good in the background, you need a method in the DAO class to load the podcasts and episodes without using the LiveData wrapper.

Open **db\PodcastDao.kt** and add the following method:

```
@Query("SELECT * FROM Podcast ORDER BY FeedTitle")
fun loadPodcastsStatic(): List<Podcast>
```

You'll also need a method that takes a single podcast and returns a list of new episodes available.

Open **repository\PodcastRepo.kt** and add the following method:

```kotlin
private fun getNewEpisodes(localPodcast: Podcast,
    callBack: (List<Episode>) -> Unit) {
  // 1
  feedService.getFeed(localPodcast.feedUrl, { response ->
    if (response != null) {
      // 2
      val remotePodcast = rssResponseToPodcast(localPodcast.feedUrl,
        localPodcast.imageUrl, response)
      remotePodcast?.let {
        // 3
        val localEpisodes = podcastDao.loadEpisodes(localPodcast.id!!)
        // 4
        val newEpisodes = remotePodcast.episodes.filter { episode ->
          localEpisodes.find { episode.guid == it.guid } == null
        }
        // 5
        callBack(newEpisodes)
      }
    } else {
      callBack(listOf())
    }
  })
}
```

This method takes a subscribed podcast and downloads its latest episodes. This uses the network to download the episodes in the background, therefore, it accepts a `callBack` method as the second argument. It executes the `callBack` method after the episodes are retrieved. Let's walk through it step-by-step:

1. Use the `feedService` to download the latest podcast episodes.

2. Convert the `feedService response` to the `remotePodcast` object.

3. Load the list of local episodes from the database.

4. Filter the `remotePodcast` episodes to contain only the ones that are not found in the `localEpisodes` list and assign to `newEpisodes`.

5. Pass the `newEpisodes` list to the `callBack` method.

6. Return an empty list if the `feedService` does return a response.

You'll also need a new method that updates an existing podcast with a new episode.

Add the following method:

```kotlin
private fun saveNewEpisodes(podcastId: Long, episodes: List<Episode>) {
  launch(CommonPool) {
    for (episode in episodes) {
      episode.podcastId = podcastId
      podcastDao.insertEpisode(episode)
```

```
      }
    }
  }
```

This method inserts the list of `episodes` into the database for the given `podcastId`.

Before you can create the main podcast update method, you need one small class. This class will hold the update details for a single podcast.

Add the following inner class to the `PodcastRepo` class:

```
class PodcastUpdateInfo (val feedUrl: String, val name: String,
    val newCount: Int)
```

Now you're ready to create the podcast update method.

Add the following method:

```
fun updatePodcastEpisodes(callback: (List<PodcastUpdateInfo>) -> Unit) {
  // 1
  val updatedPodcasts: MutableList<PodcastUpdateInfo> = mutableListOf()
  // 2
  val podcasts = podcastDao.loadPodcastsStatic()
  // 3
  var processCount = podcasts.count()
  // 4
  for (podcast in podcasts) {
    // 5
    getNewEpisodes(podcast, { newEpisodes ->
      // 6
      if (newEpisodes.count() > 0) {
        saveNewEpisodes(podcast.id!!, newEpisodes)
        updatedPodcasts.add(PodcastUpdateInfo(podcast.feedUrl,
            podcast.feedTitle, newEpisodes.count()))
      }
      // 7
      processCount--
      if (processCount == 0) {
        // 8
        callback(updatedPodcasts)
      }
    })
  }
}
```

This method walks through all of the subscribed podcasts and updates them with the latest episodes. It executes the passed in callback method with a summary of the podcasts that were updated. Here's the step-by-step explanation:

1.   Initialize an empty list of `PodcastUpdateInfo` objects.

2.   Load the subscribed podcasts from the database without the `LiveData` wrapper.

3. `processCount` is initialized to keep track of the background processing.

4. The podcasts are processed one at a time.

5. `getNewEpisodes()` is called to fetch any new episodes. Because `getNewEpisodes()` runs in the background, it won't actually run until the loop has iterated over all podcasts and returned to the caller. The `processCount` is used as a way to track when all background processing has completed. When `processCount` reaches 0, it's time to pass the `updatedPodcasts` list to the callback method.

6. If there were new episodes, they are saved to the database, and the `updatedPodcasts` list is appended with a new `PodcastUpdateInfo` object. This object stores the feed URL, podcast name and the numbers of episodes added.

7. The process count is decremented.

8. If the process count reaches 0, indicating that all podcasts have been processed, then the callback method gets called and passes the list of updated podcasts.

Firebase JobDispatcher

Now that all of the support code is in place to update podcast episodes, you can turn your attention back to job scheduling.

Using the `JobDispatcher` class consists of the following steps:

1. Define a custom `JobService` class that executes the job logic.

2. Create a `FirebaseJobDispatcher` object.

3. Define a `Job` with the required scheduling parameters and your `JobService` class.

4. Schedule the `Job` through the `FirebaseJobDispatcher` object.

JobService

Your first task is to define a class that extends `JobService`. This class gets activated by the `JobDispatcher` when the job is ready to run.

The Firebase JobDispatcher library must be added to the project first.

Open the module **build.gradle** file and add the following line to the dependencies section:

```
implementation "com.firebase:firebase-jobdispatcher:0.8.5"
```

Sync the project.

In the **service** package, create a new Kotlin file named **EpisodeUpdateService.kt** and replace the contents with the following:

```kotlin
class EpisodeUpdateService : JobService() {

  override fun onStartJob(jobParameters: JobParameters): Boolean {
    return true
  }

  override fun onStopJob(jobParameters: JobParameters): Boolean {
    return true
  }
}
```

> **Note**: Make sure to use `com.firebase.jobdispatcher.JobService` and `com.firebase.jobdispatcher.JobParameters` instead of the built-in `android.app.job.JobService` and `android.app.job.JobParameters` imports.
>
> Kotlin's autocomplete will attempt to bring in the `JobParameters` parameters on `onStartJob` and `onStopJob` as optionals, since the Java methods where they are declared do not have nullability annotations. This is an effort to keep unannotated Java APIs from accidentally causing crashes in your Kotlin code. In this particular case, it's known from documentation that the parameters are guaranteed to be there at runtime so that you can remove the `?`. Since the method isn't annotated, the compiler won't complain.

You're required to define two methods on your `JobService` class:

- **onStartJob()**: This is where you'll perform the episode updating logic. This method should return `true` if you're processing the job on a background thread, which you will be doing. If your job does something simple and returns without starting a background thread, then you should return `false`.

  The job dispatcher will call this method when it's time for you to perform your work. A `WakeLock` will be kept on your application as long as the job is running to make sure the application doesn't get killed by the system before the job completes.

  Upon completion of your job logic, the `jobFinished()` method on your `JobService` must be called to release the `WakeLock` and prevent battery drain. Keep in mind that `onStartJob()` is called on the main thread and is expected to return to the system quickly. You'll need to make sure the episode update logic runs in the background.

- **onStopJob()**: This is where you'll stop the currently running job if it hasn't completed yet. Don't ignore this call or the app will likely not behave correctly.

  Android will release the `WakeLock` on the app when this is called. You should return `true` if you want the job to be retried again later. Return `false` if the job can be dropped.

The job dispatcher will call this method if the criteria for running the job is no longer valid. For instance, if the job should only run when the device is plugged in, then unplugging the device will trigger a call to `onStopJob()`.

Just like any other `Service` in Android, the job service must be registered in the manifest file.

Open **AndroidManifest.xml** and add the following beneath the main `activity` element, but still within the `application` element:

```
<service
    android:exported="false"
    android:name=".service.EpisodeUpdateService">
  <intent-filter>
    <action android:name="com.firebase.jobdispatcher.ACTION_EXECUTE"/>
  </intent-filter>
</service>
```

Now you can start adding some supporting methods to the job service.

The purpose of using job scheduling is to allow the episodes to be checked in the background, even if the app is not running. But what happens then? Right now, nothing happens until the user returns to the application, but they're not likely to do that if they don't know there are new episodes.

You need a way to notify the user from outside the app when new episodes are available. This is where Android Notifications come in to play.

Notifications

Notifications are Android's way of letting you display information outside of your application. The notifications appear as icons in the notification display area at the top of the screen as shown here:

You use the `NotificationManager` class to trigger notifications based on a `Notification` object that is created with the `NotificationCompat.Builder` class.

When you create a notification it requires the following items at a minimum:

1. **Small icon**: Set with `setSmallIcon()`.

2. **Title**: Set with `setContentTitle()`.

3. **Detailed text**: set with `setContentText()`.

Starting with API level 26 (Oreo), you also need a notification channel. This gives the user more control over the types of notifications they get from the application.

When you create a notification channel, you define some initial settings such as vibration, but then the user can customize each channel and decide how it behaves. For PodPlay, you'll use a single notification channel.

In addition to the required settings, there are many more ways to customize notifications. PodPlay will stick with the basics, but you're encouraged to view the documentation at https://developer.android.com/reference/android/support/v4/app/NotificationCompat.Builder.html to learn more about the other notification options.

Before creating the notification channel, you need a unique channel ID.

Open **EpisodeUpdateService.kt** and add the following companion object to the `EpisodeUpdateService` class:

```
companion object {
  val EPISODE_CHANNEL_ID = "podplay_episodes_channel"
}
```

This defines a channel ID that will identify this channel to the notification system. This can be any string that is unique to your app.

Add the following method that will create the PodPlay notification channel:

```
// 1
@RequiresApi(Build.VERSION_CODES.O)
private fun createNotificationChannel() {
  // 2
  val notificationManager =
      getSystemService(Context.NOTIFICATION_SERVICE) as
        NotificationManager
  // 3
  if (notificationManager.getNotificationChannel(EPISODE_CHANNEL_ID)
      == null) {
    // 4
    val channel = NotificationChannel(EPISODE_CHANNEL_ID, "Episodes",
        NotificationManager.IMPORTANCE_DEFAULT)
    notificationManager.createNotificationChannel(channel)
  }
}
```

1. Since notification channels are only supported in API 26 or newer, the `RequiresApi` annotation is used to notify the compiler that this method should only be called when running on API 26 or newer (in this case API 26 is the letter 'O' for 'Oreo' and therefore we use `Build.VERSION_CODES.O`).

2. The notification manager is retrieved using `getSystemService()`. You should never create the notification manager directly.

3. The notification manager is used to check if the channel already exists.

4. If the channel does not exist, then a new `NotificationChannel` object is created with the name "Episodes". The notification manager is instructed to create the channel.

Now you'll create the method to display a single notification. This method requires a couple of new string resources.

Open **res\values\strings.xml** and add the following:

```
<string name="episode_notification_title">New episodes</string>
<string name="episode_notification_text">%1$d new episode(s) for %2$s</string>
```

The `%1$d` and `%2$s` bits are placeholders for parameters that are passed in when this string is accessed.

`%1` indicates that it's a placeholder for the first parameter, `$d` indicates that this first parameter will be a digit. Similarly, `%2$s` indicates that the second parameter will be a string.

Jump back to **EpisodeUpdateService.kt** and add a new constant to the companion object:

```
val EXTRA_FEED_URL = "PodcastFeedUrl"
```

Then add the following method:

```kotlin
private fun displayNotification(podcastInfo:
    PodcastRepo.PodcastUpdateInfo) {
  // 1
  val contentIntent = Intent(this, PodcastActivity::class.java)
  contentIntent.putExtra(EXTRA_FEED_URL, podcastInfo.feedUrl)
  val pendingContentIntent = PendingIntent.getActivity(this, 0,
      contentIntent, PendingIntent.FLAG_UPDATE_CURRENT)
  // 2
  val notification = NotificationCompat.Builder(this, EPISODE_CHANNEL_ID)
      .setSmallIcon(R.drawable.ic_episode_icon)
      .setContentTitle(getString(R.string.episode_notification_title))
      .setContentText(getString(R.string.episode_notification_text,
```

```
        podcastInfo.newCount, podcastInfo.name))
    .setNumber(podcastInfo.newCount)
    .setAutoCancel(true)
    .setContentIntent(pendingContentIntent)
    .build()
  // 4
  val notificationManager =
      getSystemService(Context.NOTIFICATION_SERVICE)
        as NotificationManager
  // 5
  notificationManager.notify(podcastInfo.name, 0, notification)
}
```

> **Note**: If given the choice of imports for `NotificationCompat`, make sure to choose `android.support.v4.app.NotificationCompat`.

1. The notification manager needs to know what content to display when the user taps the notification. You do this by providing a `PendingIntent` that points to the `PodcastActivity`.

    When the user taps the notification, the system will use the intent within the `PendingIntent` to launch the `PodcastActivity`. The podcast `feedUrl` is set as an extra on the intent, and you'll use this information to display the podcast details screen.

2. The `Notification` is created with the following options:

    setSmallIcon(): Set to the PodPlay episode icon.

    setContentTitle(): This is the main title shown above the detailed text.

    setContentText(): This is the detailed text. It will let the user know the name of the podcast and the number of new episodes available.

    setNumber(): This tells Android the number of new items associated with this notification. In some cases, this number is shown to the right of the notification.

    setAutoCancel(): Setting this to `true` tells Android to clear the notification once the user taps on it.

    setContentIntent(): Sets the pending intent that was defined earlier.

3. The notification manager is retrieved using `getSystemService`.

4. The notification manager is instructed to notify the user with the notification object created by the builder.

The first parameter defines a tag, and the second parameter is an `id` number. These two items combine to create a unique name for the notification. In this case, the podcast name is unique enough, so the `id` number is always 0. If `notify()` is called multiple times with the same tag and ID then it will replace any existing notification with the same tag and id.

Finally, you're ready to update `onStartJob()` to implement update logic and trigger the notifications.

Replace the contents of `onStartJob()` with the following:

```kotlin
// 1
val db = PodPlayDatabase.getInstance(this)
val repo = PodcastRepo(FeedService.instance, db.podcastDao())
// 2
launch(CommonPool) {
  // 3
  repo.updatePodcastEpisodes({ podcastUpdates ->
    // 4
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
      createNotificationChannel()
    }
    // 5
    for (podcastUpdate in podcastUpdates) {
      displayNotification(podcastUpdate)
    }
    // 6
    jobFinished(jobParameters, false)
  })
}

return true
```

1. Instantiate a repo object.

2. Define a coroutine to run the update process in the background.

3. Call `repo.updatePodcastEpisodes()` to update the podcast episodes.

4. If the device is running Android O or later, create the required notification channel.

5. Call `displayNotification()` for each updated podcast.

6. After all of the podcasts have been processed, call `jobFinished()` to let the job dispatcher know that the job is complete.

JobDispatcher Scheduling

Now that `EpisodeUpdateService` is updating podcast episodes and notifying the user, you'll finish up by using the Firebase JobDispatcher to schedule the `EpisodeUpdateService`.

Firebase JobDispatcher provides several features to control when jobs are executed. This helps ensure that PodPlay is a good citizen and doesn't drain battery unnecessarily or adversely impact the performance of other applications.

Besides controlling the interval that your job should execute, you can place other constraints on when the job should execute. These constraints include network, charging state and idle state.

For example, with the network type, you can request the job only runs if the network is unmetered (i.e., not on a cell network). You can combine multiple constraints.

An excellent place to configure and start the `JobDispatcher` is in the main podcast activity.

First, you'll need a new constant to define the job tag.

Open **ui\PodcastActivity.kt** and add the following line to the companion object:

```kotlin
private val TAG_EPISODE_UPDATE_JOB = "com.raywenderlich.podplay.episodes"
```

Add the following method:

```kotlin
private fun scheduleJobs() {
    // 1
    val dispatcher = FirebaseJobDispatcher(GooglePlayDriver(this))
    // 2
    val oneHourInSeconds = 60*60
    val tenMinutesInSeconds = 60*10
    val episodeUpdateJob = dispatcher.newJobBuilder()
        .setService(EpisodeUpdateService::class.java)
        .setTag(TAG_EPISODE_UPDATE_JOB)
        .setRecurring(true)
        .setTrigger(Trigger.executionWindow(oneHourInSeconds,
          (oneHourInSeconds + tenMinutesInSeconds)))
        .setLifetime(Lifetime.FOREVER)
        .setConstraints(
            Constraint.ON_UNMETERED_NETWORK,
            Constraint.DEVICE_CHARGING
        )
        .build()

    dispatcher.mustSchedule(episodeUpdateJob)
}
```

That's all you need to kick off a job with `FirebaseJobDispatcher`.

1. Instantiate the `FirebaseJobDispatcher` using the `GooglePlayDriver`.
    `FirebaseJobDispatcher` is designed to allow different drivers to be swapped in to
    control the low-level job scheduling. The only driver currently available is the
    `GooglePlayDriver`.

2. Create a new job builder and use it to build the `episodeUpdateJob`. The following
    parameters are set on the job:

    setService: Tells the job to use the `EpisodeUpdateService` service when it's time to
    run the job.

    setTag: Sets a unique tag to identify the job. You can use this tag to cancel the job.

    setRecurring: Setting this to `true` will cause the job to repeat.

    setTrigger: This controls how often the job repeats. `Trigger.executionWindow`
    defines the earliest time and the latest time the job should start from the last time
    it was executed. The times are in seconds and are defined as 1 hour and 1 hour 10
    minutes. Keep in mind that even with this window, the job still has to meet all
    constraints before it's executed.

    setLifetime: This tells the job to work even when the device is rebooted. If you want
    the job to end when the device is rebooted, you can pass in
    `Lifetime.UNTIL_NEXT_BOOT`.

    setConstraints: The job is set to only run on an unmetered network and only when
    the device is plugged in.

> **Note**: You'll need to remove the unmetered network setting if you want to test
> on an emulator.

Because the job should continue to execute even if the device is rebooted, the
application must specify one additional permission in the manifest file.

Open **AndroidManifest.xml** and add the following permission line just above the
opening tag of `application`:

```
<uses-permission
android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
```

Now you just need to call `scheduleJobs()` when the activity is started. Go back to **PodcastActivity.kt** and add the following line to the end of `onCreate()`:

```
scheduleJobs()
```

Notification Intent

At this point, the episode job will run, and the notifications will work. If the user taps the notification, it will activate the `PodcastActivity`. The only thing left is to handle the notification intent and use it to display the podcast details.

Currently, the only time the app navigates to the podcast details screen is when the user taps a podcast. When this happens, the podcast is made active in the view model and `onShowDetails()` is called. You'll simulate this same behavior when the notification intent is received.

First, you need a new method in the podcast view model to set the activate podcast based on a feed URL.

Open **viewmodel\PodcastViewModel.kt** and add the following method:

```
fun setActivePodcast(feedUrl: String,
    callback: (PodcastSummaryViewData?) -> Unit) {

  val repo = podcastRepo ?: return

  repo.getPodcast(feedUrl, { podcast ->
    if (podcast == null) {
      callback(null)
    } else {
      activePodcastViewData = podcastToPodcastView(podcast)
      activePodcast = podcast
      callback(podcastToSummaryView(podcast))
    }
  })
}
```

This method loads the podcast from the database based on the `feedUrl`. If the podcast is found, it's converted to a podcast view and set as the active podcast. The podcast summary view data is then passed to the callback.

Now you can look for the intent data in the podcast activity and use it to set the active podcast and display the details screen.

Open **PodcastActivity.kt** and add the following to the end of `handleIntent()`:

```
val podcastFeedUrl = intent
    .getStringExtra(EpisodeUpdateService.EXTRA_FEED_URL)
if (podcastFeedUrl != null) {
```

```
    podcastViewModel.setActivePodcast(podcastFeedUrl, {
      it?.let { podcastSummaryView -> onShowDetails(podcastSummaryView) }
    })
  }
```

The `podcastFeedUrl` is extracted from the intent. If it's not `null`, then `setActivePodcast()` is called on the view model. After it retrieves the podcast, `setActivePodcast()` executes the callback and passes in the `podcastSummaryView` object. Finally, `onShowDetails()` is called with the `podcastSummaryView` to display the podcast podcast details screen.

Build and run the app.

You may find it a little difficult to test the new features. You'll only see evidence that it's working when one of your subscribed podcasts are updated with new episodes, and this may not happen for days depending on the frequency of the podcast releases.

One way to force the notification to kick in is to remove a single episode when you subscribe to a podcast. This will result in the initial subscription missing an episode and will cause the podcast update logic to download the missing episode and trigger the notification.

If you want to test with this method, open **PodcastViewModel.kt** and add the following line in `saveActivePodcast()` before the call to `repo.Save()`:

```
  it.episodes = it.episodes.drop(1)
```

This drops the first episode from the Podcast you are subscribing to before it's saved to the database.

You may also want to reduce the execution window times on the job to have the job run without waiting an hour.

> **Note**: Firebase `JobScheduler` may not kick in while the app is being debugged on the device. If you wait the reduced amount of time and nothing happens, try disconnecting your device from the debugger and then running through the subscribe/wait steps again.

Here's the notification area showing two notifications icons for PodPlay:

If you pull down on the notification area you'll see the notification details:
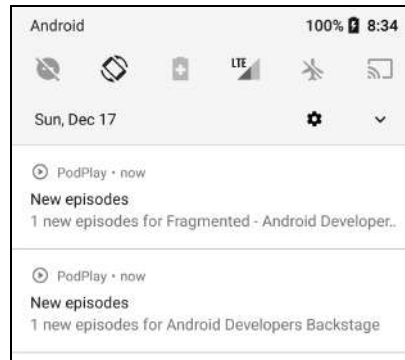
Tap on a notification, and it will launch the podcast details page.

Where to go from here?

After testing, don't forget to remove the temporary code you added to drop the first podcast when subscribing, and put back in the original execution window times.

Congratulations on making it this far! You've completed the main podcast management part of the application. In the next chapter, you'll finally make the PodPlay app live up to its namesake by implementing the media playback interface!

Chapter 26: Podcast Playback

By Tom Blankenship

So far, you've built a decent podcast management app — too bad there's no way to listen to content. Time to fix that!

In this chapter, you'll learn how to build a media player that plays audio and video podcasts, and integrates into the Android ecosystem. Building a good media player takes some work. The payoff, however, is an app that works well in the foreground and also while the user performs other tasks on their device.

Getting started

If you are following along with your own app, the starter project for this chapter includes some additional icons that you'll need to complete the chapter. You can either begin this chapter with the **starter** project or copy all of the drawable resources from the starter project into yours:

- src/main/res/drawable/ic_pause_white.png

- src/main/res/drawable/ic_play_arrow_white.png

- src/main/res/drawable/ic_episode_icon.png

Make sure to copy the files from all of the drawable folders, including everything with the `.hdpi`, `.mdpi`, `.xhdpi`, `.xxhdpi` and `.xxxhdpi` extensions.

Media player basics

> **Note**: The Media classes mentioned here have backward compatible versions that you'll use when building the app. The **Compat** part of the class names have been left out for brevity (i.e., `MediaPlayer` = `MediaPlayerCompat`).

The architecture for an app that requires media playback can be confusing. Getting a birds-eye view of how it works is often the best place to start:
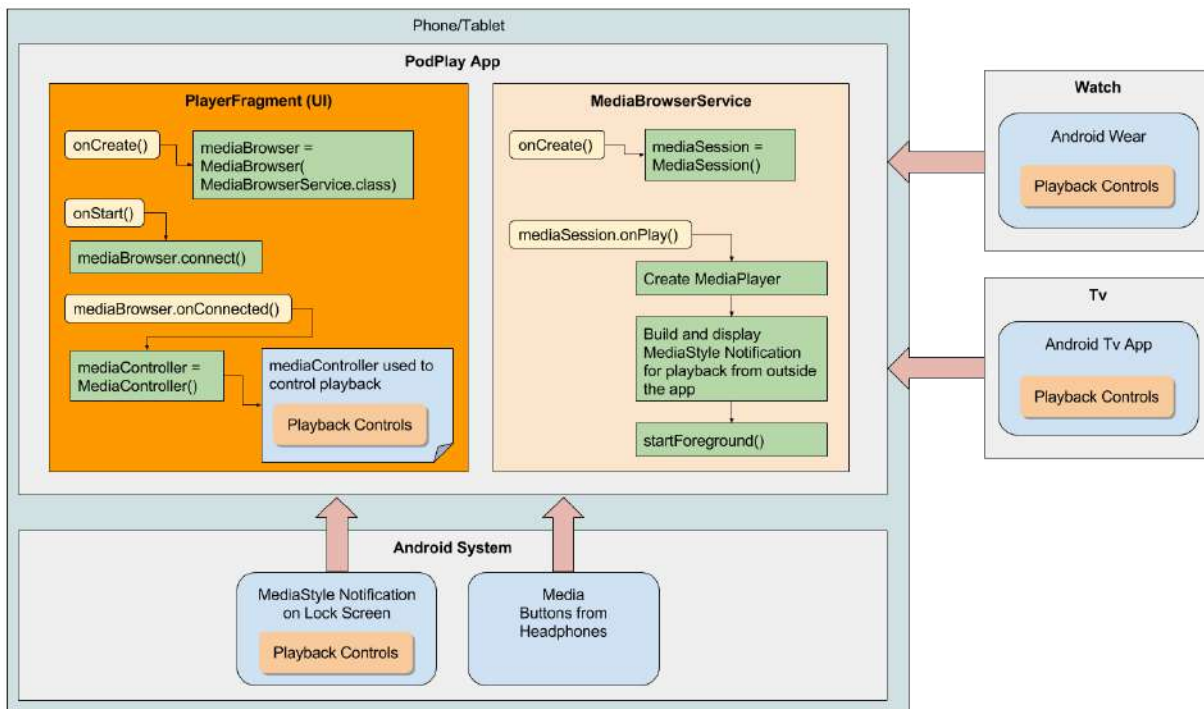
As daunting as this diagram can be, it's meant to show you that adding media playback to an Android app requires two large pieces: the playback UI (`PlayerFragment`) and the playback service (`MediaBrowserService`).

MediaPlayer

The built-in core tool that Android provides for media playback is the **MediaPlayer** class. This class handles both audio and video and can play content stored locally or streamed from an external URL. `MediaPlayer` has standard calls for loading media, starting playback, pausing playback and seeking to a playback position.

MediaSession

Android provides another class named **MediaSession** that is designed to work with any media player, either the built-in `MediaPlayer` or one of your choosing. The `MediaSession` provides callbacks for `onPlay()`, `onPause()` and `onStop()` that you'll use to create and control the media player.

One significant advantage to using a `MediaSession` is that systems other than your application can access it.

MediaController

The **MediaController** is used directly by the user interface, which in turn, communicates with a `MediaSession`, isolating your UI code from the `MediaSession`. `MediaController` provides callbacks for major `MediaSession` events and this can be used to update your UI.

MediaBrowserService

For a better listening experience, you'll let the podcast play in the background and give the user playback controls from outside the PodPlay app. There are many ways a user may want to control audio from outside an application, and **MediaBrowserService** makes it possible.

`MediaBrowserService` runs as foreground service when playing audio. When a service is running in foreground mode, Android makes sure it sticks around.

With other background services, Android tends to kill them off — which isn't something you want when the user is listening to a long-running podcast.

One central feature of the `MediaBrowserService` is that it's discoverable and other apps can use it to playback your media, which allows advanced features, such as playback from Android Wear or Android Auto devices.

MediaBrowser

To control the `MediaBrowserService` service, you'll use the **MediaBrowser** class. `MediaBrowser` connects to the `MediaBrowserService` service and provides it with a `MediaController`. Your UI will then use a `MediaController` to control the playback operations. Other apps can also use their own `MediaBrowser` to connect to the PodPlay `MediaBrowserService`.

Building the MediaBrowserService

The `MediaBrowserService` is where all the hard work of managing the podcast playback will happen. You'll start with a basic implementation that's just enough to get a podcast playing and then expand the service later.

In the `service` package, create a new file named **PodplayMediaService.kt** with the following contents:

```kotlin
class PodplayMediaService : MediaBrowserServiceCompat() {

  override fun onCreate() {
    super.onCreate()
  }

  override fun onLoadChildren(parentId: String,
      result: Result<MutableList<MediaBrowserCompat.MediaItem>>) {
    // To be implemented
  }

  override fun onGetRoot(clientPackageName: String,
      clientUid: Int, rootHints: Bundle?): BrowserRoot? {
    // To be implemented
    return null
  }
}
```

This represents the basic outline of a `MediaBrowserServiceCompat` class with overloaded methods for `onLoadChildren()` and `onGetRoot()`. We'll come back to these methods later in the chapter.

Just like other services, `PodplayMediaService` needs an entry in the manifest.

Open **AndroidManifest.xml** and add the following under the main `<application>` section:

```xml
<service android:name=".service.PodplayMediaService">
  <intent-filter>
    <action android:name="android.media.browse.MediaBrowserService" />
  </intent-filter>
</service>
```

This allows a `MediaBrowser` to find your media browser service.

Create a MediaSession

At the heart of the `MediaBrowserService` is the `MediaSession`. As PodPlay and other applications interact through `MediaBrowserService`, `MediaSession` responds. But before it can, you need to create the `MediaSession` when the service first starts.

Open **PodplayMediaService.kt** and add the following property:

```
private lateinit var mediaSession: MediaSessionCompat
```

Now add the following method:

```
private fun createMediaSession() {
  // 1
  mediaSession = MediaSessionCompat(this, "PodplayMediaService")
  // 2
  mediaSession.setFlags(MediaSessionCompat.FLAG_HANDLES_MEDIA_BUTTONS or
      MediaSessionCompat.FLAG_HANDLES_TRANSPORT_CONTROLS)
  // 3
  setSessionToken(mediaSession.sessionToken)
  // 4
  // Assign Callback
}
```

Let's walk through the code:

1. The `mediaSession` property is initialized with a new `MediaSessionCompat` object.

2. `setFlags` indicates which actions the media session supports. If you miss this step, the media session won't respond to any events! You'll support media buttons (i.e., play/pause hardware buttons on headphones), and transport controls such as play and pause commands from a media controller. When you build the `MediaBrowser` part of PodPlay, it will use the transport controls.

3. The unique token for the media session is retrieved and applied as the session token on the `PodplayMediaService`, which links the service to the media session.

4. The only missing part is assigning a `Callback` class to the media session. You'll create this next.

To finish out the initialization of the media session, you'll need to define a `MediaSessionCompat.Callback` to handle media events.

In the `service` package, create a new file named **PodplayMediaCallback.kt** and replace its contents with the following:

```
class PodplayMediaCallback(val context: Context,
                           val mediaSession: MediaSessionCompat,
                           var mediaPlayer: MediaPlayer? = null) :
    MediaSessionCompat.Callback() {

  override fun onPlayFromUri(uri: Uri?, extras: Bundle?) {
    super.onPlayFromUri(uri, extras)
    println("Playing ${uri.toString()}")
    onPlay()
  }
```

```
  override fun onPlay() {
    super.onPlay()
    println("onPlay called")
  }

  override fun onStop() {
    super.onStop()
    println("onStop called")
  }

  override fun onPause() {
    super.onPause()
    println("onPause called")
  }
}
```

This is just the skeleton code for the `Callback`; it doesn't do anything yet. Although you can handle other events, these are sufficient for the PodPlay app.

You'll come back to this later, and fill in the details of each callback method. Now, you can finish out the media session initialization.

In **PodplayMediaService.kt**, add the following to the end of `createMediaSession()`:

```
val callBack = PodplayMediaCallback(this, mediaSession)
mediaSession.setCallback(callBack)
```

This creates a new instance of `PodplayMediaCallback` and sets it as the media session callback.

Add the following to the end of `onCreate()`:

```
createMediaSession()
```

Before diving into the detailed implementation on `PodplayMediaService`, let's connect a `MediaBrowser` to the service and test the communication between the browser and service.

Connecting the MediaBrowser

There's no podcast episode player UI in the app yet — which is where you'd normally create the `MediaBrowser` and connect it to the `PodplayMediaService` — so for now, you'll add the `MediaBrowser` code to the podcast details screen instead.

There are four steps to complete when adding `MediaBrowser` capabilities to an activity or fragment:

1. Create the `MediaBrowser` object and connect it to the `MediaBrowserService`.

2. Define a `MediaBrowser.ConnectionCallback` to handle the browser service connection messages.

3. Define a `MediaController.Callback` class to handle data and state changes from the browser service.

4. Connect and disconnect the `MediaBrowser` based on lifecycle events.

Create callbacks

You'll define the callback classes before adding the `MediaBrowser` object.

First, create the `MediaController.Callback` class. This class will receive messages when the playback state changes and is where you would typically update your player UI to reflect the current state.

Open **PodcastDetailsFragment.kt** and add the following inner class:

```kotlin
inner class MediaControllerCallback: MediaControllerCompat.Callback() {
  override fun onMetadataChanged(metadata: MediaMetadataCompat?) {
    println("metadata changed to $
{metadata?.getString(MediaMetadataCompat.METADATA_KEY_MEDIA_URI)}")
  }
  override fun onPlaybackStateChanged(state: PlaybackStateCompat?) {
    println("state changed to $state")
  }
}
```

You haven't implemented a playback UI yet, so the callback methods will just print out information for now.

Now, create the `MediaBrowser.ConnectionCallback` class. This requires a `MediaControllerCallback` object and a `MediaBrowser` object.

Add the following properties to the top of the `PodcastDetailsFragment` class:

```kotlin
private lateinit var mediaBrowser: MediaBrowserCompat
private var mediaControllerCallback: MediaControllerCallback? = null
```

Add the following method:

```kotlin
private fun registerMediaController(token: MediaSessionCompat.Token) {
  // 1
  val mediaController = MediaControllerCompat(activity, token)
  // 2
  MediaControllerCompat.setMediaController(activity, mediaController)
  // 3
  mediaControllerCallback = MediaControllerCallback()
  mediaController.registerCallback(mediaControllerCallback!!)
}
```

1. Create the `MediaController` and associate it with the session token from the `MediaSession` object. This connects the media controller with the media session.

> **Note**: Don't confuse this `MediaController` class with the one from the Android widget library. The `MediaController` widget is designed to provide a basic UI for media playback controls. This `MediaController` class is part of the Android media session package, and it used to communicate with an active media session.

2. Assign the `MediaController` to the activity so that it can be retrieved later with `getMediaController()`.

3. Create a new instance of `MediaControllerCallback` and set it as the callback object for the media controller.

Add the following inner class:

```kotlin
inner class MediaBrowserCallBacks:
  MediaBrowserCompat.ConnectionCallback() {
    // 1
  override fun onConnected() {
    super.onConnected()
    // 2
    registerMediaController(mediaBrowser.sessionToken)
    println("onConnected")
  }

  override fun onConnectionSuspended() {
    super.onConnectionSuspended()
    println("onConnectionSuspended")
    // Disable transport controls
  }

  override fun onConnectionFailed() {
    super.onConnectionFailed()
    println("onConnectionFailed")
    // Fatal error handling
  }
}
```

When you create the media browser object, an instance of `MediaBrowserCallBacks` is passed to the constructor. The `MediaBrowserService` will eventually call `onConnected()` upon successful connection to the `MediaBrowserService`, or it will call `onConnectionFailed()` if there's an issue.

1. `onConnected()` is called after a successful connection. This is your chance to assign a `MediaController` controller to the activity, and to register the `MediaControllerCallback` class with the `mediaController`.

2. The `MediaController` is registered.

Init the MediaBrowser

With the two callback classes created, you're ready to create the media browser object. This asynchronously kicks off the connection to the browser service.

Add the following method:

```
private fun initMediaBrowser() {
  mediaBrowser = MediaBrowserCompat(activity,
      ComponentName(activity, PodplayMediaService::class.java),
      MediaBrowserCallBacks(),
      null)
}
```

Here, you instantiate a new `MediaBrowserCompat` object using the following arguments:

1. **context**: The current activity hosting the fragment.

2. **serviceComponent**: This tells the media browser that it should connect to the `PodplayMediaService` service.

3. **callback**: The callback object to receive connection events.

4. **rootHints**: Optional service specific hints to pass along as a Bundle object.

Now you can call this method when the fragment is created. Add the following line to the end of `onCreate()`:

```
initMediaBrowser()
```

The final step is to connect the media browser and unregister the media controller at the appropriate times.

Connect the MediaBrowser

The media browser should be **connected** when the activity or fragment is **started**. Add the following method:

```
override fun onStart() {
  super.onStart()
    if (mediaBrowser.isConnected) {
      if (MediaControllerCompat.getMediaController(activity) == null) {
        registerMediaController(mediaBrowser.sessionToken)
      }
    } else {
      mediaBrowser.connect()
    }
}
```

First, check to see if the media browser is already connected. This will happen when a configuration change occurs, such as a screen rotation. If it is connected, then all that's needed is to register the media controller. If it's not connected, then you call `connect()`, and delay the media controller registration until the connection is completed.

Unregister the controller

The media controller callbacks should be **unregistered** when the activity or fragment is **stopped**.

Add the following method:

```kotlin
override fun onStop() {
  super.onStop()
  if (MediaControllerCompat.getMediaController(activity) != null) {
    mediaControllerCallback?.let {
      MediaControllerCompat.getMediaController(activity)
          .unregisterCallback(it)
    }
  }
}
```

If the media controller is available and the `mediaControllerCallback` is not `null`, the media controller callbacks object is unregistered.

It's time to make sure everything is connected correctly before adding some playback code.

Build and run the app. Display the details for a podcast, and tap on a single episode.

Look at Logcat. Things didn't go as planned.

There are error messages from the `MediaBrowserService` and the `MediaBrowser`, and `onConnectionFailed()` was called on your `MediaBrowserCallBacks` object.

```
I/MediaBrowserService: No root for client com.raywenderlich.podplay from
service android.service.media.MediaBrowserService$ServiceBinder$1
E/MediaBrowser: onConnectFailed for
ComponentInfo{com.raywenderlich.podplay/
com.raywenderlich.podplay.service.PodplayMediaService}
I/System.out: onConnectionFailed
```

Handle media browsing

To properly handle media browsing, there's one part of `PodplayMediaService` you need to complete.

onGetRoot() and onLoadChildren() are designed to work in concert and provide a hierarchy of media content to a media browser. A media browser will call these two methods to get a list of browsable menu items to show the user.

onGetRoot() should return the root media ID of the content tree. onLoadChildren() should return the list of child media items given a parent media ID. If onGetRoot() returns null then the connection fails.

Media browsing is an optional feature, and a media browser can still connect to and control a media service without full media browsing capabilities. PodPlay will not allow media browsing, but you still need to return an empty root ID from onGetRoot().

Define a new media ID representing the empty root media and return it in onGetRoot().

Open **PodplayMediaService.kt** and add the following companion object:

```
companion object {
  private const val PODPLAY_EMPTY_ROOT_MEDIA_ID =
      "podplay_empty_root_media_id"
}
```

Replace the contents of onGetRoot() with the following:

```
return MediaBrowserServiceCompat.BrowserRoot(
    PODPLAY_EMPTY_ROOT_MEDIA_ID, null)
```

Now, tell onLoadChildren() to return an empty list of children for the empty root ID.

Replace the contents of onLoadChildren() with the following:

```
if (parentId.equals(PODPLAY_EMPTY_ROOT_MEDIA_ID)) {
  result.sendResult(null)
}
```

Build and run the app. Display the details for a podcast, and tap on a single episode again.

Look at Logcat, and you'll see the onConnected message indicating the media browser connected to the media browser service without any problems.

```
I/System.out: onConnected
```

Sending playback commands

With the successful connection in place, it's time to test out the ability to send play commands and recognize state changes.

For now, to keep things simple, you'll send a play command to the
`PodplayMediaService` when the user taps on a podcast episode.

Start by adding some code to detect when the user taps on an episode.

Open **EpisodeListAdapter.kt** and add the following to the top of the class:

```
interface EpisodeListAdapterListener {
  fun onSelectedEpisode(episodeViewData: EpisodeViewData)
}
```

`PodcastDetailsFragment` will implement this interface and be notified when the user
taps an episode.

Update the `EpisodeListAdapter` definition to match the following:

```
class EpisodeListAdapter(
    private var episodeViewList: List<EpisodeViewData>?,
    private val episodeListAdapterListener: EpisodeListAdapterListener) :
    RecyclerView.Adapter<EpisodeListAdapter.ViewHolder>() {
```

This adds in the `episodeListAdapterListener` argument to the constructor.

Update the ViewHolder definition to the following:

```
class ViewHolder(
    v: View, private
    val episodeListAdapterListener: EpisodeListAdapterListener) :
    RecyclerView.ViewHolder(v) {
```

This adds the **episodeListAdapterListener** argument to the class declaration.

Update the return in `onCreateViewHolder()` to add in the new argument:

```
return ViewHolder(LayoutInflater.from(parent.context)
    .inflate(R.layout.episode_item, parent, false),
        episodeListAdapterListener)
```

Add the following method to the `ViewHolder` class:

```
init {
  v.setOnClickListener {
    episodeViewData?.let {
      episodeListAdapterListener.onSelectedEpisode(it)
    }
  }
}
```

You set an `onClickListener` on the view holder. When the user taps an episode,
`onSelectedEpisode()` is called on the adapter listener.

That's it! `EpisodeListAdapter` will now call `onSelectedEpisode()` when the user taps an episode.

From here, you can make `PodcastDetailsFragment` implement the `episodeListAdapterListener` interface. First, define a method to start the playback from an `EpisodeViewData` item.

Open **PodcastDetailsFragment.kt** and add the following method:

```kotlin
private fun startPlaying(
    episodeViewData: PodcastViewModel.EpisodeViewData) {
  val controller = MediaControllerCompat.getMediaController(activity)
  controller.transportControls.playFromUri(
      Uri.parse(episodeViewData.mediaUrl), null)
}
```

This method takes a single `EpisodeViewData` item and uses the media controller transport controls to initiate the media playback. The call to `playFromUri()` triggers the `onPlayFromUri()` callback in `PodplayMediaService`.

Next, implement the `episodeListAdapterListener` interface in `PodcastDetailsFragment`.

Update the `PodcastDetailsFragment` class definition as follows:

```kotlin
class PodcastDetailsFragment : Fragment(), EpisodeListAdapterListener {
```

Add the following method to implement the `onSelectedEpisode` logic:

```kotlin
override fun onSelectedEpisode(episodeViewData: EpisodeViewData) {
  // 1
  var controller = MediaControllerCompat.getMediaController(activity)
  // 2
  if (controller.playbackState != null) {
    if (controller.playbackState.state ==
        PlaybackStateCompat.STATE_PLAYING) {
      // 3
     controller.transportControls.pause()
    } else {
      // 4
      startPlaying(episodeViewData)
    }
  } else {
    // 5
    startPlaying(episodeViewData)
  }
}
```

This is called when the user taps an episode. Use this opportunity to either play or pause the current episode depending on the current playback state.

Let's go over things in detail:

1. You get the media controller that was previously assigned to the activity.

2. If the playback state is not `null`, then the state is checked.

3. If the playback state is "playing", then you pause the episode using the transport controls.

4. If the playback state is "paused", then you call `startPlaying()` to play the episode.

5. If the playback state is `null,` then you call `startPlaying()` to play the episode.

In `setupControls()`, update the call to `EpisodeListAdapter()` to pass in the `EpisodeListAdapterListener` argument:

```
episodeListAdapter =
    EpisodeListAdapter(podcastViewModel.activePodcastViewData?.episodes,
        this)
```

Updating media session state

Finally, update the media service to set the playback states based on the incoming play commands.

Open **PodplayMediaCallback.kt** and add the following method to the class:

```
private fun setState(state: Int) {
  var position: Long = -1

  val playbackState = PlaybackStateCompat.Builder()
  .setActions(
      PlaybackStateCompat.ACTION_PLAY or
          PlaybackStateCompat.ACTION_STOP or
          PlaybackStateCompat.ACTION_PLAY_PAUSE or
          PlaybackStateCompat.ACTION_PAUSE)
  .setState(state, position, 1.0f)
  .build()

  mediaSession.setPlaybackState(playbackState)
}
```

This is a helper method to set the current state on the media session. The media session state is configured with a `PlaybackState` object that provides a `Builder` to set all of the options. This takes a simple playback `state` such as `STATE_PLAYING` and uses it to construct the more complex `PlaybackState` object. `setActions()` specifies what states the media session will allow.

Now you can use this method to update the state as playback commands are processed.

Add the following line to the end of `onPlayFromUri()`:

```
mediaSession.setMetadata(MediaMetadataCompat.Builder()
    .putString(MediaMetadataCompat.METADATA_KEY_MEDIA_URI,
        uri.toString())
    .build())
```

Metadata is set on the `mediaSession` object to use the `METADATA_KEY_MEDIA_URI` key. You can set a variety of metadata on the media session — more will be added later. This data is used by media browsers to display details about the audio track being played.

Add the following line to the end of `onPlay()`:

```
setState(PlaybackStateCompat.STATE_PLAYING)
```

When receiving the play command, the media session playback state is set to `STATE_PLAYING`.

Add the following line to the end of `onPause()`:

```
setState(PlaybackStateCompat.STATE_PAUSED)
```

When receiving the pause command, the media session playback state is set to `STATE_PAUSED`.

You aren't playing or pausing anything yet, but at least the state is set correctly!

Build and run the app. Once again, display the details for a podcast, then tap on a single episode and then tap on it again.

You'll see the following output in Logcat showing that the `onPlay` and `onPause` methods are getting called in the media service, and the state changes are getting picked up by the media controller callbacks.

```
I/System.out: onConnected
I/System.out: onPlayFromUri https://audio.simplecast.com/2be4cd5d.mp3
I/System.out: onPlay
I/System.out: metadata changed to https://audio.simplecast.com/
2be4cd5d.mp3
I/System.out: state changed to PlaybackState {state=3, position=0,
buffered position=0, speed=1.0, updated=71964629, actions=519, error
code=0, error message=null, custom actions=[], active item id=-1}
I/System.out: onPause
I/System.out: state changed to PlaybackState {state=2, position=0,
buffered position=0, speed=1.0, updated=71975052, actions=519, error
code=0, error message=null, custom actions=[], active item id=-1}
```

Using MediaPlayer

Now that you have the `MediaBrowser` talking to the `MediaBrowserService`, it's time to hear some audio!

It's up to you to provide the media playback capabilities in response to the media session events. You can use any means you want — including third-party media players — to play back the media.

For PodPlay, Android's built-in `MediaPlayer` will do the job. In this section, after creating the `MediaPlayer`, you'll add a few helper methods to control playback.

To begin using `MediaPlayer`, you'll initialize it when playback is first requested for a given media item.

You'll store the most recently requested media item and keep track of whether the item is new or not.

Add the following properties to the `PodplayMediaCallback` class:

```
private var mediaUri: Uri? = null
private var newMedia: Boolean = false
private var mediaExtras: Bundle? = null
```

`mediaUri` will keep track of the currently playing media item, and `newMedia` will indicate if it's a new item. `mediaExtras` will keep track of the media information passed into `onPlayFromUri()`.

First, create a method to store a new media item and set the metadata on the media session.

Add the following method:

```
private fun setNewMedia(uri: Uri?) {
  newMedia = true
  mediaUri = uri
}
```

This sets the `newMedia` flag to true, and stores the current media in `mediaUri`.

Next, create a method to grab audio focus.

Add the following method:

```
private fun ensureAudioFocus(): Boolean {
  val audioManager = this.context.getSystemService(
      Context.AUDIO_SERVICE) as AudioManager
  val result = audioManager.requestAudioFocus(null,
      AudioManager.STREAM_MUSIC,
```

```
        AudioManager.AUDIOFOCUS_GAIN)
    return result == AudioManager.AUDIOFOCUS_REQUEST_GRANTED
}
```

Android uses the concept of audio focus to make sure that apps cooperate with each other and the system, ensuring that audio is played at the appropriate times. Only one app has audio focus at a time, although more than one app can play audio at the same time.

For instance, if you have a navigation app running that needs to announce an upcoming turn, it will request audio focus. If another app, such as PodPlay is playing a podcast, it will receive notification that it should pause or lower the volume while the navigation instructions are announced.

`ensureAudioFocus()` asks the system to give the PodPlay app audio focus. It returns `true` if the focus was granted, or `false` otherwise.

Update `onPlay()` to surround the code with a call to `ensureAudioFocus()` as follows:

```
if (ensureAudioFocus()) {
  mediaSession.isActive = true
  initializeMediaPlayer()
  prepareMedia()
  startPlaying()
}
```

You'll also need a method to give up audio focus. Add the following method:

```
private fun removeAudioFocus() {
  val audioManager = this.context.getSystemService(
      Context.AUDIO_SERVICE) as AudioManager
  audioManager.abandonAudioFocus(null)
}
```

Now, create a method to initialize the `MediaPlayer`.

Add the following method:

```
private fun initializeMediaPlayer() {
  if (mediaPlayer == null) {
    mediaPlayer = MediaPlayer()
    mediaPlayer!!.setOnCompletionListener({
      setState(PlaybackStateCompat.STATE_PAUSED)
    })
  }
}
```

This creates a new instance of the `MediaPlayer` if it doesn't already exist. It also sets up a listener for when playback completes and pauses the player upon completion.

Remove the call to `mediaSession.setMetadata` from `onPlayFromUri()` since it will be called here instead.

Create a method to prepare the media for the `MediaPlayer`.

Add the following method to `PodplayMediaService`:

```
private fun prepareMedia() {
  if (newMedia == true) {
    newMedia = false
    mediaPlayer?.let { mediaPlayer ->
      mediaUri?.let {
        mediaPlayer.reset()
        mediaPlayer.setDataSource(context, mediaUri)
        mediaPlayer.prepare()
        mediaSession.setMetadata(MediaMetadataCompat.Builder()
          .putString(MediaMetadataCompat.METADATA_KEY_MEDIA_URI,
              mediaUri.toString())
          .build())
      }
    }
  }
}
```

If it's a new media item and the media player and media URI are valid, the media player state is reset, and the data source is set to the media item. Once the data source is set, then prepare is called. `prepare()` puts the `MediaPlayer` in an initialized state ready to play the media provided as the data source.

Previously, the `setState()` you defined assigned a playback position of –1. Now that you have a media player, this can be updated to grab the actual position from the player.

Add the following after the `var position: Long = -1` line in `setState()`:

```
mediaPlayer?.let {
  position = it.getCurrentPosition().toLong()
}
```

Add the following method to start the playback of the audio media.

```
private fun startPlaying() {
  mediaPlayer?.let { mediaPlayer ->
    if (!mediaPlayer.isPlaying()) {
      mediaPlayer.start()
      setState(PlaybackStateCompat.STATE_PLAYING)
    }
  }
}
```

If the `mediaPlayer` is not `null` and it's not already playing, then it is instructed to play the media. The media session state is updated to `STATE_PLAYING`.

Add the following method to pause playback of the audio media.

```kotlin
private fun pausePlaying() {
  removeAudioFocus()
  mediaPlayer?.let { mediaPlayer ->
    if (mediaPlayer.isPlaying) {
      mediaPlayer.pause()
      setState(PlaybackStateCompat.STATE_PAUSED)
    }
  }
}
```

Start by removing the audio focus from the app. If the `mediaPlayer` is not `null` and it's already playing, then it's instructed to pause the media. The media session state is updated to `STATE_PAUSED`.

Finally, you need to handle the case where playback is stopped.

Add the following method to `PodplayMediaCallback`:

```kotlin
private fun stopPlaying() {
  removeAudioFocus()
  mediaSession.isActive = false
  mediaPlayer?.let { mediaPlayer ->
    if (mediaPlayer.isPlaying) {
      mediaPlayer.stop()
      setState(PlaybackStateCompat.STATE_STOPPED)
    }
  }
}
```

This is similar to `pausePlaying()`, but it sets the media session to inactive and the state to `STATE_STOPPED`.

That's all of the supporting methods; now you just need to call them at the appropriate times.

Add the following lines before the call to `onPlay()` in `onPlayFromUri()`:

```kotlin
if (mediaUri == uri) {
  newMedia = false
  mediaExtras = null
} else {
  mediaExtras = extras
  setNewMedia(uri)
}
```

If the `uri` passed in is the same as before, then the `newMedia` flag is set to false, and mediaExtras is set to null. There is no need to set the new media or `mediaExtras` if a new media item is not being set. If the `uri` is new, then the media extras are stored and `setNewMedia()` is called.

Replace the call to `setState()` in `onPlay()` with the following lines:

```
initializeMediaPlayer()
prepareMedia()
startPlaying()
```

The media player is initialized, the media is prepared for playback, and then the media player is told to start playing.

Replace the call to `setState()` in `onPause()` with the following:

```
pausePlaying()
```

Call `stopPlaying()` when the event comes in. Add the following line to the end of `onStop()`:

```
stopPlaying()
```

Build and run the app.

Display the details for a podcast and tap on an episode. Make sure your audio is turned up on your device or emulator. The episode should start streaming within a few seconds.

> **Note**: If you don't hear any sound and are running on Emulator, check your computer's default sound output. Also, check Logcat and if you see playback errors, try restarting both the Emulator and Android Studio, then retry.

Tap the episode again, and the playback will pause. Tap the same episode and playback will begin again where it left off.

Congratulations, you're finally able to listen to a podcast!

Now that basic playback is working, it's time to take the service to the next level and make it a true foreground service. As it stands now, the service runs in the background and is likely to get killed by Android at any time. It will also get shut down if you close the PodPlay app.

Foreground service

To keep the audio playing, `PodplayMediaService` will be set as a foreground service. Any foreground service requires that it display a visible notification to the user. This will be done at the time the podcast begins playing.

Media notification

To display the notification, you'll build it using the same APIs as you did the new episode notification in the last chapter, but this time the expanded notification will display playback controls. You'll use a special style named **MediaStyle** on the notification that automatically displays and handles the playback controls.

Two possible actions will be assigned to the notification. A play action is used if the media is not currently playing. A pause action is used if the media is currently playing. Whenever the media playback state changes, the notification will be replaced and the appropriate action assigned.

Start by creating the two possible notification actions:

Open **PodplayMediaService.kt** and add the following method:

```
private fun getPausePlayActions():
    Pair<NotificationCompat.Action, NotificationCompat.Action>  {
  val pauseAction = NotificationCompat.Action(
      R.drawable.ic_pause_white, getString(R.string.pause),
      MediaButtonReceiver.buildMediaButtonPendingIntent(this,
          PlaybackStateCompat.ACTION_PAUSE))

  val playAction = NotificationCompat.Action(
      R.drawable.ic_play_arrow_white, getString(R.string.play),
      MediaButtonReceiver.buildMediaButtonPendingIntent(this,
          PlaybackStateCompat.ACTION_PLAY))

  return Pair(pauseAction, playAction)
}
```

> **Note**: Make sure to choose `android.support.v4.app.NotificationCompat` for the `NotificationCompat` import.

Pause and play actions are created and returned to the caller. Each action has an associated icon, title, and pending intent. `buildMediaButtonPendingIntent()` creates a pending intent that triggers a playback action on the media service.

Add the following strings to the **strings.xml** file:

```
<string name="pause">Pause</string>
<string name="play">Play</string>
```

To decide whether to use the pause or play action, you'll need a method to determine if the `MediaPlayer` is currently playing media.

Add the following method:

```kotlin
private fun isPlaying(): Boolean {
  if (mediaSession.controller.playbackState != null) {
    return mediaSession.controller.playbackState.state ==
        PlaybackStateCompat.STATE_PLAYING
  } else {
    return false
  }
}
```

This checks the current playback state and returns `true` if it is playing.

The `Notification` also needs a pending intent to launch the main `PodcastActivity` when the notification is tapped.

Add the following method:

```kotlin
private fun getNotificationIntent(): PendingIntent {
  val openActivityIntent = Intent(this, PodcastActivity::class.java)
  openActivityIntent.setFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP)
  return PendingIntent.getActivity(
      this@PodplayMediaService, 0, openActivityIntent,
      PendingIntent.FLAG_CANCEL_CURRENT)
}
```

This creates a pending intent that will open the `PodcastActivity`.

Notifications also require a channel. Create a new channel ID and a method to create the channel.

Add the following line to the companion object in `PodplayMediaService`:

```kotlin
private const val PLAYER_CHANNEL_ID = "podplay_player_channel"
```

Add the following method:

```kotlin
@RequiresApi(Build.VERSION_CODES.O)
private fun createNotificationChannel()
{
  val notificationManager =
      getSystemService(Context.NOTIFICATION_SERVICE)
        as NotificationManager
  if (notificationManager.getNotificationChannel(PLAYER_CHANNEL_ID)
      == null) {
    val channel = NotificationChannel(PLAYER_CHANNEL_ID, "Player",
        NotificationManager.IMPORTANCE_LOW)
    notificationManager.createNotificationChannel(channel)
  }
}
```

This is similar to the channel you created for the episode update notification in the last chapter. The only difference is the channel ID.

Now you can build out the notification. Add the following method:

```
// 1
private fun createNotification(mediaDescription: MediaDescriptionCompat,
                                bitmap: Bitmap?): Notification {
  // 2
  val notificationIntent = getNotificationIntent()
  // 3
  val (pauseAction, playAction) = getPausePlayActions()
  // 4
  val notification = NotificationCompat.Builder(
      this@PodplayMediaService, PLAYER_CHANNEL_ID)
  // 5
  notification
      .setContentTitle(mediaDescription.title)
      .setContentText(mediaDescription.subtitle)
      .setLargeIcon(bitmap)
      .setContentIntent(notificationIntent)
      .setDeleteIntent(
          MediaButtonReceiver.buildMediaButtonPendingIntent(this,
          PlaybackStateCompat.ACTION_STOP))
      .setVisibility(NotificationCompat.VISIBILITY_PUBLIC)
      .setSmallIcon(R.drawable.ic_episode_icon)
      .addAction(if (isPlaying()) pauseAction else playAction)
      .setStyle(
          android.support.v4.media.app.NotificationCompat.MediaStyle()
          .setMediaSession(mediaSession.sessionToken)
          .setShowActionsInCompactView(0)
          .setShowCancelButton(true)
          .setCancelButtonIntent(
              MediaButtonReceiver.buildMediaButtonPendingIntent(this,
              PlaybackStateCompat.ACTION_STOP)))

  // 6
  return notification.build()
}
```

Let's go over this in detail:

1. The method accepts a `MediaDescriptionCompat` object and a bitmap. These contain all the details required to construct the notification.

2. The main notification intent is created. This will be set as the content intent on the notification. This is what allows the `PodcastActivity` to launch when the notification is tapped.

3. The pause and play actions are created.

4. The notification builder is created using the player channel ID.

5. The builder is used to create the details of the notification.

    setContentTitle: Sets the main title on the notification from the media description title.

setContentText: Sets the content text on the notification from the media description subtitle.

setLargeIcon: Sets the icon (album art) to display on the notification.

setContentIntent: Set the content intent, so PodPlay is launched when the notification is tapped.

setDeleteIntent: Send an `ACTION_STOP` command to the service if the user swipes away the notification.

setVisibility: Make sure the transport controls are visible on the lock screen.

setSmallIcon: Set the icon to display in the status bar.

addAction: Add either the play or pause action based on the current playback state.

setStyle: Uses the special `MediaStyle`` to create a style that is designed to display up to five transport control buttons in the expanded view.

The following items are used to control how the `MediaStyle` behaves:

setStyle.setMediaSession: Indicates that this is an active media session. The system uses this as a flag to activate special features such as showing album artwork and playback controls on the lock screen.

setStyle.setShowActionsInCompactView: Indicates which action buttons to display in compact view mode. This takes up to three index numbers to specify the order of the controls.

setStyle.setShowCancelButton: Displays a cancel button on versions of Android before Lollipop (API 21).

setStyle.setCancelButtonIntent(): Pending intent to use when the cancel button is tapped.

6. The notification is built and returned to the caller.

Now let's tie this all together and create a method to display the notification.

First, you'll need a unique notification ID when starting the foreground service.

Add the following to the companion object:

```
private const val NOTIFICATION_ID = 1
```

Add the following method to `PodplayMediaCallback`:

```kotlin
private fun displayNotification() {
  // 1
  if (mediaSession.controller.metadata == null) {
    return
  }
  // 2
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    createNotificationChannel()
  }
  // 3
  val mediaDescription = mediaSession.controller.metadata.description
  // 4
  Glide.with(this)
      .asBitmap()
      .load(mediaDescription.iconUri)
      .into(object : SimpleTarget<Bitmap>() {
        // 5
        override fun onResourceReady(resource: Bitmap?,
            transition: Transition<in Bitmap>?) {
          // 6
          val notification = createNotification(mediaDescription,
              resource)
          // 7
          ContextCompat.startForegroundService(
              this@PodplayMediaService,
              Intent(this@PodplayMediaService,
                PodplayMediaService::class.java))
          // 8
          startForeground(NOTIFICATION_ID, notification)
        }
      })
}
```

> **Note**: Make sure to choose `com.bumptech.glide.request.transition.Transition` as the `Transition` import.

1. If there is no `metadata` on the `mediaSession.controller`, then the method is abandoned.

2. Android O or newer requires a notification channel.

3. The `MediaDescription` is extracted from the media session.

4. `Glide` is used to load in the album artwork. You've used `Glide` before to load an image directly into an `ImageView`. This is a slightly different use case where you define your own target for the image in the `into()` call.

5. Glide calls `onResourceReady()` on your target when the image has been downloaded or retrieved from the cache.

6. After the image is loaded, you create the notification.

7. `startForegroundService()` starts the service in foreground mode.

8. `startForeground()` displays the notification icon. You pass in a unique notification ID and the notification object.

Now, display the notification when the playback starts or pauses, and hide it when playback stops.

To do this, you need to know when playback has started, and that is handled in the **PodplayMediaCallback** class.

You'll create a listener object on `PodplayMediaCallback` so it can emit some key events to the `MediaBrowserService` class.

> **Note**: You may be wondering why the notification code wasn't included directly in the `PodplayMediaCallback` class instead of setting up the listener and handling it in `MediaBrowserService`. The reason is that `PodplayMediaCallback` will be shared by the video player in the next chapter and notifications are specific to the media browser service implementation.

Open **PodplayMediaCallback.kt** and add the following interface to the class:

```
interface PodplayMediaListener {
   fun onStateChanged()
   fun onStopPlaying()
   fun onPausePlaying()
}
```

Three methods are defined that will be called by `PodplayMediaCallback` in response to key playback events.

First, you'll need a listener property that can be set by the media browser service.

Add the following property to `PodplayMediaCallback`:

```
var listener: PodplayMediaListener? = null
```

Call `onStateChanged()` when the state changes to playing or paused.

Add the following to the end of `setState()`:

```
if (state == PlaybackStateCompat.STATE_PAUSED ||
    state == PlaybackStateCompat.STATE_PLAYING) {
  listener?.onStateChanged()
}
```

Call `onStopPlaying()` when playback stops.

Add the following to the end of `stopPlaying()`:

```
listener?.onStopPlaying()
```

Call `onPausePlaying()` when playback pauses.

Add the following to the end of `pausePlaying()`:

```
listener?.onPausePlaying()
```

Now you can implement `PodplayMediaListener` on the media browser service.

Open **PodplayMediaService.kt** and update the class declaration to the following:

```
class PodplayMediaService : MediaBrowserServiceCompat(),
    PodplayMediaListener {
```

Add the following methods to implement the `PodplayMediaListener` interface:

```
override fun onStateChanged() {
  displayNotification()
}

override fun onStopPlaying() {
  stopSelf()
  stopForeground(true)
}

override fun onPausePlaying() {
  stopForeground(false)
}
```

`onStateChanged()` displays the notification when the state changes between play and paused.

`onStopPlaying()` stops the service and removes it from the foreground. You pass in `true` to remove the notification at the same time. It's important to stop the service when playback stops; otherwise, it will keep running indefinitely.

`onPausePlaying()` removes the service from the foreground but passes in `false`, so the notification is not removed.

Finally, set the listener on the media session callback.

In **PodplayMediaService.kt**, add the following line in `createMediaSession()` before the call to `mediaSession.setCallback()`:

```
callBack.listener = this
```

Media metadata

There's still one missing part. You haven't told the media service about the details of the podcast episode yet. You need to pass in the additional episode details and add them to the media session metadata.

Open **PodcastDetailsFragment.kt** and replace the following line in `startPlaying()`:

```
controller.transportControls.playFromUri(
    Uri.parse(episodeViewData.mediaUrl), null)
```

with this:

```
val viewData = podcastViewModel.activePodcastViewData ?: return
val bundle = Bundle()
bundle.putString(MediaMetadataCompat.METADATA_KEY_TITLE,
    episodeViewData.title)
bundle.putString(MediaMetadataCompat.METADATA_KEY_ARTIST,
    viewData.feedTitle)
bundle.putString(MediaMetadataCompat.METADATA_KEY_ALBUM_ART_URI,
    viewData.imageUrl)

controller.transportControls.playFromUri(
    Uri.parse(episodeViewData.mediaUrl), bundle)
```

This grabs the active podcast data and uses it to create a bundle with some extra information to pass along to the `playFromUri()` call.

It's up to you what keys to use and what information to pass in the Bundle. For consistency, use the same keys here that will be used when setting the metadata on the media session.

Now you can update the media service to read in the values from the bundle and set them as metadata on the media session.

Open **PodplayMediaCallback.kt** and replace the call to `setMetadata` in `prepareMedia()` with the following:

```
mediaExtras?.let { mediaExtras ->
  mediaSession.setMetadata(MediaMetadataCompat.Builder()
  .putString(MediaMetadataCompat.METADATA_KEY_TITLE,
      mediaExtras.getString(MediaMetadataCompat.METADATA_KEY_TITLE))
  .putString(MediaMetadataCompat.METADATA_KEY_ARTIST,
      mediaExtras.getString(MediaMetadataCompat.METADATA_KEY_ARTIST))
  .putString(MediaMetadataCompat.METADATA_KEY_ALBUM_ART_URI,
      mediaExtras.getString(
          MediaMetadataCompat.METADATA_KEY_ALBUM_ART_URI))
  .build())
}
```

This takes the three items set on the Bundle and uses them to set the metadata on the media session. This will be used by the notification and the other media players to display details about the currently playing podcast episode.

Final pieces

One more item is required to stop the playback if the user dismisses the app from the recent applications list. Add the following method to the `PodplayMediaService` class:

```
override fun onTaskRemoved(rootIntent: Intent?) {
  super.onTaskRemoved(rootIntent)
  mediaSession.controller.transportControls.stop()
}
```

`onTaskRemoved()` is called if the user swipes away the app in the recent apps list. This stops the playback and removes the service. This is all you would need if running on API 21 or higher. For versions before API 21, you have to use a built-in broadcast receiver to get button events from the notification.

Add the following to the `<application>` section in **AndroidManifest.xml**:

```
<receiver
android:name="android.support.v4.media.session.MediaButtonReceiver" >
  <intent-filter>
    <action android:name="android.intent.action.MEDIA_BUTTON" />
  </intent-filter>
</receiver>
```

There's one last minor change to help improve the look of the album art when shown in the notification view: update the `iTunesPodcast` model to use a higher resolution version of the album artwork.

Open **PodcastResponse.kt** and rename `artworkUrl30` to `artworkUrl100` in the `iTunesPodcast` class as follows:

```
val artworkUrl100: String,
```

Open **SearchViewModel.kt** and replace `artworkUrl30` with `artworkUrl100` in `itunesPodcastToPodcastSummaryView()`:

```
itunesPodcast.artworkUrl100,
```

Build and run the app. Once again, display the details for a podcast, and tap on an episode to start it playing. This time, a notification icon will display in the status bar. Pull down the notification to reveal the expanded view. Tap on the pause button to stop the playback.
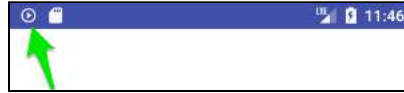

Depending on the version of Android you're running, the notification will display with a different style. Notice on Android Oreo that the notification takes on a tint color based on the album artwork.
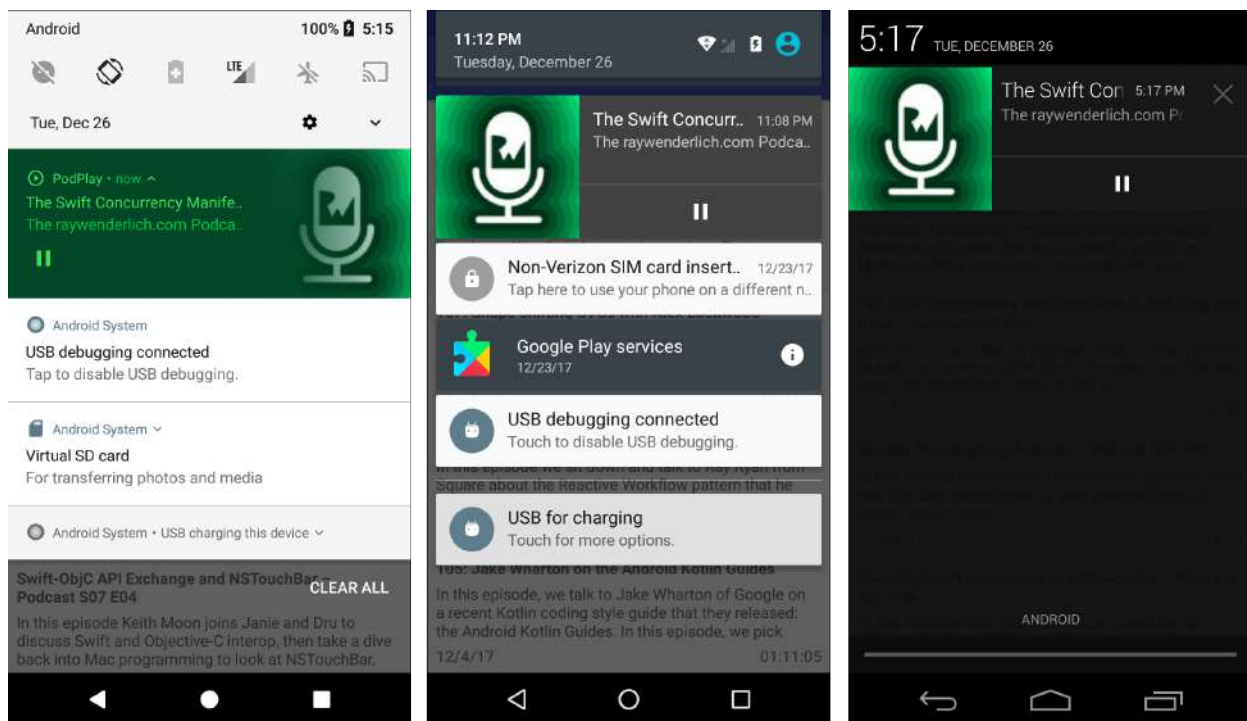
From left to right, Android Oreo (8), Android Marshmallow (6), Android Lollipop (5)

Exit out of the PodPlay app. The podcast will keep playing, and you can still control it from the notification view.

Turn off the phone and display the lock screen. The notification will show in the lock screen, allowing you to control the playback.
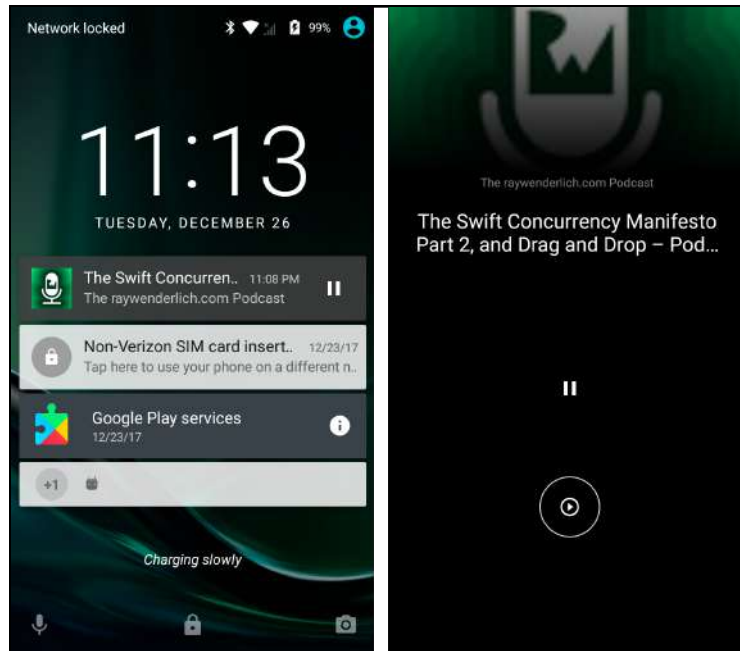
Android Marshmallow Lockscreens

If you have an Android Wear watch that's connected to your device, it will display a media playback screen allowing you to control the playback from the watch.
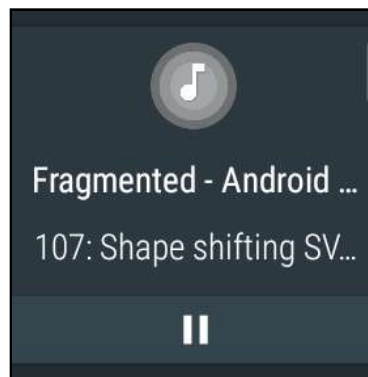
Android Wear

Where to go from here?

That was a lot of work to get playback working, but it's worth it to have podcasts that play properly in the background! Take a break and find a relaxing podcast to listen to while you get ready for the next chapter.

In the final chapter of this section, you'll wrap up the PodPlay app by building a full episode details screen with playback controls. Plus, you'll add a few more finishing touches.

Chapter 27: Episode Player

By Tom Blankenship

In the last chapter, you succeeded in adding audio playback to the app, but you stopped short of adding any built-in playback features. In this final chapter of the book, you'll finish up the PodPlay app by adding a full playback interface and support for videos.

If you are following along with your own app, the starter project for this chapter includes additional resources you'll need to complete the section. You can either begin this chapter with the **starter** project or copy the following drawable resources from the starter project into yours. Make sure to copy the .png files from all of the various dpi folders.

- res/drawable-?dpi/ic_forward_30_white.png

- res/drawable-?dpi/ic_replay_10_white.png

- res/drawable/ic_play_pause_toggle.xml

Getting started

You'll start by adding a new fragment to display the details for a single episode. This fragment gets loaded when the user taps on an episode.

The episode detail screen will provide an overview of the episode and playback controls. The design will look like this:
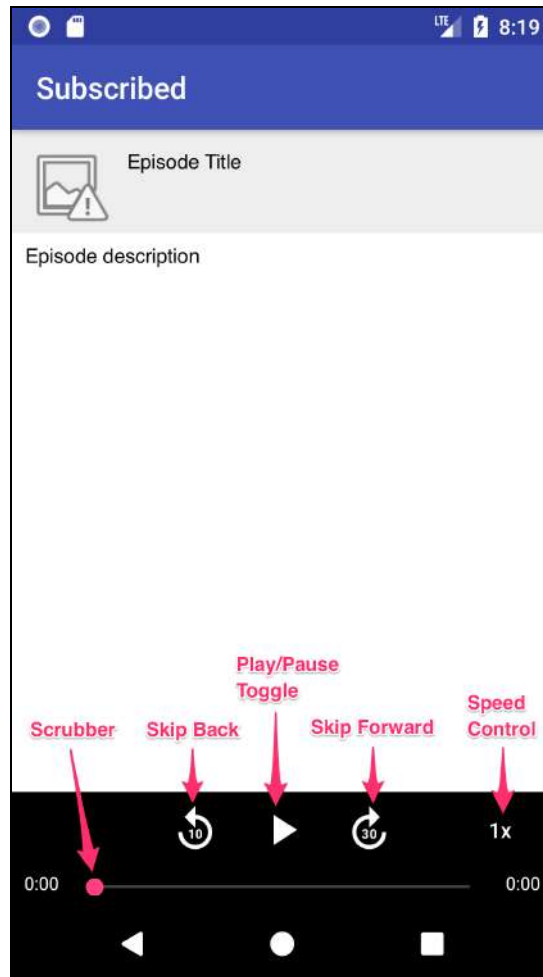
The album art is in the upper-left corner. The episode title is to the right. The description takes up the entire center of the layout. Because episode descriptions can be long, the `TextView` is scrollable so the user can see the full description.

At the bottom is the player controls area. This area has a black background and the following controls:

- **Play/Pause toggle**: Starts and stops playback.

- **Skip back**: Skips back 10 seconds.

- **Skip forward**: Skips forward 30 seconds.

- **Speed control**: Allows the playback speed to be increased.

- **Scrubber**: Displays playback progress and allows scrubbing to any part of the episode.

You'll start by creating the basic layout.

Episode player layout

In the **res/layout** folder, create a new file named **fragment_episode_player.xml**, and replace the contents with the following:

```xml
<android.support.constraint.ConstraintLayout
  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:layout_width="match_parent"
  android:layout_height="match_parent"
  android:background="@android:color/black"
  tools:context="com.raywenderlich.podplay.ui.EpisodePlayerFragment">

  <SurfaceView
      android:id="@+id/videoSurfaceView"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      app:layout_constraintEnd_toEndOf="parent"
      app:layout_constraintStart_toStartOf="parent"
      app:layout_constraintTop_toTopOf="parent"
      android:visibility="invisible"/>

  <android.support.constraint.ConstraintLayout
    android:id="@+id/headerView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="#eeeeee"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent">

  </android.support.constraint.ConstraintLayout>

  <TextView
      android:id="@+id/episodeDescTextView"
      android:layout_width="0dp"
      android:layout_height="0dp"
      android:background="@android:color/white"
      android:padding="8dp"
      android:scrollbars="vertical"
      app:layout_constraintBottom_toTopOf="@+id/playerControls"
      app:layout_constraintEnd_toEndOf="parent"
      app:layout_constraintStart_toStartOf="parent"
      app:layout_constraintTop_toBottomOf="@+id/headerView"
```

```
        tools:text="Episode description"/>

    <android.support.constraint.ConstraintLayout
        android:id="@+id/playerControls"
        android:layout_width="match_parent"
        android:layout_height="76dp"
        android:background="@android:color/background_dark"
        app:layout_constraintBottom_toBottomOf="parent">

    </android.support.constraint.ConstraintLayout>

  </android.support.constraint.ConstraintLayout>
```

This uses `ConstraintLayout` for the main layout, along with an embedded `ConstraintLayout` to contain the `headerView`. There's also an embedded `ConstraintLayout` to contain the `playerControls` area.

Finally, there's a `SurfaceView`, which takes up the entire view and is hidden by default; it's only visible when a video is playing. The player controls will overlay the video.

It's time to add the album art and episode title.

In the `headerView ConstraintLayout` section, add the following:

```
<ImageView
  android:id="@+id/episodeImageView"
  android:layout_width="60dp"
  android:layout_height="60dp"
  android:layout_marginStart="8dp"
  android:layout_marginTop="8dp"
  android:src="@android:drawable/ic_menu_report_image"
  app:layout_constraintStart_toStartOf="parent"
  app:layout_constraintTop_toTopOf="parent"/>

<TextView
    android:id="@+id/episodeTitleTextView"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:text=""
    app:layout_constraintBottom_toBottomOf="@+id/episodeImageView"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/episodeImageView"
    app:layout_constraintTop_toTopOf="@+id/episodeImageView"
    tools:text="Episode Title"/>
```

This places the image view in the upper-left corner and the episode title in the right.

Now to take care of the primary player transport controls. In the `playerControls` `ConstraintLayout` section, add the following:

```xml
<ImageButton
    android:id="@+id/replayButton"
    android:layout_width="34dp"
    android:layout_height="34dp"
    android:layout_marginEnd="24dp"
    android:layout_marginTop="8dp"
    android:background="@android:color/transparent"
    android:scaleType="fitCenter"
    android:src="@drawable/ic_replay_10_white"
    app:layout_constraintEnd_toStartOf="@+id/playToggleButton"
    app:layout_constraintTop_toTopOf="parent"/>

<Button
    android:id="@+id/playToggleButton"
    android:layout_width="34dp"
    android:layout_height="34dp"
    android:layout_marginTop="8dp"
    android:background="@drawable/ic_play_pause_toggle"
    android:scaleType="fitCenter"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.5"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"/>

<ImageButton
    android:id="@+id/forwardButton"
    android:layout_width="34dp"
    android:layout_height="34dp"
    android:layout_marginStart="24dp"
    android:layout_marginTop="8dp"
    android:background="@android:color/transparent"
    android:scaleType="fitCenter"
    android:src="@drawable/ic_forward_30_white"
    app:layout_constraintStart_toEndOf="@+id/playToggleButton"
    app:layout_constraintTop_toTopOf="parent"/>

<Button
    android:id="@+id/speedButton"
    android:layout_width="54dp"
    android:layout_height="34dp"
    android:layout_marginEnd="8dp"
    android:layout_marginTop="8dp"
    android:background="@android:color/transparent"
    android:text="1x"
    android:textColor="@android:color/white"
    android:textSize="14sp"
    android:textAllCaps="false"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent"/>
```

This adds the skip back, play/pause, skip forward and speed buttons at the top of the play controls section.

Next, after the player transport control buttons, add the following within the
`playerControls ConstraintLayout` section:

```xml
<TextView
    android:id="@+id/currentTimeTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    android:layout_marginStart="8dp"
    android:text="0:00"
    android:textColor="@android:color/white"
    android:textSize="12sp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="@+id/seekBar"/>

<SeekBar
    android:id="@+id/seekBar"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:progressBackgroundTint="@android:color/white"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toStartOf="@+id/endTimeTextView"
    app:layout_constraintStart_toEndOf="@+id/currentTimeTextView"/>

<TextView
    android:id="@+id/endTimeTextView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    android:layout_marginEnd="8dp"
    android:text="0:00"
    android:textColor="@android:color/white"
    android:textSize="12sp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="@+id/seekBar"/>
```

This adds the seek bar (scrubber) with current and end times to the bottom of the
player controls section.

Episode player fragment

It's time to build out the episode player fragment. This fragment will display the
episode layout and handle all playback logic. You'll move the media related code from
the `PodcastDetailsFragment` class into this new episode player fragment.

In the **ui** package, create a new file named **EpisodePlayerFragment.kt** with the following content:

```kotlin
class EpisodePlayerFragment : Fragment() {

  companion object {
    fun newInstance(): EpisodePlayerFragment {
      return EpisodePlayerFragment()
    }
  }

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    retainInstance = true
  }

  override fun onCreateView(inflater: LayoutInflater,
                            container: ViewGroup?,
                            savedInstanceState: Bundle?): View?{
    return inflater.inflate(R.layout.fragment_episode_player,
        container, false)
  }

  override fun onActivityCreated(savedInstanceState: Bundle?) {
    super.onActivityCreated(savedInstanceState)
  }

  override fun onStart() {
    super.onStart()
  }

  override fun onStop() {
    super.onStop()
  }
}
```

This is the minimum code required to display the fragment. It provides a companion object to create an instance of the fragment and loads the `fragment_episode_player` layout in `onCreateView()`.

> **Note**: Make sure to choose `android.support.v4.app.Fragment` for the **Fragment** import.

Episode player navigation

Before finishing the fragment code, hook up the navigation.

`PodcastActivity` will control the navigation, but it'll need to know when the user selects an episode in the detail view. For that, you can add a new method to the `OnPodcastDetails` listener which gets triggered when the selection is made.

Open **PodcastDetailsFragment.kt** and add the following code to the `OnPodcastDetails`
interface.

```
fun onShowEpisodePlayer(episodeViewData: EpisodeViewData)
```

Replace the code in `onSelectedEpisode()` with the following:

```
listener?.onShowEpisodePlayer(episodeViewData)
```

When the user selects an episode, this calls `onShowEpisodePlayer()` on the listener — in
this case, `PodcastActivity`.

Now you can implement `onShowEpisodePlayer()` in the podcast activity.

Open **PodcastActivity.kt** and add the following new method to satisfy the
OnPodcastDetailsListener interface:

```
override fun onShowEpisodePlayer(episodeViewData: EpisodeViewData) {
}
```

Before you can add the code for this method, you need some supporting code. Start with
a method that creates the episode player fragment.

Open **PodcastActivity.kt** and add the following code to the companion object:

```
private const val TAG_PLAYER_FRAGMENT = "PlayerFragment"
```

This tag keeps track of the episode player fragment.

Now, add the following method:

```
private fun createEpisodePlayerFragment(): EpisodePlayerFragment {
  var episodePlayerFragment =
      supportFragmentManager.findFragmentByTag(TAG_PLAYER_FRAGMENT) as
      EpisodePlayerFragment?

  if (episodePlayerFragment == null) {
    episodePlayerFragment = EpisodePlayerFragment.newInstance()
  }

  return episodePlayerFragment
}
```

This method uses the `supportFragmentManager.findFragmentByTag()` to first check if
the player fragment was created before. If not, then a new instance is created using
`EpisodePlayerFragment.newInstance()`. The episode player fragment is then returned
to the caller.

You can use the existing `PodcastViewModel` class to keep track of the currently active episode. This will make it simple to retrieve the active episode from the new episode player fragment.

Open **PodcastViewModel.kt** and add the following property to the class:

```
var activeEpisodeViewData: EpisodeViewData? = null
```

Back in the podcast activity, you need a method to create and show the player fragment. This will look similar to the existing `showDetailsFragment()` method.

Open **PodcastActivity.kt** and add the following new method:

```
private fun showPlayerFragment() {
  val episodePlayerFragment = createEpisodePlayerFragment()

  supportFragmentManager.beginTransaction().replace(
      R.id.podcastDetailsContainer,
      episodePlayerFragment,
      TAG_PLAYER_FRAGMENT
    ).addToBackStack("PlayerFragment").commit()
  podcastRecyclerView.visibility = View.INVISIBLE
  searchMenuItem.isVisible = false
}
```

This method creates the episode player fragment, displays the fragment and hides the podcast list recycler view. It then hides the search menu item.

Now that all of the supporting methods are in place, you're ready to implement `onShowEpisodePlayer()`.
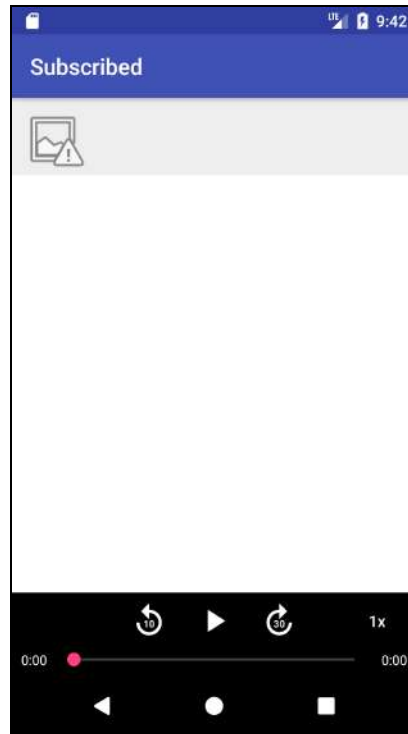
Add the following to `onShowEpisodePlayer()`:

```
podcastViewModel.activeEpisodeViewData = episodeViewData
showPlayerFragment()
```

This sets the active episode on the podcast view model and calls `showPlayerFragment()` to display the player fragment.

Build and run the app.

Display the details for a podcast and tap on an episode. Although the episode player fragment is displayed, it'll be blank since you haven't populated any of the views yet. Press the back button, and it will navigate back to the podcast details screen.

Episode player details

It's time to get some episode data on the player screen. You'll use the active episode view data from the podcast view model to populate the views.

Open **EpisodePlayerFragment.kt** and add the following property to the class:

```
private lateinit var podcastViewModel: PodcastViewModel
```

Now, add the following method:

```
private fun setupViewModel() {
  podcastViewModel = ViewModelProviders.of(activity)
      .get(PodcastViewModel::class.java)
}
```

This assigns the `podcastViewModel` property to the active podcast view model.

Next, add the following to the bottom of `onCreate()`:

```
setupViewModel()
```

The view model is set up when the fragment is created.

Now, you need to create a method to set up the view controls using the view model data. Add the following new method:

```
private fun updateControls() {
  // 1
  episodeTitleTextView.text =
      podcastViewModel.activeEpisodeViewData?.title

  // 2
  val htmlDesc =
      podcastViewModel.activeEpisodeViewData?.description ?: ""
  val descSpan = HtmlUtils.htmlToSpannable(htmlDesc)
  episodeDescTextView.text = descSpan
  episodeDescTextView.movementMethod = ScrollingMovementMethod()

  // 3
  Glide.with(activity)
    .load(podcastViewModel.activePodcastViewData?.imageUrl)
    .into(episodeImageView)
}
```

Let's take this one item at a time:

1. Set the episode title text view to the episode title.

2. Just like the podcast description that is shown on the podcast details view, the episode description can have HTML formatting that will cause display issues if set directly on a text view widget. This code uses the previously created `htmlToSpannable()` method to clean up the episode description and make it display correctly.

3. Use Glide to load in the podcast album art and assign it to the episode image view widget.

Add the call to `updateControls()` to the bottom of onActivityCreated():

```
  updateControls()
```

Build and run the app.

Load a podcast episode to view the details. If the episode description is long enough, you can scroll to read the full content.
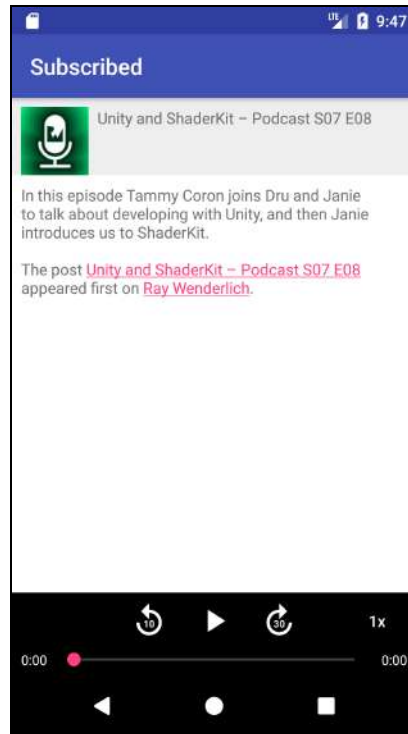
Episode player controls

Now you can turn your attention to the player controls. You'll get the basic play, pause and skip controls working first; then you'll focus on the seek bar and speed control.

Start by moving some media playback code from the `PodcastDetailsFragment` class to the new `EpisodePlayerFragment` class.

> **Note**: Make sure to delete the code from `PodcastDetailsFragment` when moving it to `EpisodePlayerFragment`.

Let's break this process out step-by-step:

1. Move the following properties from **PodcastDetailsFragment.kt** to **EpisodePlayerFragment.kt**:

   ```
   private lateinit var mediaBrowser: MediaBrowserCompat
   private var mediaControllerCallback: MediaControllerCallback? = null
   ```

   If Android Studio changes `MediaControllerCallback` to `PodcastDetailsFragment.MediaControllerCallback`, change it back to `MediaControllerCallback`; it will show a compile error until you get to step five.

2. Move the `startPlaying()` method from **PodcastDetailsFragment.kt** to **EpisodePlayerFragment.kt**.

3. Move the code below `super.onStart()` from `onStart()` in **PodcastDetailsFragment.kt** to `onStart()` in **EpisodePlayerFragment.kt**.

4. Move the code below `super.onStop()` from `onStop()` in **PodcastDetailsFragment.kt** to `onStop()` in **EpisodePlayerFragment.kt**.

5. Move the `MediaBrowserCallBacks` and `MediaControllerCallback` inner classes from **PodcastDetailsFragment.kt** to **EpisodePlayerFragment.kt**.

6. Move the `initMediaBrowser()` method from **PodcastDetailsFragment.kt** to **EpisodePlayerFragment.kt**.

7. Move the `registerMediaController()` method from **PodcastDetailsFragment.kt** to **EpisodePlayerFragment.kt**.

8. Move the call to `initMediaBrowser()` from `onCreate()` in **PodcastDetailsFragment.kt** to `onCreate()` in **EpisodePlayerFragment.kt**.

Play/Pause button

Now it's time to hook up the Play/Pause button to start and stop playback.

Add the following method to **EpisodePlayerFragment**:

```
private fun togglePlayPause() {
  val controller = MediaControllerCompat.getMediaController(activity)
  if (controller.playbackState != null) {
    if (controller.playbackState.state ==
        PlaybackStateCompat.STATE_PLAYING) {
      controller.transportControls.pause()
    } else {
      podcastViewModel.activeEpisodeViewData?.let { startPlaying(it) }
    }
  } else {
    podcastViewModel.activeEpisodeViewData?.let { startPlaying(it) }
  }
}
```

This is very similar to the playback code you created in the previous chapter — it gets the current media controller, then it either pauses or starts playback, based on its current state.

Add the following method to listen for the tap on the play/pause button:

```
private fun setupControls() {
  playToggleButton.setOnClickListener {
```

```
    togglePlayPause()
  }
}
```

This sets a listener on `playToggleButton` and calls `togglePlayPause()` when it's tapped.

That's enough to get the media playing, but you also need to update the play/pause button to show the pause icon when playing and the play icon when paused.

You can update the button icon directly in `togglePlayPause()`, but that won't keep it in sync if playback is changed from outside the app. To keep the play/pause button in sync — regardless of how the state is changed — use the `onPlaybackStateChanged()` event from the media controller.

First, create a method to handle playback state changed.

Add the following method:

```
private fun handleStateChange(state: Int) {
  val isPlaying = state == PlaybackStateCompat.STATE_PLAYING
  playToggleButton.isActivated = isPlaying
}
```

This sets the play/pause button state to activated if the media is playing or not activated if the media is paused. This results in the button icon changing because the button background in the layout XML is set to the **ic_play_pause_toggle.xml** selector. If you open this selector, you'll see that it specifies the play button for the inactive state and the pause button for the active state.

Call this method when the playback state changes. Add the following to `onPlaybackStateChanged()` in the `MediaControllerCallback` inner class:

```
val state = state ?: return
handleStateChange(state.getState())
```

Finally, add the call to `setupControls()` before the call to `updateControls()` in `onActivityCreated()`:

```
setupControls()
```

Speed control button

Next, you'll hook up the speed control button. This button will increase the speed by 0.25x times each time it's tapped up to a maximum of 2.0x. It will go to 0.75x after reaching the max of 2.0x.

Unlike the play and pause commands, the media session doesn't have a built-in command to change the playback speed. So how do you inform that media browser service that you want to change the speed? The answer is a custom command!

You'll add a new method to intercept custom commands when they come into the media session callback class. The custom command will have a name and a Bundle object with the command parameters.

First, define some constants for the custom command name and the key used in the Bundle object.

Open **PodplayMediaCallback.kt** and add the following companion object:

```
companion object {
  const val CMD_CHANGESPEED = "change_speed"
  const val CMD_EXTRA_SPEED = "speed"
}
```

This defines a speed change command string and key for the speed.

Next, update `setState()` to handle a speed option.

Change the `setState()` declaration to the following:

```
private fun setState(state: Int, newSpeed: Float? = null) {
```

This allows an optional `newSpeed` parameter to be passed to `setState()`.

Before making the changes to `setState()`, take a look at the `setState` call that's executed on the `PlaybackStateCompat.Builder()` object. Notice there's a speed parameter as part of the state.

This speed parameter does not change the playback speed! It only sets the state on the Media Session. You need to change the speed setting directly on the MediaPlayer to affect the playback speed.

In `setState()`, add the following before the call to `PlaybackStateCompat.Builder()`:

```
// 1
var speed = 1.0f
// 2
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
  if (newSpeed == null) {
    // 3
    speed = mediaPlayer?.getPlaybackParams()?.speed ?: 1.0f
  } else {
    // 4
    speed = newSpeed
  }
```

```
  mediaPlayer?.let {
    // 5
    try {
      it.playbackParams = it.playbackParams.setSpeed(speed)
    }
    catch (e: Exception) {
      // 6
      mediaPlayer?.reset()
      mediaPlayer?.setDataSource(context, mediaUri)
      mediaPlayer?.prepare()
      // 7
      it.playbackParams = it.playbackParams.setSpeed(speed)
      // 8
      mediaPlayer?.seekTo(position.toInt())
      // 9
      if (state == PlaybackStateCompat.STATE_PLAYING) {
        mediaPlayer?.start()
      }
    }
  }
}
```

Let's go over this one step at a time:

1. Start by setting the default `speed` to 1.0.

2. The MediaPlayer gained the ability to change the speed beginning with Android 6.0 (Marshmallow). If the version supports speed control, then the code block is executed.

3. If no new speed has been specified, then `speed` is set to the media player's current speed.

4. If a new speed is present, then `speed` is set to the new speed.

5. The media player speed is updated to the new speed by setting a new `mediaPlayer.playbackParams` property. You can't change the speed directly on the `playbackParams`. A new `playbackParams` object must be assigned to the media player. This call can throw an exception on some versions of Android, so it is surrounded by a try block.

6. If the update to `playbackParams` throws an exception, then the player needs to be reset to clear the state. After a reset, the data source must be set again on the player.

7. Now that the player has been reset, it's safe to update the `playbackParams`.

8. Resetting the player will set the playback position back to 0. `seekTo()` is called to set it back to the previous position.

9. If the state is set to playing, then the player is started after the reset.

In `setState()`, update the call to `setState()` on `PlaybackStateCompat.Builder()` to pass in the speed for the third parameter:

```
.setState(state, position, speed)
```

Next, add a method to extract the speed from a bundle object and call `setState()` with the speed:

```kotlin
private fun changeSpeed(extras: Bundle) {
  var playbackState = PlaybackStateCompat.STATE_PAUSED
  if (mediaSession.controller.playbackState != null) {
    playbackState = mediaSession.controller.playbackState.state
  }
  setState(playbackState, extras.getFloat(CMD_EXTRA_SPEED))
}
```

When the speed is changed, you don't want to change the playback state. This is accomplished by taking the current playback state and passing it into `setState()`. `playbackState` is set to the current playback state if it is valid. If not, `playbackState` is set to the default state of `STATE_PAUSED`. You call `setState()` with `playbackState` and the new playback speed.

Now you can add the method to process the custom command.

Add the following method to the `PodplayMediaCallback` class:

```kotlin
override fun onCommand(command: String?, extras: Bundle?,
    cb: ResultReceiver?) {
  super.onCommand(command, extras, cb)
  when (command) {
    CMD_CHANGESPEED -> extras?.let { changeSpeed(it) }
  }
}
```

> **Note**: Make sure to select `import android.os.ResultReceiver` for the `ResultReceiver` import.

`onCommand()` is called by the media session when a custom command is received. You check for the `CMD_CHANGESPEED` command and then call `changeSpeed()` with the `extras` Bundle object.

Now, the episode player fragment just needs to send the custom command when the user changes the speed.

First, you'll need a property to keep track of the current playback speed. Open **EpisodePlayerFragment.kt** and add the following property to the **EpisodePlayerFragment** class:

```
private var playerSpeed: Float = 1.0f
```

This property keeps track of the current speed.

Next, add a method to change the speed by sending the custom command to the media controller.

Add the following method:

```
private fun changeSpeed() {
  // 1
  playerSpeed += 0.25f
  if (playerSpeed > 2.0f) {
    playerSpeed = 0.75f
  }
  // 2
  val bundle = Bundle()
  bundle.putFloat(CMD_EXTRA_SPEED, playerSpeed)
  // 3
  val controller = MediaControllerCompat.getMediaController(activity)
  controller.sendCommand(CMD_CHANGESPEED, bundle, null)
  // 4
  speedButton.text = "${playerSpeed}x"
}
```

Let's break this down.

1. Increase `playerSpeed` by 0.25. If the speed goes past 2.0, it's set back to 0.75.

2. Create a bundle and set the `CMD_EXTRA_SPEED` key to the value of `playerSpeed`.

3. The `CMD_CHANGESPEED` command is sent to the media controller along with the bundle object.

4. Update the speed button text label to show the current playback speed.

You also need to make sure the speed control label is correct after a screen rotation. Add the following line to the end of `updateControls()`:

```
speedButton.text = "${playerSpeed}x"
```

Now, the speed control button needs to call `changeSpeed()`.

Add the following to the end of `setupControls()`:

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
  speedButton.setOnClickListener {
```

```
    changeSpeed()
  }
} else {
  speedButton.visibility = View.INVISIBLE
}
```

This first checks to see if the device supports the speed setting. If it does, the `onClickListener` is set on the speed button. The listener calls `changeSpeed()` when the user taps the speed button. If the device does not support speed control, then the speed button is hidden.

Skip buttons

OK, it's time to implement the skip forward and back functionality. The media controller allows you to change the playback position directly. To perform a skip, you'll take the current playback position, add a plus or minus offset to get a new position, and then set the new position.

Before adding the changes to the player fragment, you need to update the media browser to allow seeking to a specific playback position. This is done by overriding an additional method in the `PodplayMediaCallback` class.

Open **PodplayMediaCallback.kt** and add the following method:

```
override fun onSeekTo(pos: Long) {
  super.onSeekTo(pos)
  // 1
  mediaPlayer?.seekTo(pos.toInt())
  // 2
  val playbackState: PlaybackStateCompat? =
      mediaSession.controller.playbackState
  // 3
  if (playbackState != null) {
    setState(playbackState.state)
  } else {
    setState(PlaybackStateCompat.STATE_PAUSED)
  }
}
```

`onSeekTo()` is called by the media session when the `seekTo` command is received.

Here's what's going on.

1. Call `seekTo()` on the `mediaPlayer` to change the playback position.

2. Retrieve the playback state from the media controller.

3. Call `setState()` so any media browser clients will know about the change in position. This is an important step, as it keeps all media browser client UIs in sync.

If `playbackState` is not `null`, then `setState()` is called with the current state. This ensures that the player keeps playing or stays paused depending on the current playback state.

If `playbackState` is `null`, then playback state is set to paused.

Next, you need a method in the episode player fragment that performs the seek using the media controller.

Open **EpisodePlayerFragment.kt** and add the following method:

```
private fun seekBy(seconds: Int) {
  val controller = MediaControllerCompat.getMediaController(activity)
  val newPosition = controller.playbackState.position + seconds*1000
  controller.transportControls.seekTo(newPosition)
}
```

This starts by grabbing the media controller and then computes a new playback position by adding to the current playback position. The seconds are multiplied by 1000 to convert to milliseconds as used by the media controller.

Call `seekTo()` on the media controller transport controls.

This will invoke the `onSeekTo()` method you defined in the media browser service.

Now you'll add listeners on the skip buttons and call the new `seekBy` method.

Add the following to the bottom of `setupControls()`:

```
forwardButton.setOnClickListener {
  seekBy(30)
}
replayButton.setOnClickListener {
  seekBy(-10)
}
```

This sets a listener on the `forwardButton` that calls `seekBy()` with a forward skip of 30 seconds. It sets a listener on the `replayButton` that calls `seekBy()` with a backward skip of 10 seconds.

The skip buttons are now fully operational.

Build and run the app.

Bring up a podcast episode and test out the playback controls. You can play and pause the episode, skip forward and backward and change the speed.

Pull down the notification drawer and pause the playback from there. Notice that the play/pause button icon in the app stays in sync.

The final playback control to implement is the scrubber with its associated time labels.

Scrubber control

There are a few steps required to make the scrubber functional.

1. Update the end time label to reflect the episode duration.

2. Keep the scrubber position and current time label updated to match the current playback position.

3. Update the playback position when the user drags the scrubber.

Setting the end time label is reasonably straightforward, but not as straightforward as it seems. You may be tempted to take the duration stored in the **Episode** model and use it to set the label. Unfortunately, the duration provided in the RSS feed is not always accurate, and not always formatted consistently.

The safest way to set the end time is to get the episode duration from the media controller metadata. There's just one problem; your media browser service doesn't set the duration! You need to fix that first.

Open **PodplayMediaCallback.kt** and add the following line to `MediaMetadataCompat.Builder()` calls in `prepareMedia()`:

```
.putLong(MediaMetadataCompat.METADATA_KEY_DURATION,
    mediaPlayer.duration.toLong())
```

This takes the duration reported by the media player and sets the proper metadata key on the media session.

Now the episode player can use this metadata when the playback state changes.

Open **EpisodePlayerFragment.kt** and add the following property to the `EpisodePlayerFragment` class:

```
private var episodeDuration: Long = 0
```

This will store the current episode duration.

Add the following method:

```
private fun updateControlsFromMetadata(metadata: MediaMetadataCompat) {
  episodeDuration =
      metadata.getLong(MediaMetadataCompat.METADATA_KEY_DURATION)
  endTimeTextView.text = DateUtils.formatElapsedTime(
      episodeDuration / 1000)
}
```

> **Note**: Make sure to select `android.text.format.DateUtils` for the `DateUtils` import.

This sets the `episodeDuration` from the `METADATA_KEY_DURATION` metadata value. If the value doesn't exist, then the duration is set to 0. It then uses the duration to set the end time label.

`DateUtils.formatElapsedTime()` takes the time in seconds and returns a formatted time string as hours:minutes:seconds.

Now call this new method when the metadata changed.

Add the following to the bottom of `onMetadataChanged()` in the inner `MediaControllerCallback` class:

```
metadata?.let { updateControlsFromMetadata(it) }
```

This calls `updateControlsFromMetadata()` if the metadata is not null.

Next, you'll add code to keep the scrubber and the current time label in sync with the current playback position.

Add the following line to the end of `updateControlsFromMetadata()`:

```
seekBar.setMax(episodeDuration.toInt())
```

This sets the range of the scrubber `seekBar` to match the episode duration. This lets you set the progress value on the `seekBar` directly to the playback position in milliseconds, and it will place the progress indicator at the correct position.

Next, you'll update the current time label as the scrubber indicator position changes, and update the playback position after the user drags the scrubber indicator to a new position.

You can handle both of these tasks by implementing a change listener on the scrubber bar.

First, add a property to keep track of when the user is dragging the scrubber indicator. The reason for this will be explained shortly.

Add the following property to the `EpisodePlayerFragment` class:

```
private var draggingScrubber: Boolean = false
```

Add the following to the end of `setupControls()`:

```
// 1
seekBar.setOnSeekBarChangeListener(
    object : SeekBar.OnSeekBarChangeListener {
  override fun onProgressChanged(seekBar: SeekBar, progress: Int,
      fromUser: Boolean) {
    // 2
    currentTimeTextView.text = DateUtils.formatElapsedTime(
        (progress / 1000).toLong())
  }
  override fun onStartTrackingTouch(seekBar: SeekBar) {
    // 3
    draggingScrubber = true
  }
  override fun onStopTrackingTouch(seekBar: SeekBar) {
    // 4
    draggingScrubber = false
    // 5
    val controller = MediaControllerCompat.getMediaController(activity)
    if (controller.playbackState != null) {
      // 6
      controller.transportControls.seekTo(seekBar.progress.toLong())
    } else {
      // 7
      seekBar.progress = 0
    }
  }
})
```

Let's step through the code.

1. Set a change listener object on the `seekBar`.

2. `seekBar` calls `onProgressChanged()` each time the scrubber position changes. You use this as an opportunity to update the current time label and format it to hours:minutes:seconds.

3. `seekBar` calls `onStartTrackingTouch()` when the user starts to drag the scrubber indicator. `draggingScrubber` is set to true.

4. `seekBar` calls `onStopTrackingTouch()` when the user stops dragging the scrubber indicator. `draggingScrubber` is set to false, and the playback position is updated.

5. Retrieve the controller object from the activity.

6. If the controller playback state is valid, then seek directly to the new playback position where the user stopped dragging the scrubber indicator.

7. If the controller playback state is valid, set the scrubber position back to the beginning.

That's all you need to allow the user to drag the scrubber to any playback position.

Now you need to update the scrubber position as the play continues. There are several ways you can do this.

One option is to use a **ScheduledExecutorService** that runs a method every second. In this method, you query for the current playback state position from the media controller and update the scrubber position accordingly.

For PodPlay, you'll treat the scrubber movement as an animation. You know how much time is left in the episode and the playback speed, so you can use this to smoothly animate the scrubber indicator until it reaches the end of the scrubber bar.

You'll implement the animation using a **ValueAnimator**. You can think of the **ValueAnimator** as an engine that pumps out values at a steady rate. You'll use these values to update the scrubber as long as the playback continues.

First, you need a property to hold the **ValueAnimator** object so it can be canceled if needed.

Add the following property to the `EpisodePlayerFragment` class:

```
private var progressAnimator: ValueAnimator? = null
```

Now you can create a method to build the animation and kick it off.

Add the following method to the `EpisodePlayerFragment` class:

```
// 1
private fun animateScrubber(progress: Int, speed: Float) {
  // 2
  val timeRemaining = ((episodeDuration – progress) / speed).toInt()
  // 3
  progressAnimator = ValueAnimator.ofInt(
      progress, episodeDuration.toInt())
  progressAnimator?.let { animator ->
    // 4
    animator.duration = timeRemaining.toLong()
    // 5
    animator.interpolator = LinearInterpolator()
    // 6
    animator.addUpdateListener {
      if (draggingScrubber) {
```

```
      // 7
      animator.cancel()
    } else {
      // 8
      seekBar.progress = animator.animatedValue as Int
    }
  }
  // 9
  animator.start()
  }
}
```

Here's what's happening:

1. `animateScrubber()` takes in the current progress and playback speed.

2. You compute the time remaining until the end of the episode.

3. Create a new `ValueAnimator` with the starting and ending value of the animation and assign it to the `progressAnimator` property.

4. The animation duration is set to the time remaining. This stops the animation when it reaches the end of the episode.

5. By default, the `ValueAnimator` uses a non-linear time interpolation where it accelerates at the beginning and decelerates at the end of the animation. The interpolation is set to linear to ensure an even animation.

6. Set an update listener on the animator. This listener is called by the animator on each step of the animation.

7. This is where the `draggingScrubber` property you set earlier comes into play. If the user is dragging the scrubber then you need to cancel the animation, or it will get into a tug-of-war with the user, and it will not end well.

8. If the user is not dragging the scrubber, then update the scrubber indicator to the current value from the animator.

9. Start the animation.

Now use this new method when the playback state changes to playing.

You can also make sure the scrubber position is updated when the playback state changes. First, update `handleStateChange()` to use the current playback position and speed.

Update `handleStateChange()` declaration to the following:

```
private fun handleStateChange(state: Int, position: Long, speed: Float) {
```

Add the following to the end of `handleStateChange()`:

```
val progress = position.toInt()
seekBar.progress = progress
speedButton.text = "${playerSpeed}x"

if (isPlaying) {
  animateScrubber(progress, speed)
}
```

This starts by getting the current progress from the playback state, and then it sets the scrubber to the current progress position and updates the speed control label. If the media is playing, then start the scrubber animation.

Update the call to `handleStateChange()` in `onPlaybackStateChanged()` to the following:

```
handleStateChange(state.state, state.position, state.playbackSpeed)
```

This passes in the additional parameters added to `handleStateChange()`.

You also need to stop the animation when the playback stops. Add the following to the beginning of `handleStateChange()`:

```
progressAnimator?.let {
  it.cancel()
  progressAnimator = null
}
```

If the animator is not `null`, then cancel it and set it back to `null`.

Finally, cancel the animation when the fragment is stopped.

Add the following after the call to `super.onStop()` in `onStop()`:

```
progressAnimator?.cancel()
```

One minor addition is needed to update the controls after the screen is rotated.

Create the following method to update the controls based on the media controller state:

```
private fun updateControlsFromController() {
  val controller = MediaControllerCompat.getMediaController(activity)
  if (controller != null) {
    val metadata = controller.metadata
    if (metadata != null) {
      handleStateChange(controller.playbackState.state,
```

```
            controller.playbackState.position, playerSpeed)
      updateControlsFromMetadata(controller.metadata)
    }
  }
}
```

This method calls `handleStateChange` and `updateControlsFromMetadata` to make sure the controls match the playback state after a screen rotation.

Now you'll call this new method from a couple of key places.

Add the call to the end of `onConnected()` in `MediaBrowserCallBacks`:

```
updateControlsFromController()
```

Add the call to `onStart()` before the else statement.

```
updateControlsFromController()
```

Build and run the app. Start playback for an episode.

Notice that the current time on the left of the scrubber stays in sync with the playback position and the end time displays the episode duration.

The scrubber indicator moves along with the playback, and you're able to drag the scrubber to jump to any playback position.
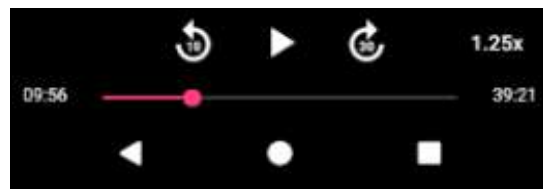
Video playback

The last feature you'll implement is video playback. If you try to play a video podcast with PodPlay now, only the audio part will play.

Unlike audio, video playback is a captive experience and is intended to run in the foreground with a UI. For this reason, you'll abandon the client/server architecture used for audio playback when playing back videos.

You'll still use a **MediaSession** and **MediaPlayer** along with the **PodplayMediaCallback** class, but it will be controlled from **EpisodePlayerFragment** instead of **MediaBrowserService**.

Identifying videos

The first thing you need is a means to identify if the episode media is a video.

Open **PodcastViewModel.kt** and add the following to **EpisodeViewData**:

```
var isVideo: Boolean = false
```

Replace the contents of `episodesToEpisodesView()` with the following:

```
return episodes.map {
  val isVideo = it.mimeType.startsWith("video")
  EpisodeViewData(it.guid, it.title, it.description, it.mediaUrl,
      it.releaseDate, it.duration, isVideo)
}
```

This checks the mime type on each episode to see if it starts with the string "video". If so, the `isVideo` property on the `EpisodeViewData` is set to `true`.

Now you need to update **EpisodePlayerFragment.kt** to handle video playback.

To start video playback, you'll need to perform a few tasks:

1. Create a media session and media player. This is handled in `MediaBrowserService` for audio files, but for video, it needs to be done in `EpisodePlayerFragment`.

2. Update the UI to make the video visible and hide the other UI elements.

3. Prepare the `SurfaceView` to playback the video.

Media session

You'll need a **MediaSession** object to manage the video playback.

Open **EpisodePlayerFragment.kt** and add the following property to the class:

```
private var mediaSession: MediaSessionCompat? = null
```

Add the following method to initialize the media session:

```
private fun initMediaSession() {
  if (mediaSession == null) {
    // 1
    mediaSession = MediaSessionCompat(activity, "EpisodePlayerFragment")
    // 2
    mediaSession?.setFlags(
        MediaSessionCompat.FLAG_HANDLES_MEDIA_BUTTONS or
            MediaSessionCompat.FLAG_HANDLES_TRANSPORT_CONTROLS)
    // 3
    mediaSession?.setMediaButtonReceiver(null)
  }
```

```
    registerMediaController(mediaSession!!.sessionToken)
  }
```

This is similar to the code created in the last chapter for **MediaBrowserService**.

1. Create the media session if it does not already exist.

2. Set flags to indicate the session will handle media buttons and transport controls.

3. Set the media button receiver to `null` so media buttons will be ignored if the app is not in the foreground.

Media player

You'll also need a **MediaPlayer** object just like you did with the **MediaBrowserService**:

Add the following property to **EpisodePlayerFragment**:

```
  private var mediaPlayer: MediaPlayer? = null
```

And you'll need to know if the user taps the play button before the media is ready to play.

Add the following property to **EpisodePlayerFragment**:

```
  private var playOnPrepare: Boolean = false
```

The media player needs a view on which to display the video. This is where the `videoSurfaceView` comes into the picture.

Once the media player has loaded the video, the `videoSurfaceView` needs to be resized to match the video aspect ratio.

Add the following method to resize the video surface view.

```
  private fun setSurfaceSize() {
    // 1
    val mediaPlayer = mediaPlayer ?: return
    // 2
    val videoWidth = mediaPlayer.videoWidth
    val videoHeight = mediaPlayer.videoHeight
    // 3
    val parent = videoSurfaceView.parent as View
    val containerWidth = parent.width
    val containerHeight = parent.height
    // 4
    val layoutAspectRatio = containerWidth.toFloat() / containerHeight
    val videoAspectRatio = videoWidth.toFloat() / videoHeight
    // 5
    val layoutParams = videoSurfaceView.layoutParams
    // 6
```

```
    if (videoAspectRatio > layoutAspectRatio) {
      layoutParams.height = (containerWidth / videoAspectRatio).toInt()
    } else {
      layoutParams.width = (containerHeight * videoAspectRatio).toInt()
    }
    // 7
    videoSurfaceView.layoutParams = layoutParams
  }
```

This method's job is to make the video view match the size of the podcast video and keep the video aspect ratio intact. It does this by taking the longest side of the video and making it fit the view, and then adjusting the other side to keep the original ratio intact.

1. If the media player is `null`, the method returns early

2. Retrieve the current width and height of the video.

3. Retrieve the current width and height of the video surface container view.

4. Compute the surface view layout aspect ratio.

5. Compute the video aspect ratio.

6. If the video ratio is larger than the surface view layout ratio, then the surface view layout width is retained, and the height is shrunk to keep the video aspect ratio.

7. If the video ratio is smaller than the surface view layout ratio, then the surface view layout height is retained, and the width is shrunk to keep the video aspect ratio.

Now you can call this from the media player initialization code.

Add the following method:

```
  private fun initMediaPlayer() {
    if (mediaPlayer == null) {
      // 1
      mediaPlayer = MediaPlayer()
      mediaPlayer?.let {
        // 2
        it.setAudioStreamType(AudioManager.STREAM_MUSIC)
        // 3
        it.setDataSource(podcastViewModel.activeEpisodeViewData?.mediaUrl)
        // 4
        it.setOnPreparedListener {
          // 5
          val episodeMediaCallback = PodplayMediaCallback(activity,
              mediaSession!!, it)
          mediaSession!!.setCallback(episodeMediaCallback)
          // 6
          setSurfaceSize()
          // 7
```

```
        if (playOnPrepare) {
          togglePlayPause()
        }
      }
      // 8
      it.prepareAsync()
    }
  } else {
    // 9
    setSurfaceSize()
  }
}
```

Let's break this down.

1. If the media player is `null`, create a new one.

2. Set the media player audio stream type to music.

3. Set the media player data source to the episode media URL.

4. Set the `onPreparedListener` method on the media player.

5. Once the media is ready, the `PodplayMediaCallback` object is created and assigned as the callback on the current media session.

6. Set the video surface size to match the video.

7. If `playOnPrepare` is true, indicating that the user has already tapped the play button, then the video is started.

8. Call `prepareAsync()` on the media player to have it prepare the video in the background.

9. If the media player is not `null` then you only need to set the video surface size. This will happen if there is a configuration change, such as a screen rotation.

The `playOnPrepare` flag should be set to true when the play button is tapped. It doesn't matter that it gets set each time, as long as you know that it was tapped at least once.

Add the following to the beginning of `togglePlayPause()`:

```
playOnPrepare = true
```

Finally, add the following method to initialize the video surface and call the new `initMediaPlayer` method:

```
private fun initVideoPlayer() {
  // 1
  videoSurfaceView.visibility = View.VISIBLE
  // 2
```

```
    val surfaceHolder = videoSurfaceView.holder
    // 3
    surfaceHolder.addCallback(object: SurfaceHolder.Callback {
      override fun surfaceCreated(holder: SurfaceHolder) {
        // 4
        initMediaPlayer()
        mediaPlayer?.setDisplay(holder)
      }
      override fun surfaceChanged(var1: SurfaceHolder, var2: Int,
          var3: Int, var4: Int) {
      }
      override fun surfaceDestroyed(var1: SurfaceHolder) {
      }
    })
  }
```

SurfaceView overview

This method warrants some explanation on how surface views interact with the media player. To display videos, the MediaPlayer object requires access to a **SurfaceView**. Surface views provide a dedicated drawing surface within your view hierarchy.

When a surface view is made visible, Android must prepare it for use. Surface views provide a **SurfaceHolder** object that can be used to determine the surface availability.

Surface holders provide a **SurfaceHolder.Callback** interface to provide notifications about the surface state. The surface view is only available when the surfaceCreated() method is called on the surface holder callback object.

With that in mind, let's go over the method one step at a time.

1. The video surface view is made visible.

2. You get a reference to the underlying surface holder.

3. You call addCallback() and provide a SurfaceHolder.Callback object to detect when the surface is created.

4. Once the surface is created, the media player is initialized, and the surface is assigned as the display object for the media player.

Now you'll add some conditional code that skips the MediaBrowser creation and usage if it is a video.

First, create a property to store the video state.

Add the following property to **EpisodePlayerFragment**:

```
private var isVideo: Boolean = false
```

Add the following to the end of `setupViewModel()`:

```
isVideo = podcastViewModel.activeEpisodeViewData?.isVideo ?: false
```

Now, set a condition around all the code that references the media browser:

Surround the call to `initMediaBrowser()` in `onCreate()` as follows:

```
if (!isVideo) {
  initMediaBrowser()
}
```

`initMediaBrowser()` is only called if the media is not a video.

Surround the code in `onStart()` with a check for `isVideo`:

```
if (!isVideo) {
  if (mediaBrowser.isConnected) {
    if (MediaControllerCompat.getMediaController(activity) == null) {
      registerMediaController(mediaBrowser.sessionToken)
    }
    updateControlsFromController()
  } else {
    mediaBrowser.connect()
  }
}
```

The media browser connection logic is only implemented if the media is not a video.

Add the following code to the end of `onStop()`:

```
if (isVideo) {
  mediaPlayer?.setDisplay(null)
}
```

Clearing the display surface is required on some versions of Android to prevent issues when the screen is rotated.

Next, add conditional code that initializes the player if the episode is a video.

Add the following before the call to `updateControls()` in `onActivityCreated()`:

```
if (isVideo) {
  initMediaSession()
  initVideoPlayer()
}
```

This initializes the media session and video player when the activity is created.

There's one last bit of conditional code for videos. When a video is playing, you want to hide the episode header, episode description and action bar, while making the media controls container partly transparent.

This will allow the video to take up the maximum amount of screen space.

Add the following method to set up the video UI changes.

```
private fun setupVideoUI() {
  episodeDescTextView.visibility = View.INVISIBLE
  headerView.visibility = View.INVISIBLE
  val activity = activity as AppCompatActivity
  activity.supportActionBar?.hide()
  playerControls.setBackgroundColor(Color.argb(255/2, 0, 0, 0))
}
```

This hides everything on the screen except the video controls. It sets the background color to a 50% transparency level.

Call `setupVideoUI()` when the video is playing:

Add the following as the first line inside the "`if (isPlaying) {`" section of `handleStateChange()`:

```
if (isVideo) {
  setupVideoUI()
}
```

You'll need to manually stop the playback when the fragment is exited.

Add the following to the end of `onStop()`:

```
if (!activity.isChangingConfigurations) {
  mediaPlayer?.release()
  mediaPlayer = null
}
```

If the fragment is not stopping due to a configuration change, then stop the playback and release the media player. If the fragment is stopped during a configuration change, such as a screen rotation, then the media player is not recreated.

There's one more change required to handle the playback controls properly when the screen is rotated.

Add the following to the end of `updateControls()`:

```
mediaPlayer?.let {
  updateControlsFromController()
}
```

If the `mediaPlayer` object is not `null`, then the controls are updated from the media controller state.

There is one minor change required in **PodplayMediaCallback.kt** to make sure the media player is not prepared a second time. This is needed because `prepareAsync()` is already called in the episode player fragment when the media is a video.

In **PodplayMediaCallback.kt**, add the follow property to the `PodplayMediaCallback` class:

```
private var mediaNeedsPrepare: Boolean = false
```

This property will be used to indicate if the media player needs to be prepared.

In `initializeMediaPlayer()`, add the following line after the call to `setOnCompletionListener()`.

```
mediaNeedsPrepare = true
```

This sets `mediaNeedsPrepare` to true only if the `mediaPlayer` is created by `PodplayMediaCallback`. When playing back videos, the `mediaPlayer` is created by the `EpisodePlayerFragment` and passed into `PodplayMediaCallback`, so `mediaNeedsPrepare` will not be set to true.

In `prepareMedia()`, replace the following code,

```
mediaPlayer.reset()
mediaPlayer.setDataSource(context, mediaUri)
mediaPlayer.prepare()
```

with this block:

```
if (mediaNeedsPrepare) {
  mediaPlayer.reset()
  mediaPlayer.setDataSource(context, mediaUri)
  mediaPlayer.prepare()
}
```

The `mediaPlayer` is only prepared if `mediaNeedsPrepare` is true.

That's all the changes required in the shared **PodplayMediaCallback** object to support video playback. All of the existing controls, including skip and speed, will work without any changes.

Build and run the app.

Find a video podcast and bring up an episode. When the episode player is first displayed, it won't look any different than a standard audio podcast. Once you tap the play button, it will show the video.

> **Note**: Depending on your connection, there can be a 1-5 second delay after you press the play button before the video starts playing.

If the video fills the screen, the playback controls will overlay the video. If you rotate the screen, the video will keep playing and adapt to the new screen orientation.

Where to go from here?

Congratulations, you now have a very functional Podcast player worthy of praise and bragging rights! Pat yourself in the back because you've accomplished a lot.

There are plenty of opportunities to improve and take the Podcast player to the next level. Here are just a few ideas:

- Start from the last playback position when a user resumes a podcast. Hint: Add a new `lastPosition` property to the Episode model, and update it when playback stops.

- Notify your users periodically with a curated list of the top podcasts. Hint: Use **Firebase Cloud Messaging**. Learn more at https://firebase.google.com/docs/cloud-messaging/.

- Add the ability to create playlists.

- Add an option to download episodes for offline listening. Hint: Check out **DownloadManager** at https://developer.android.com/reference/android/app/DownloadManager.html.

- Add an option to manually add a podcast from an RSS URL.

In the next few chapters, we're going to discuss some important topics like how to keep your app up to date, preparing to release it, even testing and publishing. So, sit back, relax and let's put a bow on these new skills of yours!

Section V: Android Compatibility

This section covers two Android topics that are almost as important as your Android app itself: how to handle the collection of Android versions out there, known as the *fragmentation problem*, and how to best keep your app up to date in the face of constant updates to Android.

Chapter 28: Android Fragmentation & Support Libraries

Chapter 29: Keeping Your App Up To Date

Chapter 28: Android Fragmentation & Support Libraries

By Darryl Bayliss

In a perfect world, every Android device would run a single version of Android and app development would be easy. Sadly, the world isn't perfect. As of May 2017, there were two *billion* active Android devices around the world, all running various versions and flavors of Android. That's an impressive statistic for Google — but terrifying to a developer who wants their app to work on as many devices as possible.

This chapter explores the history of Android versions, and how developers can target as many versions of Android as possible. You'll learn the following:

- What problems Android faces from fragmentation and why they exist.

- What the Android Support Libraries are, and how they reduce the impact of fragmentation.

- How an app you created earlier in the book uses the Support Libraries to be backwards compatible.

Android: An open operating system

To understand where the fragmentation problem came from, it's important to know how Android came to be the most popular operating system on the planet.

Google originally acquired Android by buying a company called **Android Inc** in 2005. Android Inc was a company that saw potential for mobile devices to become smarter than ever before, and Google wanted a piece of the action. Once Android was in Google's hands, they began turning Android from the prototype they bought, into a production-ready operating system.

Meanwhile, Google began to share their vision of the future of mobile with various phone manufacturers such as Samsung, LG and HTC. What Google offered to phone manufacturers was a a stable operating system: one that could be altered to work for a particular manufacturer's needs.

For Google, it was a way to reach users like never before. For the phone manufacturers, it was a way of keeping up with competition. The approach towards openness ultimately convinced phone manufacturers to adopt Android as the operating system for their devices.

When Google publicly announced Android in November 2007, it also announced the creation of the **Open Handset Alliance** (http://www.openhandsetalliance.com), a consortium of phone manufacturers agreeing to work towards a set of open standards for mobile devices. Those standards materialized in the form of Android.

To ensure these standards were openly available, the **Android Open Source Project** (https://source.android.com) was created, which allows anyone to download and contribute to the Android Operating system.

How fragmenting occurs

As years went by, it was logical that devices would eventually need to receive updates for their Android OS. However, many devices didn't receive updates for months at a time. This was because phone manufacturers had to take the time to test that the updated versions of Android were compatible with their own in-house changes they'd made to their particular flavors of Android.

Differences between stock Android and the versions that phone manufacturers included in their operating system varied. Some changes were minor UI tweaks, while others were dramatic changes to underlying components of Android that only worked for particular devices.

If you were to look at the leading Android devices of today, you'd first notice differences in the user interface. If you could dive deeper into the internals of the devices, it's likely you'd find some manufacturer-specific apps and features that you can't remove on your own. Dive deeper still, and it's possible you will find some deeply embedded processes that are unique to a particular phone manufacturer and not part of the core Android operating system.

The delay in Android updates, magnified across multiple manufacturers and devices has led to what tech journalists to declare that Android has a **fragmentation** problem.

Google has made efforts to combat the delay in Android updates getting to devices. Their stock apps are downloadable only from the Google Play store, and are only available on devices whose manufacturers pay Google a licensing fee for their Google Mobile Services suite. Google has even gone so far as to isolate certain parts of Android via a project called **Project Treble** (https://source.android.com/devices/architecture/treble), aiming to abstract away the core of Android and providing interfaces for manufacturers to use in their own Android implementations.

The Nexus and Pixel devices from Google run unmodified versions of Android, often called the "vanilla" version. This means these devices can be quickly updated with the latest version of the operating system without the need to test device-specific modifications.

These are all changes aimed at reducing the time it takes for an Android update to be received by a device. That is great for users, but fragmentation is still a reality and one you must deal with as a developer.

The Android support libraries

To ensure developers were not held back by the delayed Android updates, the engineering teams at Google introduced the **Android Compatibility Library** in 2011. This library aimed to ensure Android was easy to develop for across multiple versions of the operating system.

Since then, the library has grown to encompass a range of libraries that provide backwards compatibility for many Android features and UI components. It has since been renamed as the **Android Support Library**.

Backward compatibility across Android versions is *so* important that Android Studio, by default, uses the Support Libraries in the code it generates. You may or may not have noticed that you've been using the Support Libraries all this time as you've worked through this book.

Take a look through the **ListMaker** app you created in Section II. You'll see the Support Libraries are used throughout and how they come in handy.

The first sign of the Support Libraries can be found in the **build.gradle** file for the app module. Open `build.gradle (Module: app)` and scroll down to the `dependencies` block:

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation "org.jetbrains.kotlin:kotlin-stdlib-
jre7:$kotlin_version"
```

```
    implementation 'com.android.support:appcompat-v7:26.1.0'
    implementation 'com.android.support.constraint:constraint-layout:
1.0.2'
    implementation 'com.android.support:design:26.1.0'
    implementation 'com.android.support:support-v4:26.1.0'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.0'
    androidTestImplementation
'com.android.support.test.espresso:espresso-core:3.0.0'
}
```

The dependencies prefixed with `com.android.support:` are all part of the support library collection, specifically created for backward compatibility of newer features to older versions of Android.

It's thanks to the Support Libraries that **Constraint Layouts** are compatible all the way back to Android Gingerbread. Gingerbread was released in December 2010, while Constraint Layouts were introduced in February 2017. That's an incredible amount of support for old software.

It's obvious that Constraint Layouts are used to build up the Layout for the UI in your app. However, other uses of the other support libraries may not be so apparent.

Open the ListMaker project and then open **MainActivity.kt**. Holding the **Command** button, if you're using a Mac, hover the mouse cursor over the `AppCompatActivity` subclass at the top and left-click it:

> **Note**: For Windows / Linux users, hold Ctrl and left-click.

Android Studio will jump to the **AppCompatActivity.java** class file. `AppCompatActivity` is part of the `com.android.support:appcompat-v7:26.1.0` library. You can tell by package name at the top of the file:

```
package android.support.v7.app;
```

Support packages all begin with the `android.support` prefix as part of the package name, and often include the word `Compat` somewhere in the class name. The most obvious backwards compatible feature `AppCompatActivity` brings is the ease of supplying a **ToolBar** to the top of the Activity.

Head back to **MainActivity.kt**, hold the **Command** button, hover your mouse over `setSupportActionBar(toolbar)` in `onCreate()` and left-click to jump to the implementation.

```java
/**
 * Set a {@link android.widget.Toolbar Toolbar} to act as the
 * {@link android.support.v7.app.ActionBar} for this Activity window.
 *
 * <p>When set to a non-null value the {@link #getActionBar()} method will return
 * an {@link android.support.v7.app.ActionBar} object that can be used to control the given
 * toolbar as if it were a traditional window decor action bar. The toolbar's menu will be
 * populated with the Activity's options menu and the navigation button will be wired through
 * the standard {@link android.R.id#home home} menu select action.</p>
 *
 * <p>In order to use a Toolbar within the Activity's window content the application
 * must not request the window feature
 * {@link android.view.Window#FEATURE_ACTION_BAR FEATURE_SUPPORT_ACTION_BAR}.</p>
 *
 * @param toolbar Toolbar to set as the Activity's action bar, or {@code null} to clear it
 */
public void setSupportActionBar(@Nullable Toolbar toolbar) {
    getDelegate().setSupportActionBar(toolbar);
}
```

Android Studio will take you back in to **AppCompatActivity.java**, the same support class you saw earlier. This is important to note because the **ToolBar** was first introduced in Android Lollipop. Any devices that tried to run an app using a `ToolBar` would crash quickly because the device doesn't know what a `ToolBar` is.

This is where the Support Library and **AppCompatActivity.java** comes in. If a device is running an earlier version of Android that doesn't know what a `ToolBar` is, the Support Library provides the device with the class. This ensures the app functions as intended and developers can rely on using consistent APIs that support earlier versions of Android.

Let's take a look at a few other examples. Open **ListSelectionFragment.kt**, hold the **Command** button, hover the mouse cursor over the `Fragment()` subclass at the top and left-click it.

```
/**
 * Default constructor.  <strong>Every</strong> fragment must have an
 * empty constructor, so it can be instantiated when restoring its
 * activity's state.  It is strongly recommended that subclasses do not
 * have other constructors with parameters, since these constructors
 * will not be called when the fragment is re-instantiated; instead,
 * arguments can be supplied by the caller with {@link #setArguments}
 * and later retrieved by the Fragment with {@link #getArguments}.
 *
 * <p>Applications should generally not implement a constructor. Prefer
 * {@link #onAttach(Context)} instead. It is the first place application code can run where
 * the fragment is ready to be used — the point where the fragment is actually associated with
 * its context. Some applications may also want to implement {@link #onInflate} to retrieve
 * attributes from a layout resource, although note this happens when the fragment is attached.
 */
public Fragment() {
}
```

Android Studio will open up **Fragment.java**, the class definition for the Fragment. Scroll to the top of the class and take note of the package:

```
package android.support.v4.app;
```

Even Fragments exist in the Support Library! Although Fragments allow your UI to provide flexibility depending on the screen of a device, they were introduced in Android's Honeycomb release.

Thanks to the Support Library implementation, Fragments can now be used all the way back to Android Donut: a version of Android that was released *two years* prior to Fragments being introduced.

Head back to **ListSelectionFragment.kt** and **Command + left-click** over the `RecyclerView` defined at the top of the class.

```
 * When writing a {@link LayoutManager} you almost always want to use layout positions whereas whe
 * writing an {@link Adapter}, you probably want to use adapter positions.
 *
 * @attr ref android.support.v7.recyclerview.R.styleable#RecyclerView_layoutManager
 */
public class RecyclerView extends ViewGroup implements ScrollingView, NestedScrollingChild2 {
```

Android Studio will show you **RecyclerView.java**: the class definition for a `RecyclerView`. Scroll to the top of the class and inspect the package name:

```
package android.support.v7.widget;
```

Another support library component you've used without knowing! `RecyclerView` was first introduced to Android in 2014 with Android Lollipop. However, instead of bundling

`RecyclerView` into the Lollipop update, Google Engineers decided to put it straight into the Support Library as they had recognized how integral the libraries had become.

This decision meant `RecyclerViews` were not released in a particular version of Android. As part of the Support Libraries, they became a crucial element of UI that are backwards-compatible all the way back to Android Eclair, released in 2009.

Reducing the impact of fragmentation in your app

Although fragmentation is a real problem for Android, the engineering teams at Google have provided a way for developers to avoid its effects, ensuring your apps can reach as many users and devices as possible.

While not every single feature can be backported, the most important ones that provide consistency for the user experience are there for you to use.

Above all, use the Support Libraries whenever possible. Even if you don't think you need them, assume that your first user will use your app on the oldest version of Android possible. Optimizing for the worst experience means you're giving your users the best experience you can — on whatever device they're using!

Where to go from here?

The Support Libraries are an integral part of Android development, for without them, development across multiple Androids versions would be incredibly painful.

- For more information on how to use the Support Libraries, visit the Support Library page on the developer website at https://developer.android.com/topic/libraries/support-library/index.html.

- You can find a list of all the features the Support Libraries provide at https://developer.android.com/topic/libraries/support-library/features.html.

- Finally, you can find a list of all the Support Library dependencies you can include in your apps at https://developer.android.com/topic/libraries/support-library/packages.html.

Chapter 29: Keeping Your App Up To Date

By Darryl Bayliss

Building a great app requires hard work and determination. Continually updating your app requires not just a firm belief in the original vision, but the discipline to evolve your app as time passes. Overnight success is a rare thing. Instead, it's more likely that a trickle of users will download the app; some will uninstall it a few minutes later; and a very few will genuinely find your app useful and use it regularly, perhaps even leaving reviews. This last group contains the users you owe your attention and commitment to.

The more you commit to your app, the more value your users will see in the product. Keeping your app up-to-date is an incentive to growing that important group of users. Publishing an app is an achievement, but supporting an app over the years to come is an even greater achievement.

This chapter covers what you need to know when it's time to update your app, including:

- How to leverage data from Google to target what you should update

- How to target the latest version of Android, including preview releases

- How to decide when to drop support for older versions of Android

Following Android trends

Data that can help you make an informed decision can be invaluable in helping you make the most of your development time and money.

There are two sources you can draw on for high quality data. The first option is the **Google Play Console** (https://developer.android.com/distribute/console/features.html).

Besides providing a portal for app distribution, the console provides useful metrics about devices that have downloaded your app. This includes the device type and version of Android your users are running.

You'll dive deeper into the Google Play Console in the following chapters when you look at deploying your app, but what you need to know is that it can be a great source of information when deciding on how best to keep your app up-to-date.

If you require less targeted data and prefer a snapshot of the whole distribution of Android devices in the world, Google offers a number of dashboards at (https://developer.android.com/about/dashboards/index.html) that detail key metrics:

- Android versions

- Screen size and density

- OpenGL versions

This information is generated from devices that visited the Google Play Store within the last seven weeks, so you can rely on the dashboards to provide an accurate portrait of Android within the Google ecosystem.

Choosing the right platform to target can lead to more engaging apps if you're not preoccupied trying to backport features or trying to fallback gracefully. Looking at this dashboard can help you decide to leave versions in the dust. We'll explore this idea later on in this section.
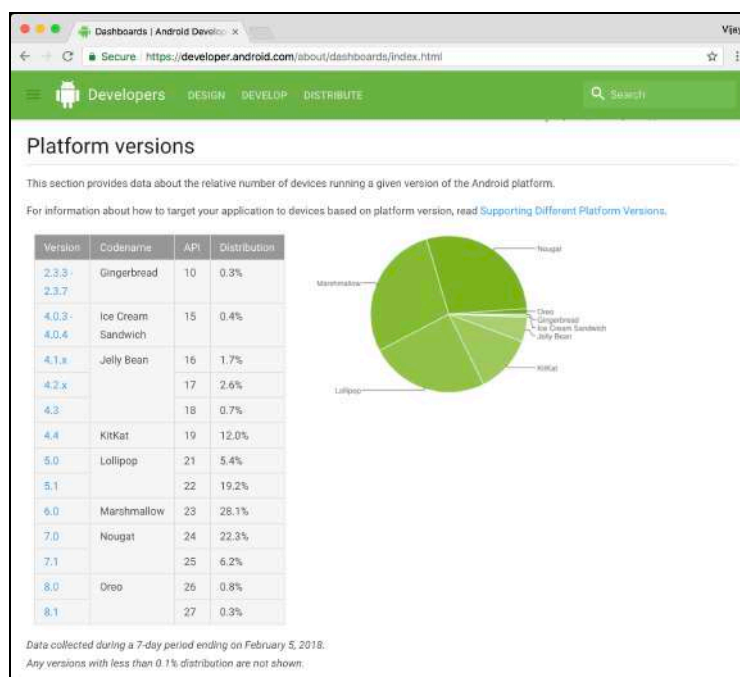
Keeping an eye on the most common screen sizes and densities can help you decide to not consider some sizes when designing your user interface. Keeping up with the latest trends here can help you shed unneeded assets and keep your APK slim.
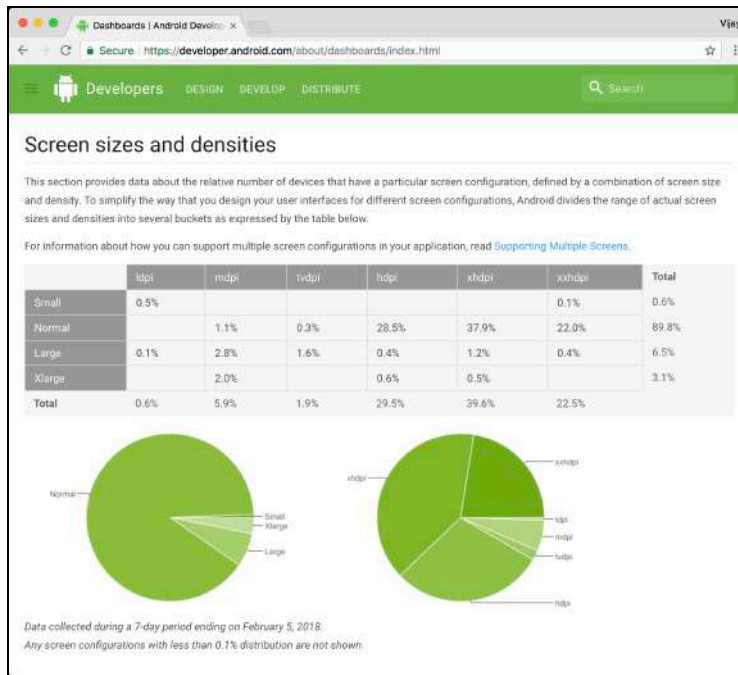
OpenGL may not be common in all apps, however, but it carries the same considerations you may have when picking and choosing platform versions or screen sizes to support. Newer versions of OpenGL contain some cool features you might need in your games, that you may have put off because not enough devices have said version. However, keeping an eye on this dashboard may be enough motivation to consider increasing the minimum bar on OpenGL support.

How you use the data Google provides to focus your development efforts depends entirely on your personal goals for your app and, ultimately, your business. Once you've decided what versions of Android to support, you'll have to decide if it's worth adding support for newer versions of Android, and whether it makes send to drop support for older versions. In the next section, you'll see what it means to keep up with the latest version of Android.

Managing Android updates

As a good developer, you want to make sure your app runs on the latest and greatest version of Android. Major updates to the Android OS occur on a yearly cycle and are announced at **Google IO** (https://events.google.com/io/), Google's developer conference, where a range of new products and services across the company are showcased.

Google also regularly releases minor updates to Android, containing everything from under-the-hood bug fixes, to entire new Android libraries for you to use.

The best way to be notified on any upcoming Android updates is to regularly check the **Android Developers Blog** (https://android-developers.googleblog.com). It's updated regularly and has lots of information about the current and future direction of Android.

Google also allows developers to download preview releases of upcoming Android versions. This gives developers a chance to iron out any issues their apps might have with the new versions before the new OS is released to the general public. Nothing is more disheartening for users than updating their devices to the latest Android release, only to find out their favorite app doesn't work.

Developers are notified of opportunities to install and test a preview release of Android through the developer blog (https://android-developers.googleblog.com/2017/11/final-preview-of-android-81-now.html), or through the developer documentation provided on the **Android Developer Website**.

Android Developer Previews are available a few weeks in advance of public release. For major updates, Google extends this to a few months, which gives you plenty of time to test your app. It also gives you an opportunity to give Google engineers feedback on any issues or bugs you find as you work with the preview version of Android.

Although you should make an effort to update your app to support the latest release of Android, it *might* not the end of the world if you don't. The engineers at Google have done some excellent work in making various libraries on Android backwards-compatible, which just might cover you for a few releases.

The takeaway here is to keep on top of new Android releases and how the update may or may not affect your app. Knowing what's coming down the pipe lets you adjust your development ahead of time — or even make the call to not update your app at all.

Working with older versions of Android

Although there is a lot of support in Android for backwards compatibility, sometimes it makes sense to break with old versions of Android and only develop for the newer versions. This is a good strategy in some cases — but it comes at a cost.

Using newer APIs means you expect a minimum version of Android in order for your app to run. If the API you're targeting doesn't exist on a older version of Android, then your app won't appear in the Google Play Store for devices with older versions of Android.

This is where you, as the developer, need to decide how to support older versions of Android. Fortunately, you have several options.

The bleeding edge approach

The first option is to be ruthless and only support the versions of Android that your app absolutely needs. This means your app is guaranteed to work, and you don't need to consider any backward compatibility for Android versions that don't support your target API. Moreover, you free yourself from having develop and test workarounds for devices that don't have the APIs you want. It sounds like a developer's dream, doesn't it?

The downside is that you'll shut out huge numbers of potential users with older devices — users who might still want to download your app and spread the word about it! That's one of the realities of dealing with fragmentation in the Android world.

The soft decline approach

The second option is to engineer your app so that it degrades gracefully for older versions of Android: newer Android users get the benefit of all your app's features, while older Android users can still use your app with some functional limitations. This means you keep the market open for your app, and you don't penalize users on older devices.

The downside is that this approach takes more development effort on your part, as you need to consider how the app will react on older devices, and whether the app will still function as you intended on older versions of Android.

The backport approach

The third option is to to rely on backported features. This involves leveraging third-party libraries or support code you write yourself to support features that older devices wouldn't normally have. This is the argument Google uses for persuading developers to use the Android Support Libraries, and many third-party libraries backport their features for the very same reason. The benefits of supporting as many Android users as possible can't be overstated.

The downside in this case is that you'll need to take the time to learn how to use these libraries in your app, or even roll your own code when there's no clear way to support your app with Google's or other third-person libraries.

Where to go from here?

The decision to drop older versions of Android, or to invest the time to support them, depends entirely on the kind of app you make, what your user base looks like, and the amount of effort you want to put into app development.

Think about the future direction of Android, think about what your users want from your app, think about your personal and business goals for your app and let that drive your choice on which approach to use. Supporting apps as new versions of Android roll out of Google is the ultimate test of a developer's commitment. Whether to stay up-to-date with new Android versions, or to drop support for older ones, is an important and difficult choice for any developer. Regular updates show users that your app is being actively developed and supported, which also bodes well for the adoption rate of your app. Leaving your app to stagnate sends a sign that you've abandoned development of the app, and users won't hesitate to look for another solution in the Play Store.

Section VI: Publishing your App

Now that you've created your app, you need to get it out to the world! This section has two chapters that teach you how to prepare your app for release, how to test your app, and how to publish your app to your waiting fans!

Chapter 30: Preparing for Release

Chapter 31: Testing and Publishing

Chapter 30: Preparing for Release

By Tom Blankenship

So you finally built that app you've been dreaming about. Now it's time to share it with the world! But where do you start?

This chapter will help you get your app ready for release. Although this chapter will focus primarily on preparing the app for the Google Play store, most of the steps will apply regardless of the publishing platform.

Here's a quick overview of each step involved:

1. Clean up any debugging code you may have in the source.

2. Check the app version information.

3. Create a release version of the app with the correct signing key.

4. Test the release version on as many devices as possible.

5. Create a Google Play Console developer account.

6. Create screenshots, promotional graphics and videos.

7. Fill out the application details on the play console.

Let's walk through these items in detail.

Code cleanup

The first step is to make sure your project and code are ready for release. Here are a few items to consider:

- Choose a good package name. Once you submit an app to the store, you cannot change the package name. The package name is embedded in **AndroidManifest.xml** but can be set in your app's **build.gradle**.

  The package name must be unique from all other apps in the Play store. One of the best ways to ensure this is to use a reverse naming convention based on a domain name that you own. For example, the **PodPlay** app published by **raywenderlich.com** has a package name of **com.raywenderlich.podplay**.

  ```
  defaultConfig {
      applicationId "com.raywenderlich.podplay"
      ...
  }
  ```

- Turn off debugging for release builds. By default, Android Studio creates debug and release build types for new projects.

  For the release build type, debugging is disabled by default. You can verify this by looking at **app.gradle** in the **buildTypes** section. Check that it has the following definition for the release build type:

  ```
  buildTypes {
    release {
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android.txt'),
            'proguard-rules.pro'
    }
  }
  ```

  If you have a `debuggable true` line in the release build type, remove it.

  `minifyEnabled` enables **ProGuard**. ProGuard is a tool that helps shrink your code for release. It removes unused code and libraries, and it obfuscates class, property and method names.

- Remove logging by deleting **Log** calls in the code or let ProGuard remove the calls during the release build.

To use ProGuard, add the following lines to **proguard-rules.pro** in the root of your project:

```
-assumenosideeffects class android.util.Log {
    public static boolean isLoggable(java.lang.String, int);
    public static int v(...);
    public static int d(...);
    public static int i(...);
}
```

This removes verbose, debug and information log calls, but it leaves warnings and errors. Make sure that any remaining warning or error messages do not log any personal data.

- Verify production settings. If your app communicates with external services, has update URLs, API keys or other configuration items that are different during development, change them to the proper production settings.

- Check for stray files in your project. Look inside **src** to make sure it contains only source files. Check **assets** and **res** for outdated raw files, drawables, layouts and other items. If found, remove them from the project.

- Perform any final localization tasks such as translating your string files to other languages.

Versioning information

Before releasing the app, make sure you have a good versioning strategy. This is critical to maintaining the app and keeping a handle on support issues that may arise.

Users should be able to identify the version number and trace it back to a specific source code snapshot; this helps with debugging.

The best place to specify your app version is in the **app.gradle** build file. Two primary settings control versioning: **versionCode** and **versionName**. These are normally located in the **defaultConfig** section as shown below:

```
defaultConfig {
    applicationId "com.raywenderlich.podplay"
    minSdkVersion 19
    targetSdkVersion 26
    versionCode 1
    versionName "1.0"
    testInstrumentationRunner
"android.support.test.runner.AndroidJUnitRunner"
}
```

- **versionCode**: This is the internal version number, which the user cannot see. It's an integer value, and you should increase it with each new build you upload to the play store. The play store uses this number to determine if one build is older than another; it will not allow installs that downgrade to an older version.

- **versionName**: This is the external version number visible to the user. You have full control over how it's formatted. Most apps use a *major.minor.point* release format for `versionName`. The key is to have a consistent formatting convention. Just don't forget to update the string with each new release.

> **Note**: The major.minor.point release scheme is often referred to as Semantic Versioning. For more information on this scheme, check out https://semver.org/.

Build release version

Each time you build and run your app during development, Android Studio produces an APK file and installs it on the emulator or device. This APK file contains your app's executable code as well as all of its resources.

When using the default debug build type, the APK produced is signed with a debug key, which is automatically generated by Android Studio. This debug APK also has a special **debuggable** flag set and includes extra information to make debugging easier.

You can't submit an APK built for debugging to the play store because Google won't allow it. Also, you should not distribute it directly to users.

To make sure the **debuggable** flag is not set, and to have Android Studio build an optimized **Release** version of the APK, use the Release build type. Just like the debug version, the release APK must be signed, but in this case, it should be with your own private signing key.

Create a signing key

Your first step in building a release version is to generate a signing key, which you'll use to sign the app. This key is stored in a keystore file, and any future versions of the same app must be signed with the same key.

This key is critical to the security of your app. It should always be kept private and in a safe place. If you lose the keystore, you won't be able to release a new version of your app under the same package name!

> **Note**: Google has a Google Play App Signing feature. This service lets Google manage your signing key, giving you some options if you lose your key or it gets compromised. When using this method, you'll sign the app with an **Upload Key**, and then Google will resign the app with your actual app signing key. This will be covered more in the next chapter, but you can learn more here: https://developer.android.com/studio/publish/app-signing.html#google-play-app-signing.

Use the following steps inside Android Studio to create your signing key:

1. Click **Build ‣ Generate signed APK…** from the menu.

2. Select the module (usually "app"), and click **Next**.
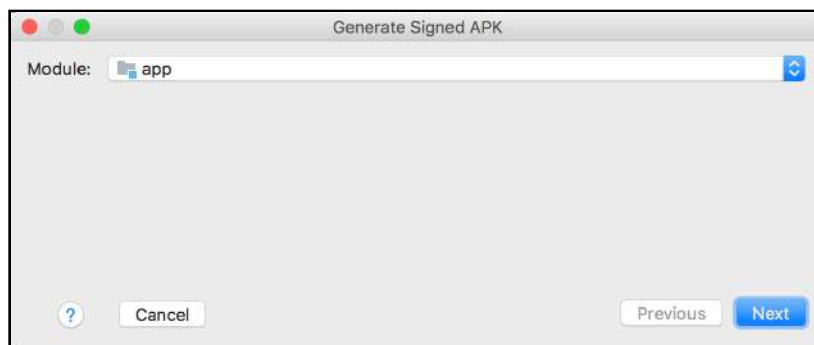
3. Select **Create new…** to create a new keystore. A keystore can hold multiple signing keys, with each one referred to by an alias name.
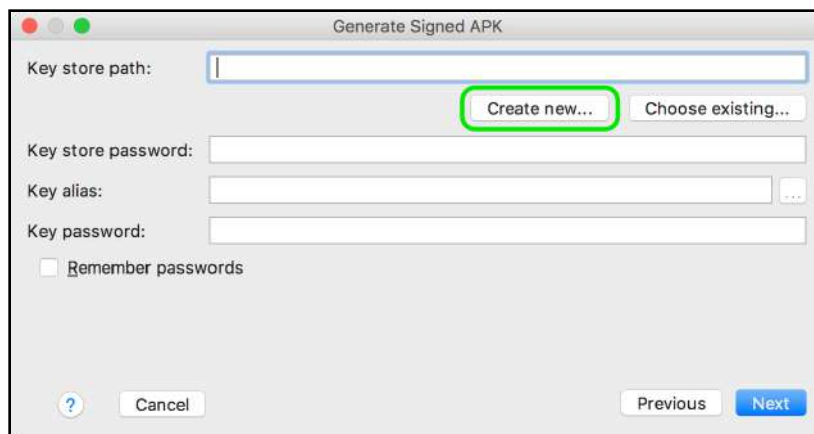
4. The "New Key Store" dialog appears.

5. Select the **Key store path** where you want to store the file. There's no standard extension required for this file; however, most people use **.jks**.

6. Fill in the keystore **Password** and repeat it in the **Confirm** field. Make sure to store this password safely, because you'll need it whenever you access the keystore.

7. Fill in the following items for the **Key**:

   Alias: Enter a name for the key. Usually the name of your application.

   Password: Enter a password to protect the keystore file.

   Confirm: Repeat your password.

   Validity (years): Leave this at 25 years. The key will expire after this time.

   Certificate: Enter your personal information in these fields. The user won't see your data, but it will be part of the signing certificate in the APK file.
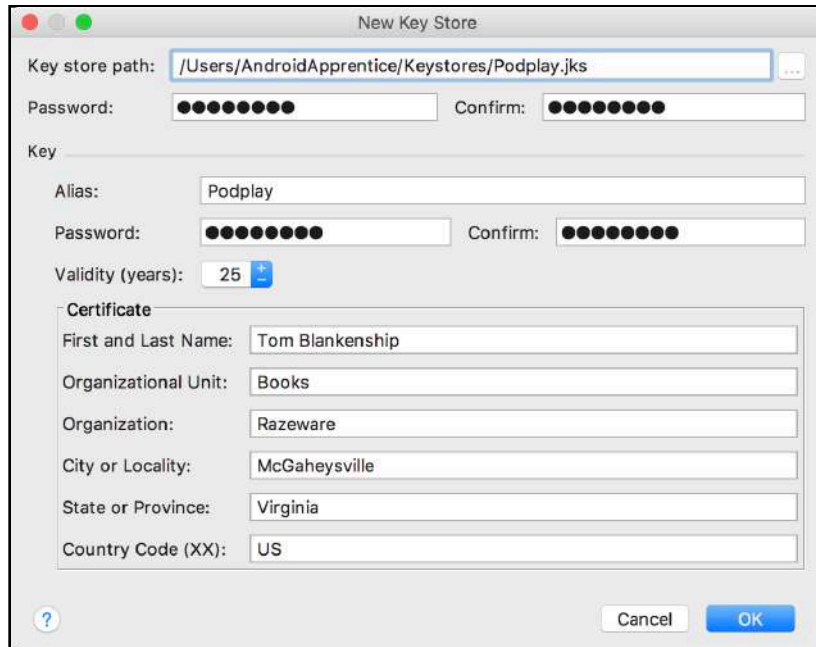
8. Click **OK**, and the original dialog, with the values already populated, will appear.
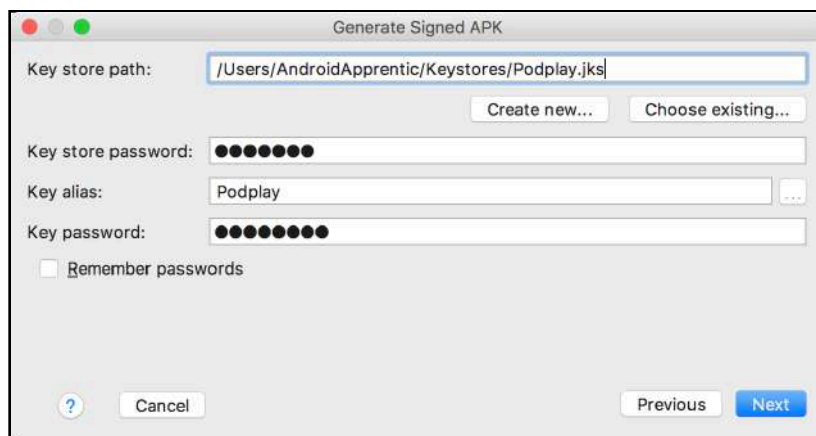
9. If don't want to enter passwords each time you build a release version, check **Remember passwords**.

10. Click **Next**.

11. Fill in the **APK Destination Folder**. Normally, this a folder outside of your main project folder.

    Be sure to select both **V1** and **V2** signatures, so your apps can be installed on older and newer device.

    The **Build Type** should be set to **release**.
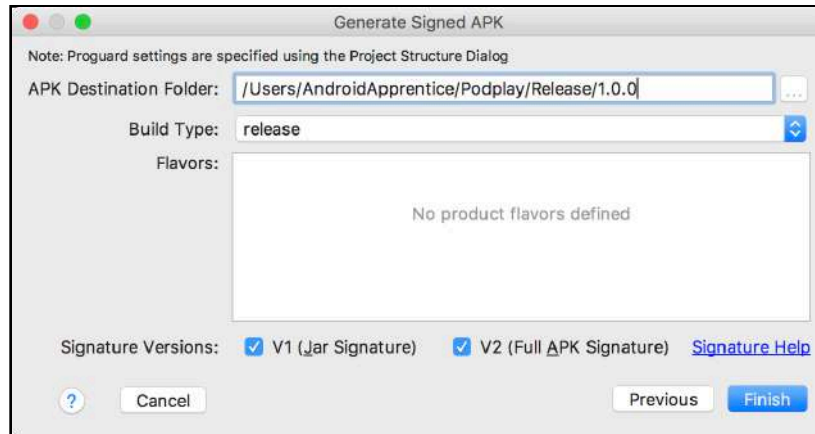
12. Click **Finish**.

Android Studio builds and signs the release APK file and places it in the destination folder. A popup will display in the bottom right corner of Android Studio when the build is complete.

The final output file is named **app-release.apk**.

You'll follow these same steps each time you build a release version. However, you can skip steps 3-7 since you already created the keystore and key.

> **Note**: It's worth mentioning one more time that it's critical that you keep your release keystore secure! If someone else gets a hold of your key, they can do all sorts of damage, such as possibly distributing malicious apps under your identity.

Check file size

Check the size of the APK file. If it's over 100MB, you won't be able to publish it as-is to the Play store. You can get around this limitation by using expansion files. This is not an issue for most applications, but if you find yourself with a large APK file, you can find details about using expansion files here: https://developer.android.com/google/play/expansion-files.html

Release testing

Test the release file on as many devices as you possibly can. Subtle bugs can show up when running the release vs. debug versions of your app, especially when running on different hardware devices. At a minimum, you'll want to test on at least one phone and one tablet.

You can use the following `adb` command to install the release APK onto a device:

```
adb -s [DeviceId] install app-release.apk
```

Google Play Store

Now that your release APK is ready, it's time to go over the steps to create a Google Play store listing.

Google Play Console signup

The first step is to sign up for a **Google Play Console** account. The Google Play Console is your gateway to managing and publishing your apps on the Google Play store.

Go here to sign in or sign up for a new Google Play Console account:

https://play.google.com/apps/publish/

Verify that you're signed in with the correct account first. Read and agree to the developer agreement, and then click **CONTINUE TO PAYMENT**. The current registration fee is $25, and it only has to be paid once.

After you finish the payment, you're taken to the Developer Profile screen. Make sure you pick a good **Developer name** as it will be shown in the Play Store below the name of your application.

The main console

Once you're finished with signup, you'll get to the main console.
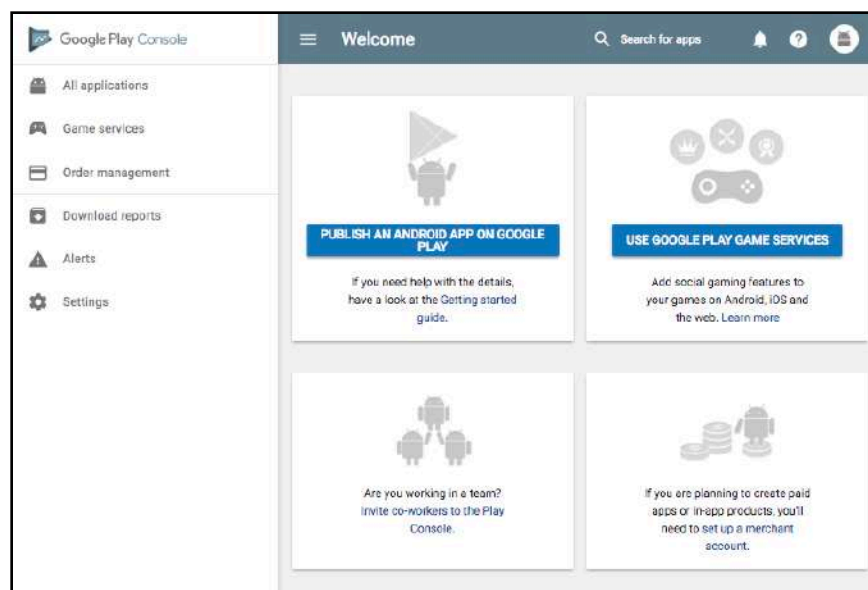
You have the following options on the left:

- **All Applications**: This is where you add new applications or manage existing ones.

- **Game Services**: This provides a lot of additional features for games. You can find more info here: https://developers.google.com/games/services/.

- **Order Management**: If you have a paid app or in-app purchases, you can manage orders here, including giving refunds.

- **Download Reports**: This section provides a variety of reports, including crashes, reviews, statistics, user acquisition and financial records.

- **Alerts**: Here you can see any alerts generated by the Play store for your apps.

- **Settings**: This provides several sub-sections.

- **Developer Account**: You can manage profile settings, add other console users, control API access and set up payment options.

- **Developer Page**: Here's where you can configure how your developer page looks in the Play store. Your developer page won't be available until you publish your first app.

- **Manage Testers**: You can manage alpha and beta testers from this section.

- **Preferences**: This is where you set notification preferences and control privacy settings.

Creating your first app

To get started, click the **PUBLISH AN ANDROID APP ON GOOGLE PLAY** button on the main console screen.

> **Note**: At this point, you're just preparing the store listing and creating a draft version of the application; nothing gets published until you use the Publish step.

First, fill in the title of your app and click **CREATE**.

This will create the application and present you with several pages of information related to the app.

The first page you'll see is the Store listing. Here's a partial view of this page:
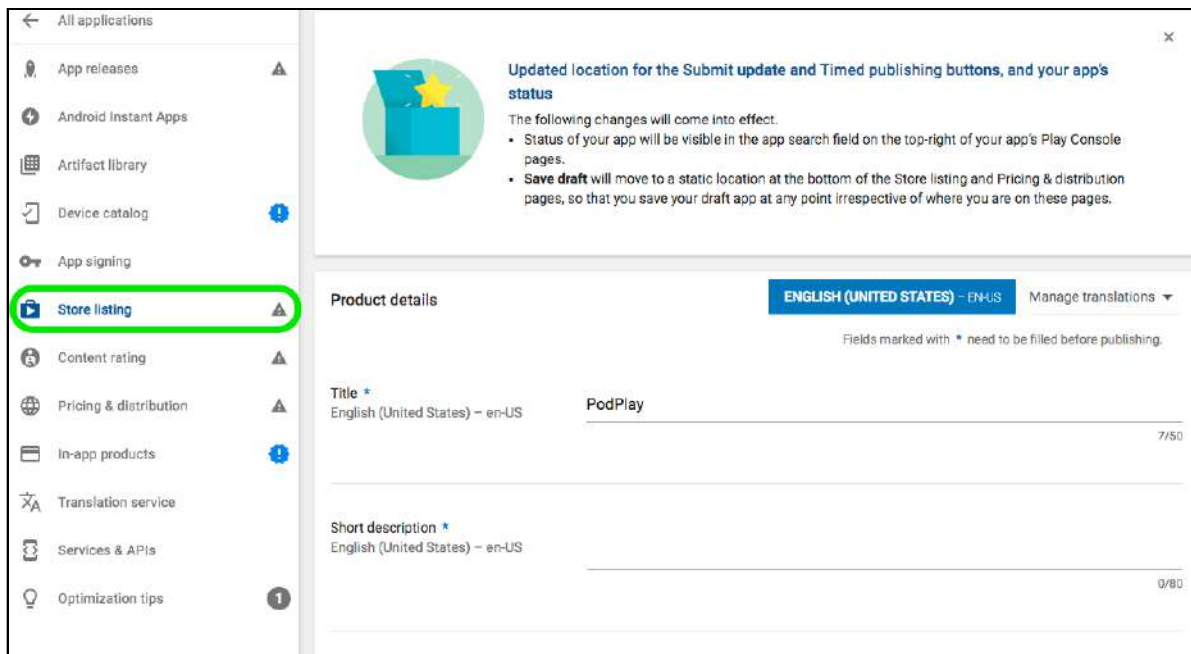
Go back to the home console screen, and you'll see the new application you just added, with a status of **Draft**.

Take a look at the left side of the screen. The items with exclamation points to the right represent the things you must complete before publishing the app.
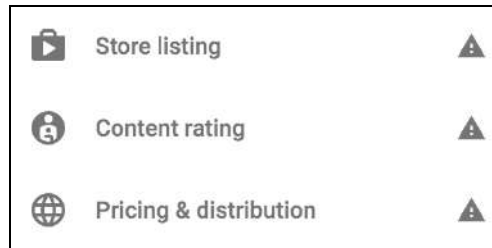
You'll start with the Store Listing first, but before you can begin, you'll need to gather a few items.

Store graphic assets

There are some graphic assets your app is expected to have. They are:

- **Screenshots**: You're required to upload at least two screenshots, although you can have up to eight per device type. The size of a screenshot has to be at least 320px on the shortest side and no longer than 3840px on the longest side. You can upload portrait or landscape orientation screenshots.

> **Note**: You can create screenshots from the emulator by using the camera icon on the emulator toolbar.

- **High-res icon**: A high-res icon is required with a size of 512px x 512px. This gets displayed in the play store only. Your app's launcher icon is still shown on the user's device.

- **Featured graphic**: The featured graphic is required and should be 1024px by 500px. It's shown at the top of your app listing.

Privacy policy

If your app requests access to sensitive information or is in the **Designed for Families** program, you must provide a link to a privacy policy. This privacy policy must discuss specific privacy policies related to the app.

Determining content rating

Google provides a questionnaire that you must complete to determine your app rating. Click **Content rating** on the left, and click **CONTINUE** to start the survey.

You're required to provide an email address for the International Age Rating Coalition (IARC). This can be the same as your primary Play Store email.

Next, select from one of the primary categories:

- Reference, News, Or Educational

- Social Networking, Forums, Blogs, And Ugc Sharing

- Content Aggregators, Consumer Stores, Or Commercial Streaming Services

- Game

- Entertainment

- Utility, Productivity, Communication, Or Other

Next, you'll walk through a series of Yes/No questions.

Click the **Learn More** link for questions you're not sure about.

After answering all of the questions, click **SAVE QUESTIONNAIRE** and then **CALCULATE RATING**.

The calculated rating is shown for different countries and regions in the world. Here's the rating for the PodPlay app after selecting "No" to all questions.

Click **APPLY RATING** to apply the rating to the store listing.

The **Content Rating** section on the left will show a green checkmark.

Pricing and distribution

You also need to provide pricing information and specify where your app will be distributed.

Click the **Pricing & Distribution** link.

By default, the app will be set to FREE. If you plan on charging for the app, you'll need to set up a Google merchant account first. The details for that won't be covered here, but setting up a merchant account involves filling out details about your business, and providing Google with a bank account in which to deposit payments.

Next, determine the countries in which you want to make the app available. You can enable individual countries, or you can allow them all by selecting the toggle at the top of the list. The next screenshot shows that PodPlay is available in 142 countries after making them all available.
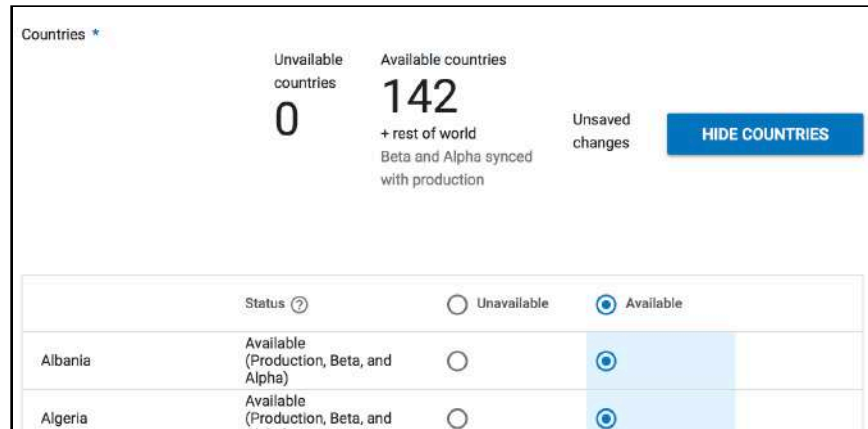
There are several items required on this page:

- **Primarily Child-Directed**: If this is set on, you must opt-in to the **Designed for Families** program.

- **Contains ads**: If this is set on, users will see a **Contains ads** label on the application.

- **Content guidelines**: You must agree to follow the Android Content Guidelines.

- **US export laws**: You must agree to comply with US export laws.

Several more optional items are shown on the page. You can read through these items to see if any of them apply to your app.

Click **SAVE DRAFT**. If you've completed everything, the **Pricing & Distribution** section will show a green checkmark.

The only item remaining is to complete the Store listing section.

Store listing

Click **Store listing** and fill out the following required items:

- **Short description**: Up to 80 characters. Mention the most important feature of your app and explain why a user would want to install it. Think of this as the app promotional text.

- **Full Description**: Up to 4000 characters. Provide the full benefits and features of your app. Use keywords in the description that users are likely to use when searching for an app like yours.

List out the main features one-by-one and highlight the most important ones. You can use rich formatting in your description, but some of it may only appear in the Google Play store app. This includes URL links, UTF-8 characters and Emojis.

- **Screenshots**: Drag your screenshots to the appropriate device tabs.
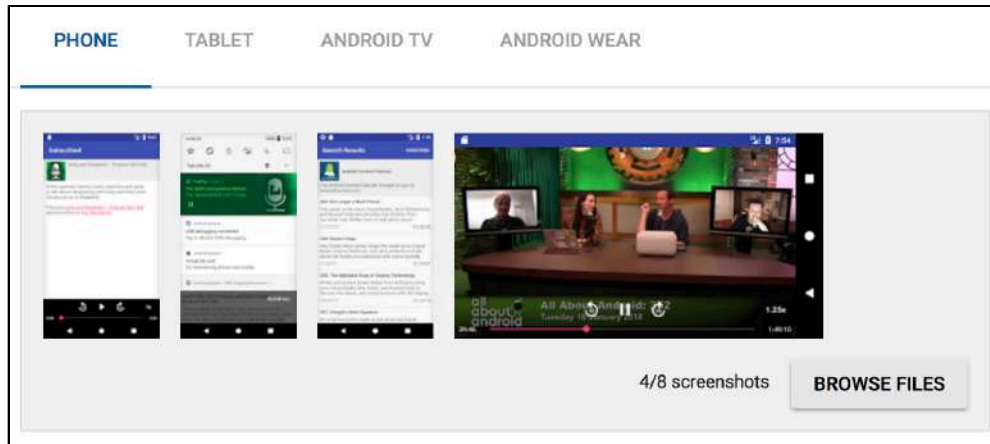
- **High-res icon**: Drag your high-res icon into place.

- **Feature graphic**: Drag your feature graphic into place.

- **Application Type**: Choose between application or game.

- **Category**: Select the category that best matches your app. **Music & Audio** was chosen for the PodPlay app.

- **Contact Details**: Check your contact details to make sure they're accurate. This information is displayed on the app page.

- **Private Policy**: Enter the URL for your privacy policy if required by your app.

Click **SAVE DRAFT**. If everything checks out, the **Store listing** section will show a green checkmark along with the other two sections you have already completed.
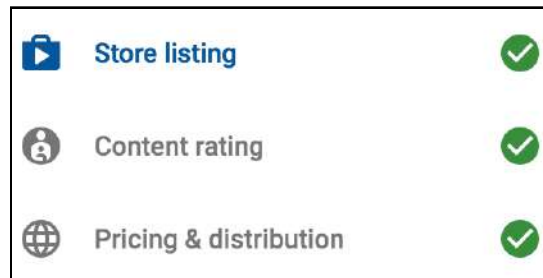
Where to go from here?

Congratulations, most of the hard work is done! All that's left is to create a new app release and upload your signed APK file. You'll cover this and the publishing step in the next chapter.

Take some time and go through all the menu items of the play console. You'll discover that Google provides developers with tons of tools to help apps succeed once they're in the Play store.

You should also check out the YouTube video *Use Android Vitals in the Google Play Console to Understand and Improve Your App's Performance* from Google I/O 2017. Members of the Google Play team go over some of the fantastic tools available to developers.

Chapter 31: Testing & Publishing

By Tom Blankenship

In this chapter, you'll complete the app publishing process and discover additional ways to distribute your app. You'll also go through the Alpha and Beta testing process to make sure your app is ready to share with the world.

> **Note**: You don't have to release your app through Alpha and Beta channels. You're free to take your initial release straight to production!

Release types

Google provides three different release types: **Alpha**, **Beta** and **Production**.

The Alpha and Beta release types provide an excellent way to get feedback to help make sure your final release is as polished and stable as possible. The only requirement for testers is an Android device and an **@gmail** or **G Suite** account.

Time to dive into the details of each release type as well as **open** vs. **closed** testing.

Alpha release

You'll start by creating an Alpha release. This release is typically done with a small group of internal testers. An Alpha release may not be stable yet, but it still needs to be tested in release mode on real devices.

Bring up the main Google Play console website and follow these steps:

1. Click **App releases** on the left side of the page. The list of **Production**, **Beta** and **Alpha** release types are shown.
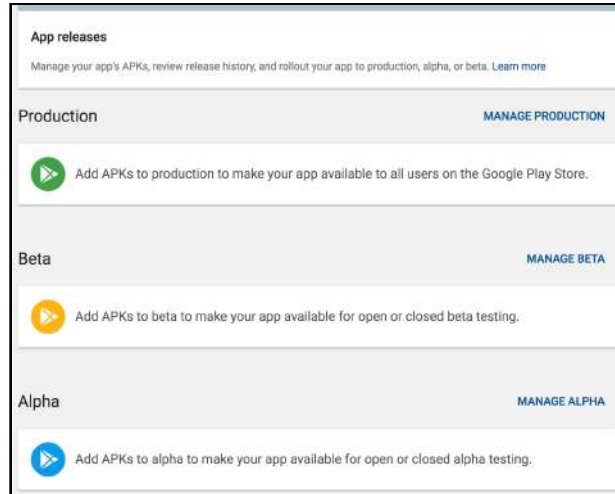
2. Click **MANAGE ALPHA**.

   This is where you can create a new Alpha release, upload the APK and manage the testers.
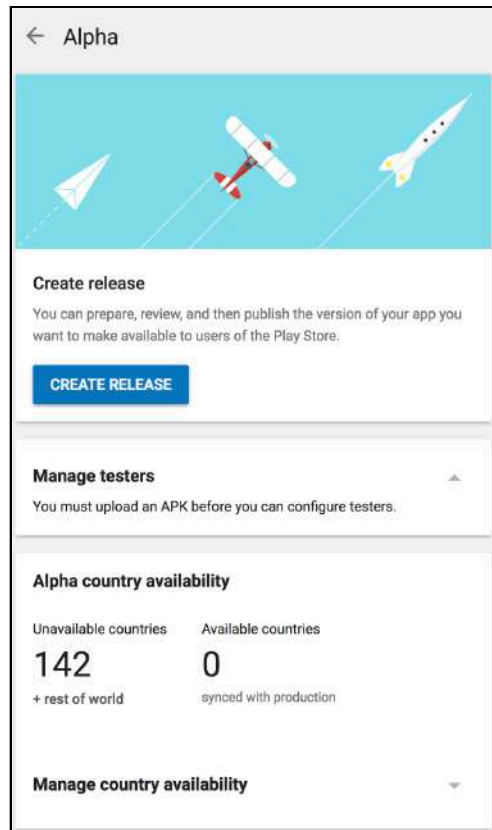
3. Click **CREATE RELEASE**.

    At this point, if you don't already have it set up, you'll be given an option to start using Google Play App Signing. For more information about app signing, refer to Chapter 10, "Preparing for Release."

    When using Google Play App Signing, your app gets signed locally with an **upload key**. After you upload the app, Google replaces the upload key with the actual app signing key. The upload key's only purpose is to authenticate you as the developer.

    The advantage to using Google Play App Signing is that even if you lose your upload key, or it's stolen, you can request that Google revoke the key, and then you can generate a new one. This puts the burden on Google to securely maintain your app signing key.

    Using Google Play App Signing during this phase makes it much easier than turning it on later. Google will automatically generate an app signing key and store it; all you have to do is upload the APK you already signed. The key you generated earlier becomes your upload key. If you decide to enable Google Play App Signing later, you'll have to go through several more steps to make the switch.

    > **Note**: If you click **CONTINUE** to opt-in, you'll be **permanently** enrolled in Google App Play Signing for this app.
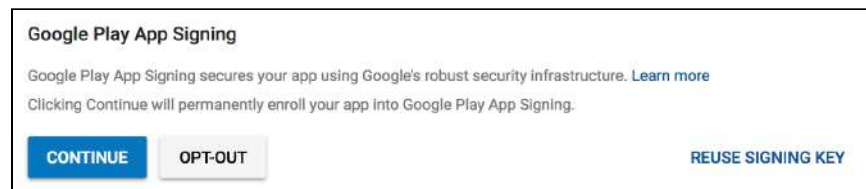
4. Click **CONTINUE** if you'd like to enroll in Google Play App Signing, and **ACCEPT** the terms of service.

    Google generates an app signing key and shows that Google Play App Signing is enabled.
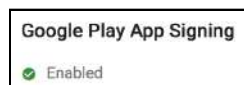
5. Click **BROWSE FILES** and select your signed release APK file.
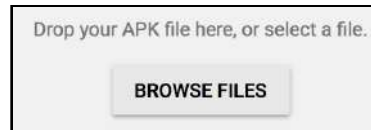
6. This shows the **versionCode** for the app, taken from the setting in the app gradle file. End users won't see this code. You can also specify a **Release name** so it's easier to identify in the play console. By default, it's the same as the **versionName** in the app gradle file.
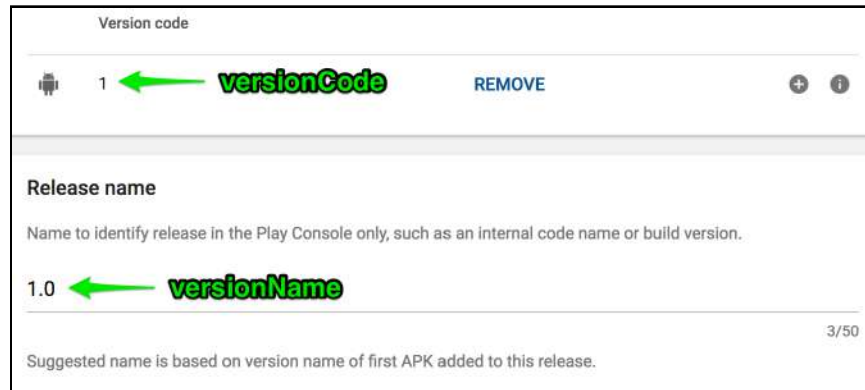
7. Enter the release notes for this version. Make sure to place the notes within the language tags as shown in the template.
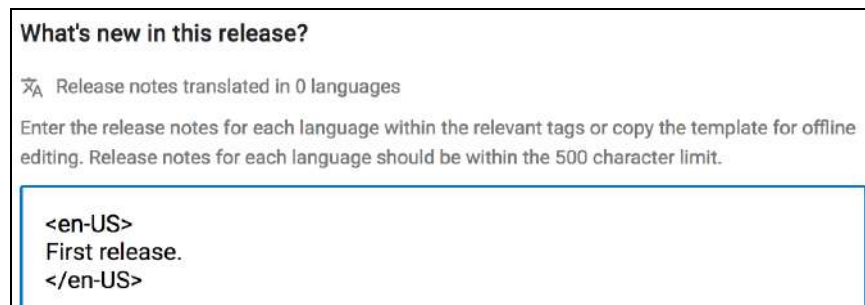
8. Click **SAVE**. Google validates what you've entered and then enables the **REVIEW** button.

9. Click **REVIEW**. This shows a summary of the release and a warning that you need to add users before you can roll it out.
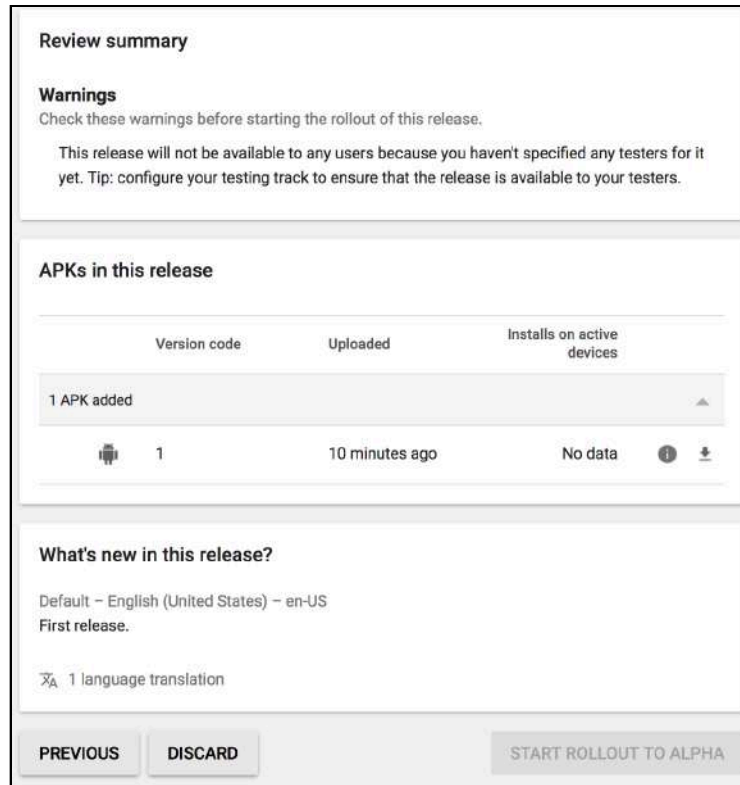
10. Click the back arrow to go back to the main Alpha release screen.
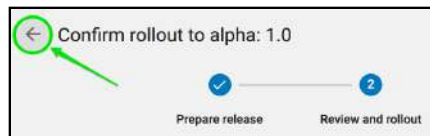
You'll see the **Manage testers** section at the top.
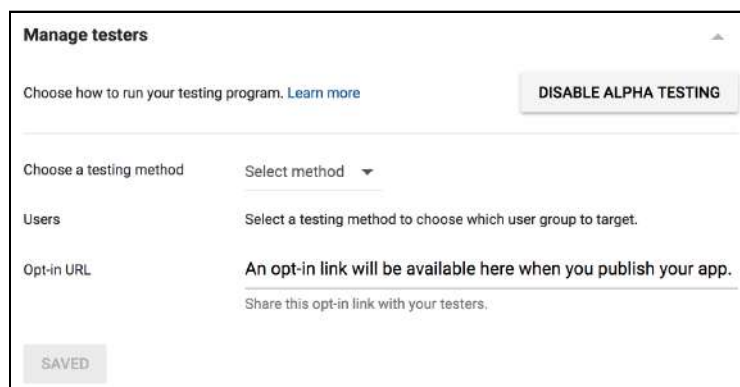
11. Select either **Open**, **Closed** or **Google Groups or Google+ Communities** testing method.

Open: Allow anyone with a link to access the release. Keep in mind that running an open test means that anyone can find your app in the Play store. If you run an open test, make sure your Play Store listing is ready for viewing.

Closed: Only allow a specific list of email accounts to access the release. It will not show up in searches of the Play Store.

Google Groups: Allow anyone within a Google Group to access the release. It also will not show up in searches of the Play Store.

If you select the **Open** method, you can optionally specify the maximum number of users allowed in the test group. You can also provide a feedback page URL that is visible on the app's release page.
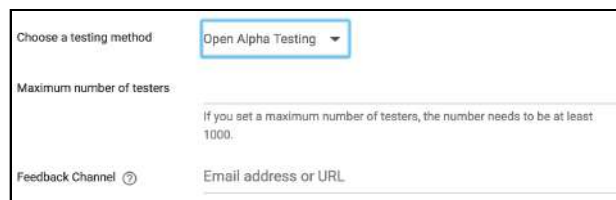
If you select the **Closed** method, you'll need to supply a list of users and email addresses. You can also provide a feedback page URL.
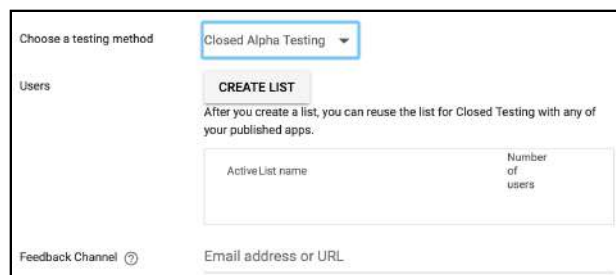
If you select the **Google Groups** method, you'll need to supply the Google Group email or Google+ Community URL.
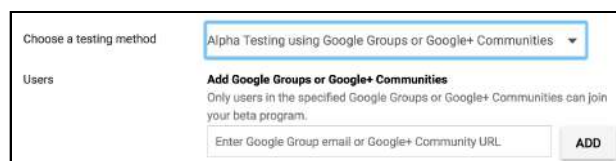
12. Fill in the required fields and click **SAVE**.

    If everything checks out, the **App releases** row on the left will show a green check, and all yellow triangles will be gone. The top of the page indicates that your app is "Ready to publish".

    Now you're ready to roll out the Alpha release to your testers.

13. Tap the **App releases** link on the left.

14. Tap the **EDIT RELEASE** button under the Alpha release section.
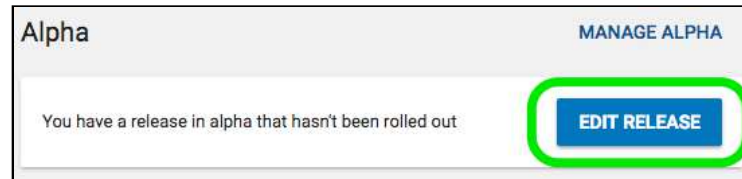
15. Tap the **REVIEW** button at the bottom of the page. This displays a summary of the release and waits for you to confirm the rollout.
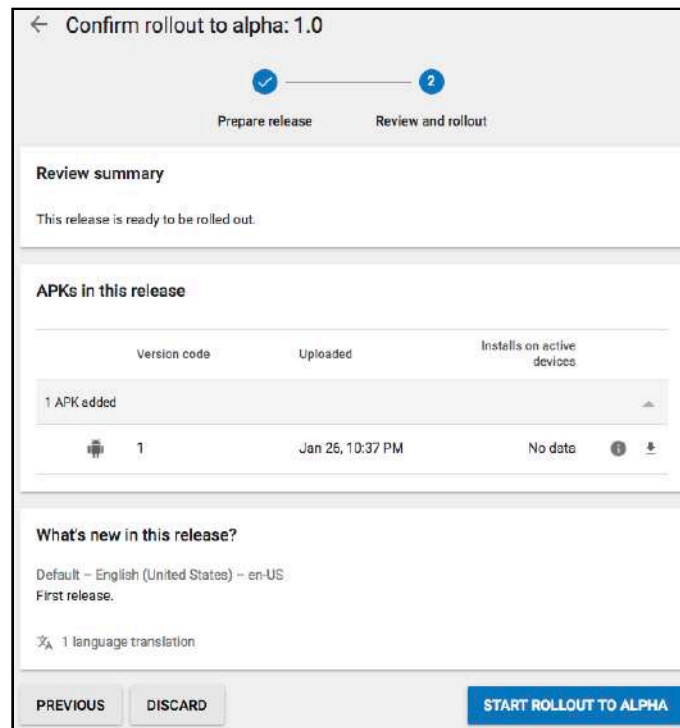
16. Tap the **START ROLLOUT TO ALPHA** button at the bottom of the page if everything looks good.

17. Tap **CONFIRM** in the popup dialog.
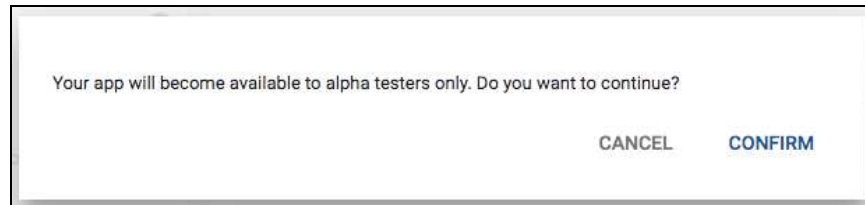
And that's it. Congratulations on publishing your first app to the Play store!

You may notice that the top of the screen shows a status of **Pending publication**.
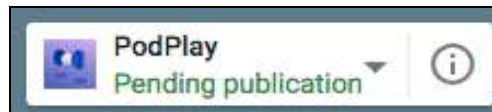
This is a temporary state while Google does all of the processing required to generate the Play store listing. Take this opportunity to reward yourself with a break, and check back in on the progress in about 30 minutes.

Even before the app is fully published, you can start exploring the new options that are now available for the application.

You have access to a Dashboard with a variety of device install information, a detailed statistics page, the Android vitals page with access to crash reports, User acquisition, User feedback sections and more.

Take a few minutes to dive into the different sections of the console and see what information is provided.

There's also a Pre-launch report that can provide valuable information about how your app is performing on real devices. On this page, you can see screenshots of your app running on a large variety of devices, security scan results and performance and crash reports.

The Pre-launch section can provide valuable insights into how your app runs on actual hardware devices even before testers have downloaded it.

*Pre-launch Performance Tab*

Pre-launch Screenshots Tab

Pre-launch Security Tab

Once the app status changes to **Published**, which you'll see at the top of the page, head back to the **App releases** page and click **MANAGE ALPHA**.

You'll notice a few new options on the page now. There's a **Rollout** history that shows how long ago the app was rolled out. There are also buttons that let you quickly release this version to beta and production.
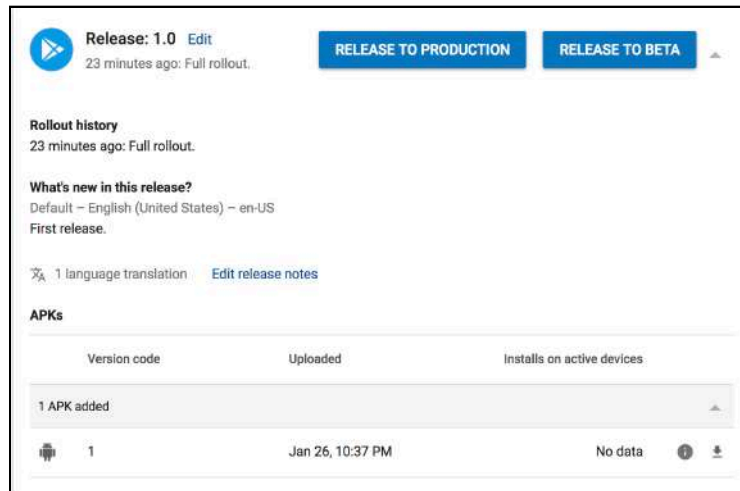

Expand the **Manage testers** section to view the new **Opt-in URL**. You can send this URL to your testers. Google won't send emails to your testers for you; you must notify them when the release is ready and include the Opt-in URL.
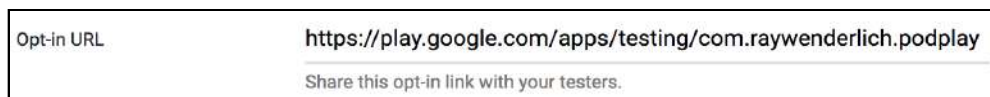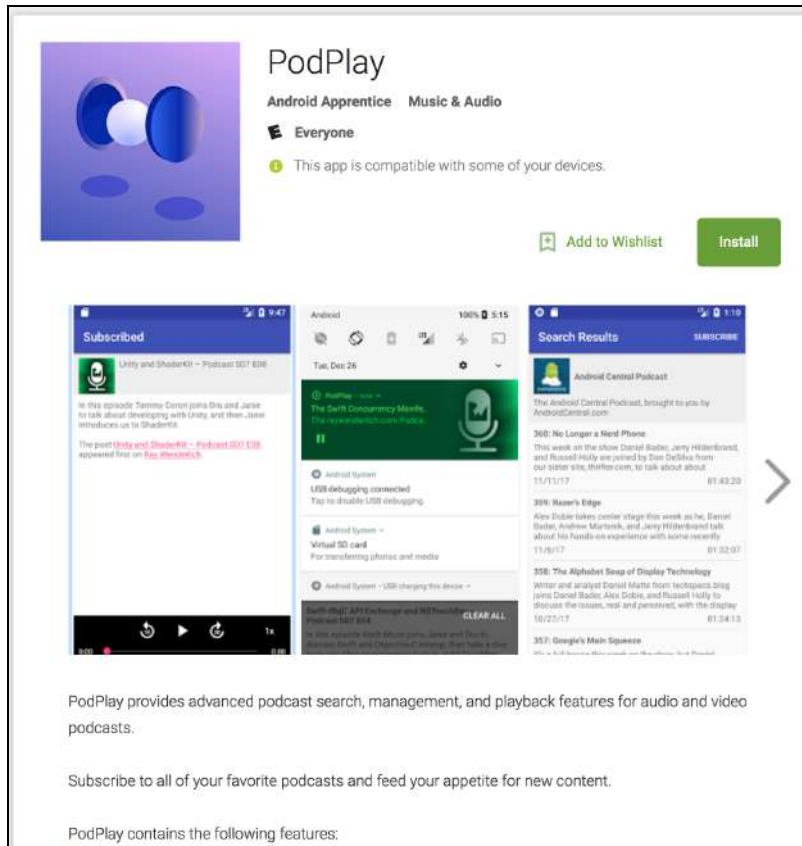

When a tester brings up the Opt-in URL, they'll see a message like the following:
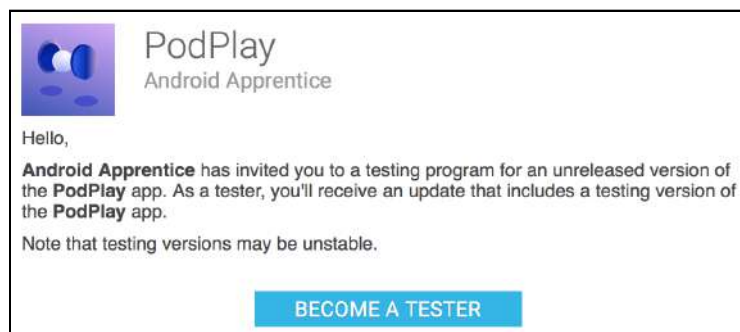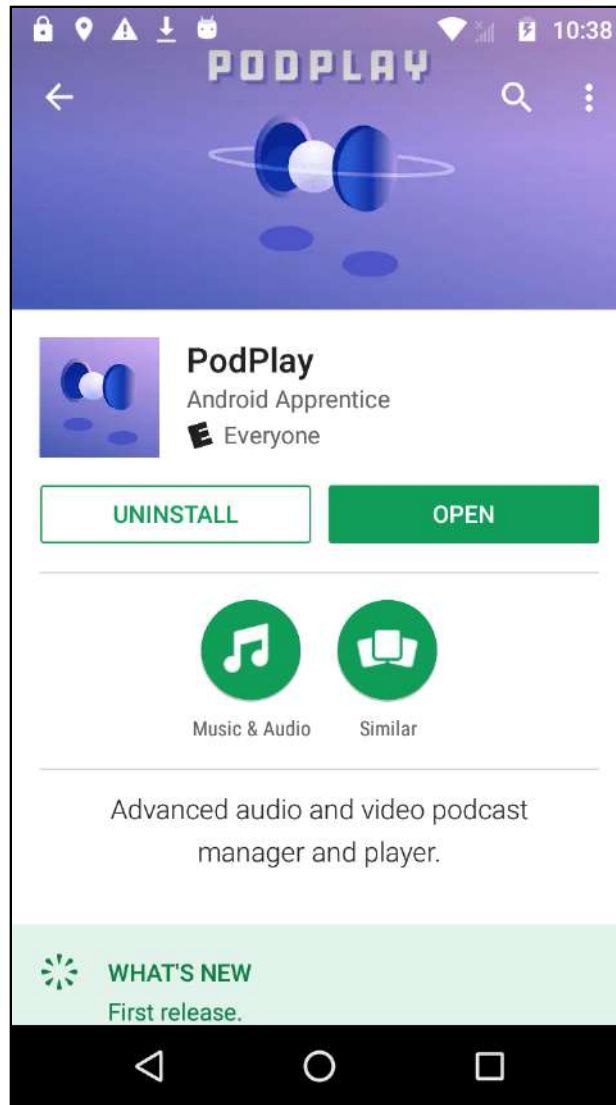

Once the user clicks **BECOME A TESTER**, they'll get a confirmation screen with a link to the Google Play store listing.
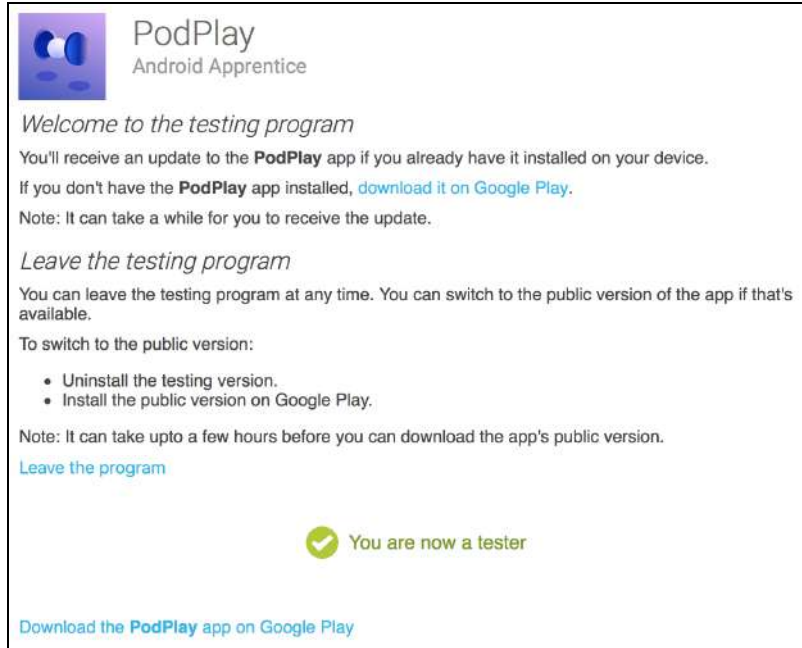
When they visit the store link from a desktop computer, the final listing will look like this:

Here's the final listing at the Play store on a device:

This looks like any other Play store listing; the only difference is that you have control over who can install the app.

Version codes

Before moving on to Beta testing, let's take a quick look at how to use Version codes through the different release phases. Typically, you want your Alpha release to have the highest version code since it should be testing the most recent changes. Your Beta release will have the next highest version code, and Production will have the lowest version code.

If an Alpha tester is a member of the Alpha and Beta test groups, and you upload a Beta with a higher version code than the Alpha, the tester gets updated to the Beta version.

Note that some users may be in the Alpha group only, Beta group only or in both groups of testers.

Here's a typical lifecycle an app might follow through the first few releases.

1. Release Version 1 to Alpha.

2. Alpha testers install Version 1 and complete testing.

3. Promote Version 1 to Beta.

4. Beta testers install Version 1 and find some issues.

5. Address issues in Version 2.

6. Release Version 2 to Alpha.

7. Beta only testers continue with Version 1. Alpha testers update to Version 2.

8. Version 2 testing is complete.

9. Promote Version 2 to Beta. Beta testers update to version 2.

10. Release Version 2 to Production, which means users can download it from the Play Store.

11. Release Version 3 to Alpha.

12. Alpha testers update to Version 3. Beta only testers and general public remain on version 2.

Beta release

Use the **Beta** option for publishing to select users that may not be internal to your organization, or run an Open Beta that lets anyone on the Play Store sign up for Beta testing.

Once you're satisfied with Alpha testing, it's a simple step to move to the Beta phase.

> **Note**: You can only have one test phase active at a time for either open or closed test types. For example, you can't have a Closed Alpha and Closed Beta happening at the same time. You can, however, have a Closed Alpha and an Open Beta at the same time.

Go to the **App release ▸ MANAGE ALPHA** section and click **RELEASE TO BETA**.

This creates a release in Beta using the same version you have in Alpha and takes you to the release screen where you can verify the details.

You can update the release notes or upload a new APK at this point. If you're just promoting the Alpha release as-is, then you don't need to make any changes.

Click the **REVIEW** button. You'll get a warning that you don't have any testers set up yet for the Beta test.

Click the back arrow to go back to the **Beta** management page and open the **Manage testers** section. These are the same steps you went through when setting up the **Alpha** testers.

Select a testing method and save your changes. If you choose **Closed** testing, you'll have the option of using the previous list of users from the **Alpha** release, or you can create an additional or separate list of users for the **Beta**.
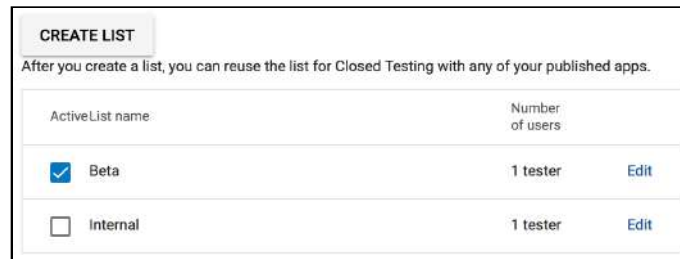
Click **EDIT RELEASE** under the Beta section and then click **REVIEW** again. This time, there's no warning about not having testers.

Click **START ROLLOUT TO BETA**. You may see additional warnings, such as pre-launch warnings about crashes on devices. It's up to you to decide if these warnings are critical, or if you should go ahead with the rollout.

After the rollout to Beta is complete, the **App release** overview page shows that the Alpha version was promoted to Beta.

Production release

Once the Alpha and Beta testing is complete, you're ready for the final release to Production! This is as simple as promoting the final Beta version to Production.

Go to the **App release ▸ MANAGE BETA** section and click **RELEASE TO PRODUCTION**.

This creates a release in Production and displays the release screen to verify the details. Just like the Alpha and Beta release, you need to review the release information before it's published to the Play store.

When your ready, click **REVIEW**. If everything checks out, the **START ROLLOUT TO PRODUCTION** button is displayed.

For first-time app publishers, this can be both an exciting and stress-inducing moment. The app you've worked so hard on will finally be available for the public to enjoy!

Don't be nervous, go ahead and click **START ROLLOUT TO PRODUCTION**.

Pay attention to any warnings that may crop up. If everything looks good, click **CONFIRM**.

Within a short amount of time, your app will be live in the Play store. Go ahead and celebrate. Throw a launch party and spread the news about your first published app!

But don't party too long, because you're not done yet. Just like a newborn child, your production app can't be left on its own. It needs some loving care and attention to thrive!

Post-production

Here are some final tips to help keep your app in top shape.

- **Review your app stats on a regular basis**. The Play Console provides a wealth of information about the number and types of installs, number and frequency of crashes and overall ratings. Look for spikes or drops in any these categories to stay on top of changes. Don't assume items will fix themselves; be proactive and address issues as soon as they appear.

- **Check reviews and look for problem trends**. It's inevitable that even the best-made apps will get some negative reviews. Look for common threads in the reviews. If a large number of users are all complaining about the same thing, that's an excellent hint to focus on that issue. Positive reviews can also provide valuable feedback about what you're doing right and help drive future product development.

- **Be aware of new Android releases**. In some cases, a new Android OS release can impact how your existing app performs. Make sure to keep up with beta releases of the Android OS, and make sure your app performs as expected before the OS is released to the public.

- **List well-known issues**. If you have issues that are known and can't be fixed quickly, consider mentioning them in the Play store description along with a workaround if possible. It's better if users know about these before being surprised after they download the app.

- **Consider using staged rollouts**. Google has built-in support for staged rollouts only for app updates, not on the initial release. When using staged rollouts, you specify the percentage of users that will be updated to the new release. You can also limit the update to specific countries. For the first release, you might consider rolling out to a single, smaller country, before targeting your primary countries.

Other publishing methods

In some cases, you may need to distribute an app without going through the Play store. It might be an enterprise app that will never go public, or it might be a side project that you're distributing to friends and family.

There are a few ways to distribute an app directly.

Email distribution

Email requires the least amount of work on your part. All you do is attach the APK file to an email, and have your users open the email on a compatible Android device.

Users will need to configure their device to allow "unknown sources" before the APK can be installed. It's a good idea to include instructions in the email when sending out the APK.

If a user is running Android 8.0 or newer, they should look for the **Install unknown apps** section in the device settings.

If a user is running a version before Android 8.0, they should enable **Unknown sources** in the **Security** section of the device settings.

When the user opens an email with an APK attached, they can download the APK.
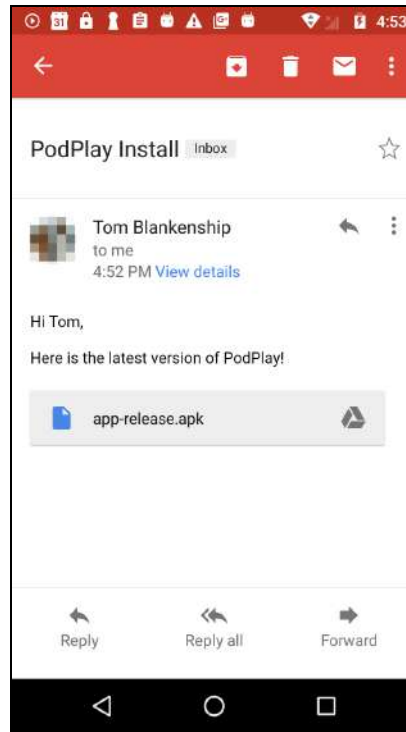
The user will then find the APK in their downloads app or by pulling down the notifications view. When they tap on the APK file, they'll be prompted to install the app.
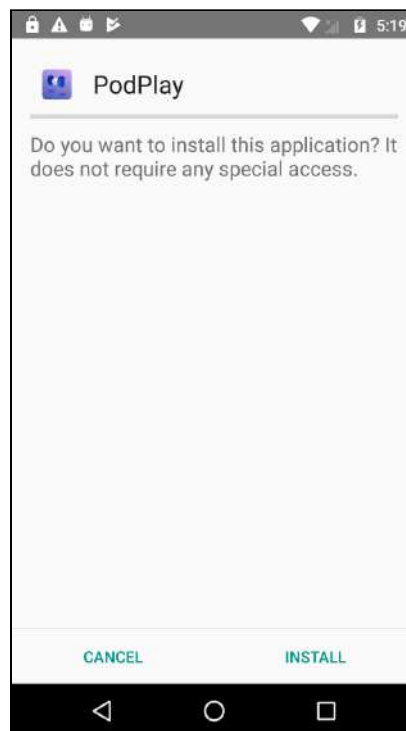
Website distribution

Another option is to host the APK file on your website. You can either send a link to the download location or point the users to the download page on your site. Whether the user taps on the link from an email or the browser on the device, they'll be prompted to install the APK.

As with email distribution, the device must be configured to allow unknown sources.

Other app stores

There are some other app stores available for publishing your app; you should take time to explore which options are available. One of the most well-known stores is the **Amazon Appstore**. Amazon's Appstore is installed by default on Amazon devices such as the **Fire TV** and **Fire Tablet**. It contains apps made especially for the Amazon products as well as many apps that can also be found in the Google Play store.

There are some fundamental differences between Google and Amazon in the way apps are purchased and how in-app billing is handled. However, in both cases, you'll get 70% of the app earnings.

One big difference is that you can't switch a free app on Google Play to a paid one. That decision must be made during the initial rollout. Amazon lets you start your app as free and change to paid at any time.

You can always start by releasing to the Google Play store and then decide later if you also want to distribute the app to other app stores.

Conclusion

Congratulations! You've completed the first steps of your journey as an Android developer. When a toddler learns to walk, it can be those first few steps that seem the most gratifying. But we all know that those initial, wobbly steps are only a starting point. Likewise, the skills and knowledge that you have gained throughout these chapters will act as your foundation for many future projects and creative endeavors.

Take time to enjoy the success that you've found in completing the material provided in this book, and then look forward. A developer's world is a blank canvas, just waiting for the stroke of the creator. Endless possibilities await; projects that need the special touch of your talent, creativity, and unique ideas. Take your next step into Android development, and press onward with confidence.

If you have any questions or comments as you work through this book, please stop by our forums at http://forums.raywenderlich.com and look for the particular forum category for this book.

Thank you again for purchasing this book. Your continued support is what makes the tutorials, books, videos, conferences and other things we do at raywenderlich.com possible, and we truly appreciate it!

Wishing you all the best in your continued Android adventures,

– Darryl, Tom, Vijay, Namrata, Ellen, Tammy, Eric and Chris

The *Android Apprentice* team

More Books You Might Enjoy

We hope you enjoyed this book! If you're looking for more, we have a whole library of books waiting for you at https://store.raywenderlich.com.

New to iOS or Swift?

Learn how to develop iOS apps in Swift with our classic, beginner editions.

iOS Apprentice

https://store.raywenderlich.com/products/ios-apprentice

The iOS Apprentice is a series of epic-length tutorials for beginners where you'll learn how to build 4 complete apps from scratch.

Each new app will be a little more advanced than the one before, and together they cover everything you need to know to make your own apps. By the end of the series you'll be experienced enough to turn your ideas into real apps that you can sell on the App Store.

These tutorials have easy to follow step-by-step instructions, and consist of more than 900 pages and 500 illustrations! You also get full source code, image files, and other resources you can re-use for your own projects.

Swift Apprentice

https://store.raywenderlich.com/products/swift-apprentice

This is a book for complete beginners to Apple's brand new programming language — Swift 4.

Everything can be done in a playground, so you can stay focused on the core Swift 4 language concepts like classes, protocols, and generics.

This is a sister book to the iOS Apprentice; the iOS Apprentice focuses on making apps, while Swift Apprentice focuses on the Swift 4 language itself.

Experienced iOS developer?

Level up your development skills with a deep dive into our many intermediate to advanced editions.

Data Structures and Algorithms in Swift

https://store.raywenderlich.com/products/data-structures-and-algorithms-in-swift
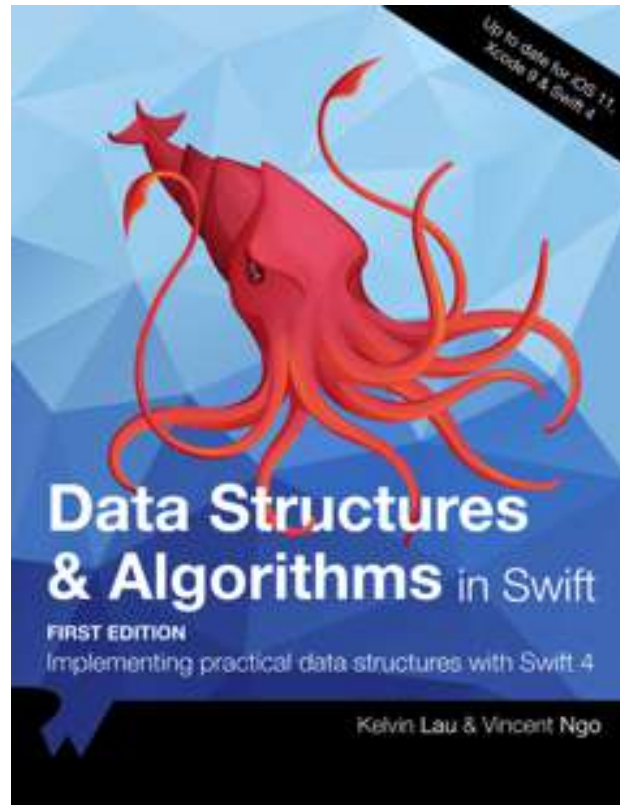
Understanding how data structures and algorithms work in code is crucial for creating efficient and scalable apps. Swift's Standard Library has a small set of general purpose collection types, yet they definitely don't cover every case!

In Data Structures and Algorithms in Swift, you'll learn how to implement the most popular and useful data structures, and when and why you should use one particular datastructure or algorithm over another. This set of basic data structures and algorithms will serve as an excellent foundation for building more complex and special-purpose constructs. As well, the high-level expressiveness of Swift makes it an ideal choice for learning these core concepts without sacrificing performance.

Realm: Building Modern Swift Apps with Realm Database

https://store.raywenderlich.com/products/realm-building-modern-swift-apps-with-realm-database
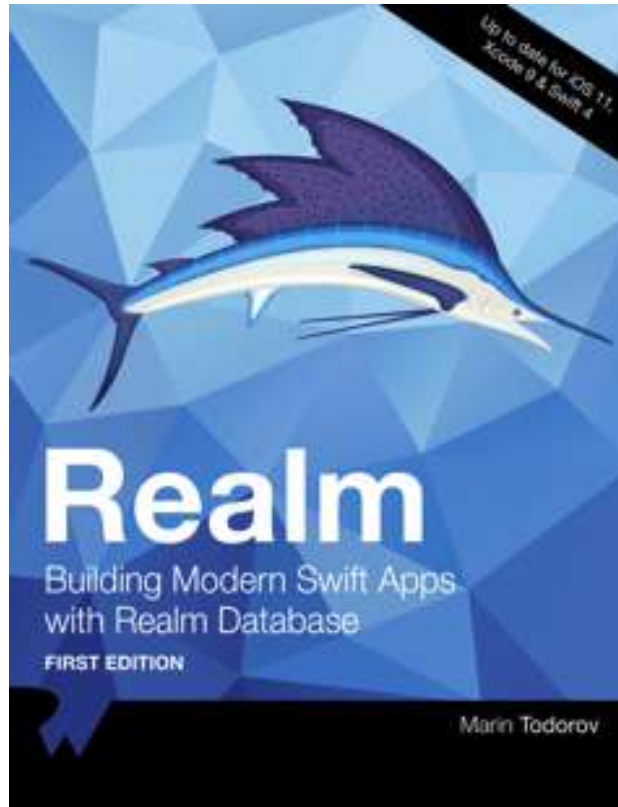
Realm Platform is a relatively new commercial product which allows developers to automatically synchronize data not only across Apple devices but also between any combination of Android, iPhone, Windows, or macOS apps. Realm Platform allows you to run the server software on your own infrastructure and keep your data in-house which more often suits large enterprises. Alternatively you can use Realm Cloud which runs a Platform for you and you start syncing data very quickly and only pay for what you use.

In this book, you'll take a deep dive into the Realm Database, learn how to set up your first Realm database, see how to persist and read data, find out how to perform migrations and more. In the last chapter of this book, you'll take a look at the synchronization features of Realm Cloud to perform real-time sync of your data across all devices.

Design Patterns by Tutorials

https://store.raywenderlich.com/products/design-patterns-by-tutorials

Design patterns are incredibly useful, no matter what language or platform you develop for. Using the right pattern for the right job can save you time, create less maintenance work for your team and ultimately let you create more great things with less effort. Every developer should absolutely know about design patterns, and how and when to apply them. That's what you're going to learn in this book!

Move from the basic building blocks of patterns such as MVC, Delegate and Strategy, into more advanced patterns such as the Factory, Prototype and Multicast Delegate pattern, and finish off with some less-common but still incredibly useful patterns including Flyweight, Command and Chain of Responsibility.

Server Side Swift with Vapor

https://store.raywenderlich.com/products/server-side-swift-with-vapor

If you're a beginner to web development, but have worked with Swift for some time, you'll find it's easy to create robust, fully-featured web apps and web APIs with Vapor 3.

Whether you're looking to create a backend for your iOS app, or want to create fully-featured web apps, Vapor is the perfect platform for you.

This book starts with the basics of web development and introduces the basics of Vapor; it then walks you through creating APIs and web backends; creating and configuring databases; deploying to Heroku, AWS, or Docker; testing your creations and more!

iOS 11 by Tutorials

https://store.raywenderlich.com/products/ios-11-by-tutorials

This book is for intermediate iOS developers who already know the basics of iOS and Swift development but want to learn the new APIs introduced in iOS 11.

Discover the new features for developers in iOS 11, such as ARKit, Core ML, Vision, drag & drop, document browsing, the new changes in Xcode 9 and Swift 4 — and much, much more.

Advanced Debugging and Reverse Engineering

https://store.raywenderlich.com/products/advanced-apple-debugging-and-reverse-engineering
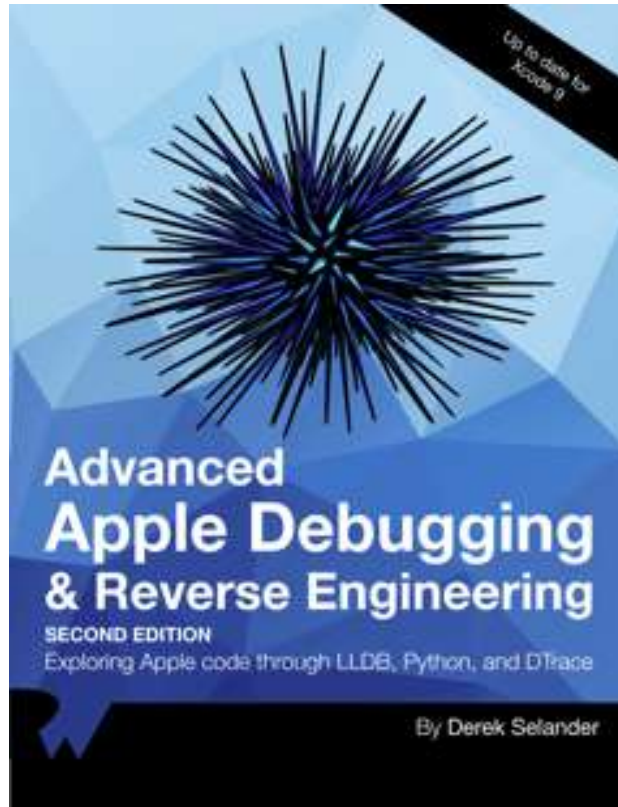
In Advanced Apple Debugging and Reverse Engineering, you'll come to realize debugging is an enjoyable process to help you better understand software. Not only will you learn to find bugs faster, but you'll also learn how other developers have solved problems similar to yours.

You'll also learn how to create custom, powerful debugging scripts that will help you quickly find the secrets behind any bit of code that piques your interest.

After reading this book, you'll have the tools and knowledge to answer even the most obscure question about your code — or someone else's.

RxSwift: Reactive Programming with Swift

https://store.raywenderlich.com/products/rxswift

This book is for iOS developers who already feel comfortable with iOS and Swift, and want to dive deep into development with RxSwift.

Start with an introduction to the reactive programming paradigm; learn about observers and observables, filtering and transforming operators, and how to work with the UI, and finish off by building a fully-featured app in RxSwift.

Core Data by Tutorials

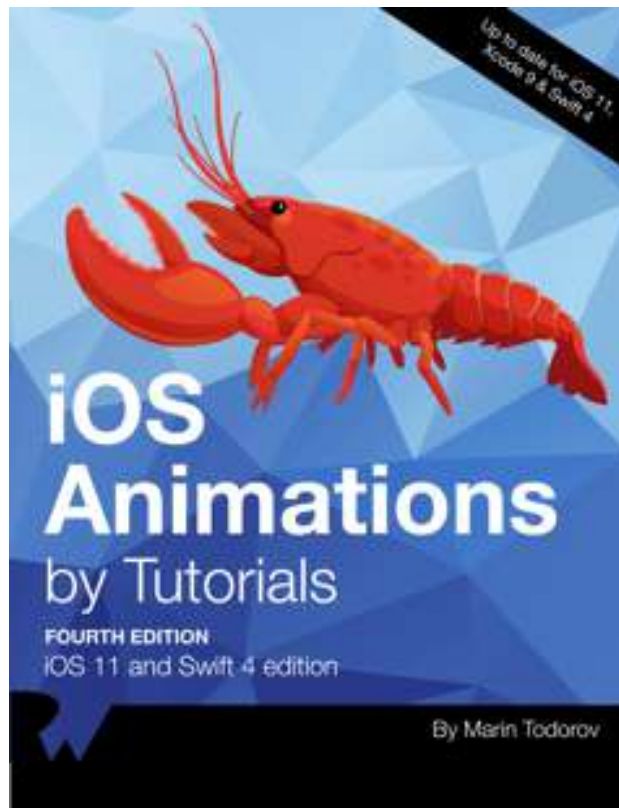https://store.raywenderlich.com/products/core-data-by-tutorials
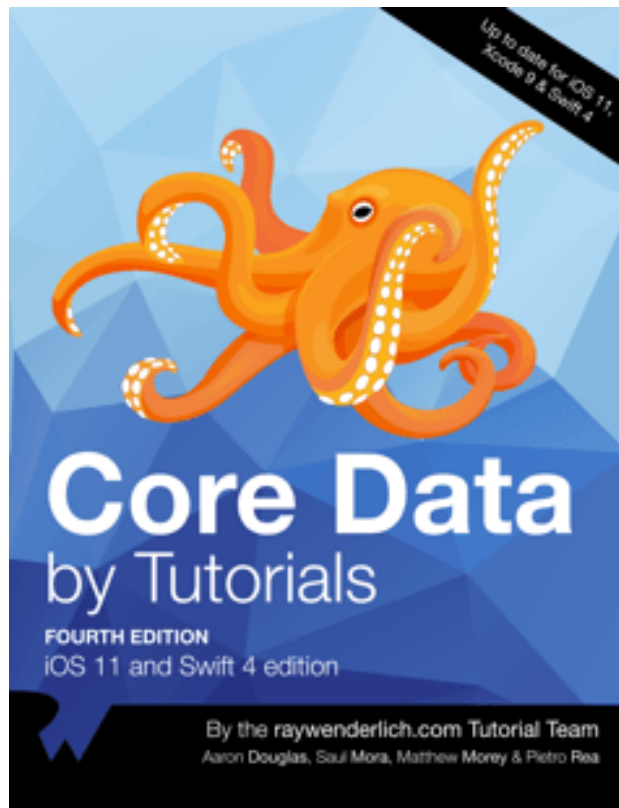
This book is for intermediate iOS developers who already know the basics of iOS and Swift 4 development but want to learn how to use Core Data to save data in their apps.

Start with with the basics like setting up your own Core Data Stack all the way to advanced topics like migration, performance, multithreading, and more!

iOS Animations by Tutorials

https://store.raywenderlich.com/products/ios-animations-by-tutorials

This book is for iOS developers who already know the basics of iOS and Swift 4, and want to dive deep into animations.

Start with basic view animations and move all the way to layer animations, animating constraints, view controller transitions, and more!

watchOS by Tutorials

https://store.raywenderlich.com/products/watchos-by-tutorials

This book is for intermediate iOS developers who already know the basics of iOS and Swift development but want to learn how to make Apple Watch apps for watchOS 4.

tvOS Apprentice

https://store.raywenderlich.com/products/tvos-apprentice

This book is for complete beginners to tvOS development. No prior iOS or web development knowledge is necessary, however the book does assume at least a rudimentary knowledge of Swift.

This book teaches you how to make tvOS apps in two different ways: via the traditional method using UIKit, and via the new Client-Server method using TVML.

Want to make games?

Learn how to make great-looking games that are deeply engaging and fun to play!

2D Apple Games by Tutorials

https://store.raywenderlich.com/products/2d-apple-games-by-tutorials

In this book, you will make 6 complete and polished mini-games, from an action game to a puzzle game to a classic platformer!

This book is for beginner to advanced iOS developers. Whether you are a complete beginner to making iOS games, or an advanced iOS developer looking to learn about SpriteKit, you will learn a lot from this book!

3D Apple Games by Tutorials

https://store.raywenderlich.com/products/3d-apple-games-by-tutorials

Through a series of mini-games and challenges, you will go from beginner to advanced and learn everything you need to make your own 3D game!

This book is for beginner to advanced iOS developers. Whether you are a complete beginner to making iOS games, or an advanced iOS developer looking to learn about SceneKit, you will learn a lot from this book!

Unity Games by Tutorials

https://store.raywenderlich.com/products/unity-games-by-tutorials

Through a series of mini-games and challenges, you will go from beginner to advanced and learn everything you need to make your own 3D game!

This book is for beginner to advanced iOS developers. Whether you are a complete beginner to making iOS games, or an advanced iOS developer looking to learn about SceneKit, you will learn a lot from this book!

Want to learn Android or Kotlin?

Get a head start on learning to develop great Android apps in Kotlin, the newest first-class language for building Android apps.

Android Apprentice

https://store.raywenderlich.com/products/android-apprentice
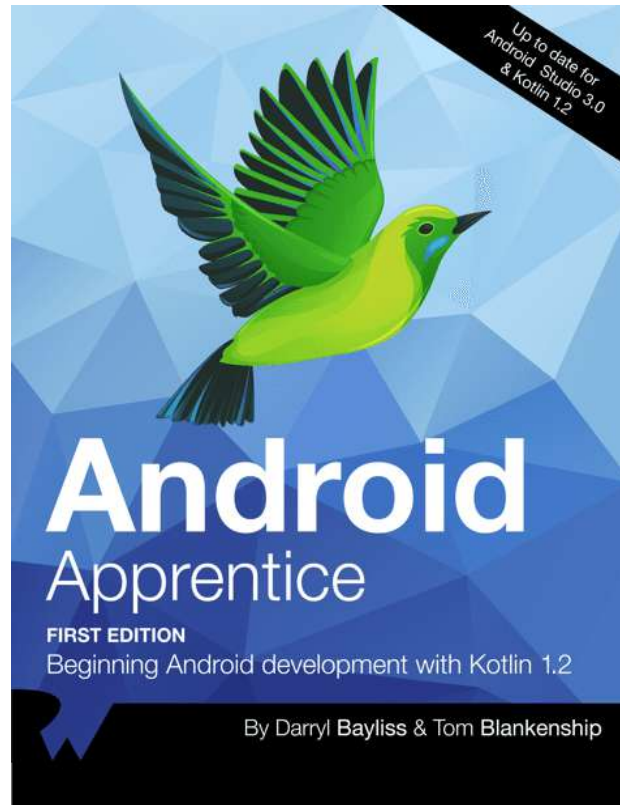
If you're completely new to Android or developing in Kotlin, this is the book for you!

The Android Apprentice takes you all the way from building your first app, to submitting your app for sale. By the end of this book, you'll be experienced enough to turn your vague ideas into real apps that you can release on the Google Play Store.

You'll build 4 complete apps from scratch — each app is a little more complicated than the previous one. Together, these apps will teach you how to work with the most common controls and APIs used by Android developers around the world.

Kotlin Apprentice

https://store.raywenderlich.com/products/kotlin-apprentice

This is a book for complete beginners to the new, modern Kotlin language.

Everything in the book takes place in a clean, modern development environment, which means you can focus on the core features of programming in the Kotlin language, without getting bogged down in the many details of building apps.

This is a sister book to the Android Apprentice the Android Apprentice focuses on making apps for Android, while the Kotlin Apprentice focuses on the Kotlin language fundamentals.