

Up to date for
Android Studio 3.3
& Kotlin 1.3



Android Apprentice

SECOND EDITION

Beginning Android Development with Kotlin

By the raywenderlich.com Tutorial Team

Namrata Bandekar, Darryl Bayliss, Tom Blankenship & Fuad Kamal

Android Apprentice, Second Edition

Darryl Bayliss, Tom Blankenship, Fuad Kamal & Namrata Bandekar

Copyright ©2019 Razeware LLC.

Notice of Rights

All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written permission of the copyright owner.

Notice of Liability

This book and all corresponding materials (such as source code) 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 action of contract, tort or otherwise, arising from, out of or in connection with the software or the use of other dealing in the software.

Trademarks

All trademarks and registered trademarks appearing in this book are the property of their own respective owners.

Dedications

"To Dhaval, for always supporting me. To Ginger, for all the laughs and hugs!"

— *Namrata Bandekar*

"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*

"To my parents, who took care of me when I was small. To my children, the apples of my eyes, who provided the light when it was dark and the inspiration to keep going. To my wife, for putting up with me. To Ray Wenderlich, and the entire Ray Wenderlich team, for establishing the gold standard for quality content and writing and showing me the way it's done."

— *Fuad Kamal*

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 or playing some fantastical video game involving magic and dragons. You can 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.



Fuad Kamal is an author on this book. He provides mobile strategy, architecture and development for the Health & Fitness markets. If you've ever been to an airport, you've likely seen his work — the flight arrival and departure screens are a Flash 7 interface he wrote towards the beginning of the millennium. He can be contacted through [anaara.com](#).



Namrata Bandekar is one of the authors and a tech editor of this book. She is a Software Engineer focusing on native Android and iOS development. When she's not developing apps, she enjoys spending her time travelling the world with her husband, SCUBA diving and hiking with her dog. She has also spoken at a number of international conferences on mobile development. Say hi to Namrata on Twitter: [@NamrataCodes](#).

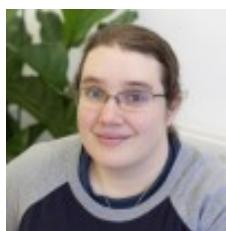
About the Editors



Kevin Moore is a technical editor for this book. He has been developing Android apps for over 8 years and at many companies. He's written several articles at www.raywenderlich.com and created the "Programming in Kotlin" video series. He enjoys creating apps for fun and teaching others how to write Android apps. In addition to programming, he loves playing volleyball and running the sound system at church.



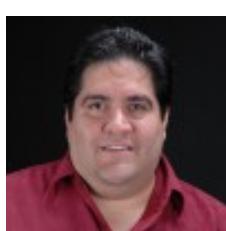
Vijay Sharma is a tech editor of this book. Vijay is a husband, a father and a senior mobile engineer. Based out of Canada's capital, Vijay has worked on dozens of apps for both Android and iOS. When not in front of his laptop, you can find him in front of a TV, behind a book, or chasing after his kids.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



Ellen Shapiro is a tech editor on this book. Ellen is an iOS developer for Bakken & Bæk'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](https://twitter.com/@designatednerd)



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, certified Agile-Scrum Master and Mac fanatic. Focusing on Apple iOS, Android Apps, NodeJS and APIs, Eric works with clients all across the US including many big-name brands. During his 30+ year career, Eric has also worked with web applications, server back-end systems, automated infrastructure deployments and more. Follow Eric on Twitter [@ericwastaken](https://twitter.com/@ericwastaken) or on his website ericsoto.net.

Table of Contents: Overview

Introduction.....	17
Book License.....	18
Book Source Code & Forums	19
About the Cover.....	20
<u>Section I: Your First Android App.....</u>	<u>21</u>
Chapter 1: Setting Up Android Studio	22
Chapter 2: Layouts	48
Chapter 3: Activities	63
Chapter 4: Debugging	80
Chapter 5: Prettifying the App	90
<u>Section II: Building a List App.....</u>	<u>107</u>
Chapter 6: Creating a New Project.....	108
Chapter 7: RecyclerViews.....	117
Chapter 8: SharedPreferences	137
Chapter 9: Communicating Between Activities	151
Chapter 10: Completing the Detail View	167
Chapter 11: Using Fragments.....	188
Chapter 12: Material Design	217
<u>Section III: Creating Map-Based Apps</u>	<u>233</u>

Chapter 13: Creating a Map-Based App.....	234
Chapter 14: User Location & Permissions.....	257
Chapter 15: Google Places	276
Chapter 16: Saving Bookmarks with Room.....	297
Chapter 17: Detail Activity	325
Chapter 18: Navigation & Photos	359
Chapter 19: Finishing Touches	389
Section IV: Building a Podcast Manager & Player ..	419
Chapter 20: Networking	420
Chapter 21: Finding Podcasts.....	439
Chapter 22: Podcast Details	462
Chapter 23: Podcast Episodes	482
Chapter 24: Podcast Subscriptions, Part One.	502
Chapter 25: Podcast Subscriptions, Part Two.	525
Chapter 26: Podcast Playback	543
Chapter 27: Episode Player	578

<u>Section V: Android Compatibility</u>	<u>617</u>
Chapter 28: Android Fragmentation & Support Libraries.....	618
Chapter 29: Keeping Your App Up to Date	626
<u>Section VI: Publishing Your App.....</u>	<u>634</u>
Chapter 30: Preparing for Release	635
Chapter 31: Testing & Publishing	651
Conclusion	675

Table of Contents: Extended

Introduction	17
Book License.....	18
Book Source Code & Forums.....	19
About the Cover	20
<u>Section I: Your First Android App</u>	<u>21</u>
Chapter 1: Setting Up Android Studio.....	22
Getting started	22
Your first Android project	25
Android Studio.....	28
Creating an Android virtual device.....	30
Setting up an Android device	36
Running the app	40
Installing new versions of Android Studio	44
Where to go from here?	47
Chapter 2: Layouts.....	48
Getting started	49
These are not the SDKs you're looking for.....	49
The Visual editor	50
Component tree view	52
Positioning your views.....	54
Adding rules to your position	55
Finishing the screen	58
Where to go from here?.....	61
Chapter 3: Activities	63
Getting started	63
Exploring Activities	68
Hooking up Views	70

Managing strings in your app	73
Progressing the game	74
Starting the game	76
Ending the game	78
Where to go from here?	79
Chapter 4: Debugging.....	80
Getting started	80
Add some logging	81
Orientation changes	83
Breakpoints	85
Restarting the game	87
Where to go from here?	89
Chapter 5: Prettifying the App	90
Getting started.....	91
Changing the app bar color	92
Animations.....	94
Adding a Dialog	98
Where to go from here?.....	106
Section II: Building a List App	107
Chapter 6: Creating a New Project	108
Getting started.....	109
Creating a new Android project	112
Targeting Android devices	114
Where to go from here?.....	116
Chapter 7: RecyclerViews.....	117
Getting started.....	118
Adding a RecyclerView	120
The components of a RecyclerView	123
Hooking up a RecyclerView.....	125
Setting up a RecyclerView Adapter.....	126
Filling in the blanks	129

Creating the ViewHolder	129
Binding data to your ViewHolder	135
The moment of truth	135
Where to go from here?	136
Chapter 8: SharedPreferences	137
Getting started	138
Adding a Dialog	139
Creating a list	142
Hooking up the Activity	145
Where to go from here?	150
Chapter 9: Communicating Between Activities	151
Getting started	151
Creating a new Activity	155
The app manifest	157
Intents	159
Intents and Parcels	160
Bringing everything together	162
Where to go from here?	165
Chapter 10: Completing the Detail View	167
Getting started	167
Coding the RecyclerView	170
Adapting the Adapter	173
Visualizing the ViewHolder	178
Getting the list back	184
Where to go from here?	187
Chapter 11: Using Fragments	188
Getting started	189
Creating a Fragment	193
What is a Fragment?	196
From Activity to Fragments	198
Showing the Fragment	203

Creating your next Fragment	206
Bringing the Activity into action	209
Where to go from here?.....	216
Chapter 12: Material Design	217
What is Material Design?.....	218
Primary and secondary colors	220
Card views.....	228
Where to go from here?	232
Section III: Creating Map-Based Apps.....	233
Chapter 13: Creating a Map-Based App.....	234
Getting started	234
About PlaceBook.....	235
Making a plan	235
Location service components.....	236
Map wizard walk-through	237
Google Maps API key	239
Maps and the emulator.....	244
Running the app	247
The difficulty of determining locations	254
Where to go from here?	256
Chapter 14: User Location & Permissions	257
Getting started	257
Adding location services.....	258
Creating the location services client	260
Querying current location.....	260
Faking locations in the emulator.....	268
Tracking the user's location	270
My location	272
Where to go from here?	275
Chapter 15: Google Places	276
Getting started	276

Places API overview	278
Selecting points of interest.....	279
Load place details	281
Custom info window	290
Where to go from here?	296
Chapter 16: Saving Bookmarks with Room	297
Getting started	297
Room overview	298
Room and Android Architecture Components.....	299
PlaceBook architecture.....	300
Development approach	301
Adding the architecture components.....	303
Room classes	304
Creating the Repository.....	310
The ViewModel	312
Adding bookmarks.....	313
Observing database changes	318
Where to go from here?	324
Chapter 17: Detail Activity	325
Getting started	325
Fixing the info window	326
Bookmark detail activity	335
Where to go from here?	358
Chapter 18: Navigation & Photos	359
Getting started	359
Bookmark navigation.....	359
Custom photos	374
Where to go from here?	388
Chapter 19: Finishing Touches.....	389
Getting started	390
Bookmark categories	390

Searching for places	401
Create ad-hoc bookmarks	405
Deleting bookmarks	407
Sharing bookmarks	410
Color scheme.....	414
Progress indicator	415
Where to go from here?.....	418
Section IV: Building a Podcast Manager & Player..	419
Chapter 20: Networking	420
Getting started.....	421
Where are the podcasts?	425
Android networking.....	425
PodPlay architecture.....	426
iTunes search service	427
Retrofit.....	428
Where to go from here?	438
Chapter 21: Finding Podcasts	439
Android search	439
Where to go from here?	461
Chapter 22: Podcast Details.....	462
Getting started	462
Defining the Layouts.....	463
Basic architecture.....	466
Details Fragment	471
Where to go from here?	481
Chapter 23: Podcast Episodes	482
Getting started	482
Updating the podcast repo.....	494
Episode list adapter	496
Where to go from here?.....	501

Chapter 24: Podcast Subscriptions, Part One.....	502
Getting started	502
Saving podcasts	503
Where to go from here?	524
Chapter 25: Podcast Subscriptions, Part Two.....	525
Getting started	525
Background methods	526
Episode update logic.....	528
Firebase JobDispatcher.....	531
Where to go from here?	542
Chapter 26: Podcast Playback	543
Getting started	543
Media player basics	544
Building the MediaBrowserService.....	546
Connecting the MediaBrowser	549
Foreground service	566
Final pieces	574
Where to go from here?	577
Chapter 27: Episode Player.....	578
Getting started	579
Video playback	606
Where to go from here?.....	616
Section V: Android Compatibility.....	617
Chapter 28: Android Fragmentation & Support Libraries.....	618
Android: An open operating system	619
How fragmenting occurs.....	619
The Android Support Libraries	621
Where to go from here?	624
Chapter 29: Keeping Your App Up to Date	626

Following Android trends	627
Android versions	628
Screen size and density	629
OpenGL versions.	630
Managing Android updates	631
Working with older versions of Android	632
Where to go from here?	633
Section VI: Publishing Your App	634
Chapter 30: Preparing for Release	635
Code cleanup	636
Versioning information	637
Build release version.	638
Check file size	643
Release testing	644
Google Play Store	644
Where to go from here?	650
Chapter 31: Testing & Publishing	651
Release types.	651
Determining the content rating	658
Pricing and distribution.	660
Version codes	666
Beta release.	667
Production release.	671
Post-production.	671
Other publishing methods	672
Conclusion	675

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 & 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 Android apps, you first need to install **Android Studio**. Android Studio is a customized IDE based on IntelliJ, an IDE by JetBrains, that provides you a powerful set of tools.

In this chapter, you'll learn how to:

- Set up Android Studio.
- Set up a physical and emulated device for development.
- Build, install and run an app on your device.

Getting started

Open a 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	3a61a587c90e358ab15d076d0306550564ad4cc5a8aa1dd0c22c6af092e976a
Mac	android-studio-ide-181.5056338-mac.dmg	989 MB	b8d2b7add6a7c776d16a8e48bd35c3e2bba18b4717131d7b9a00fa416ebe4480
Linux	android-studio-ide-181.5056338-linux.zip	1007 MB	b9ec0d44f2feaafe1e3fb1ed696bf325f9e05cfb6c1ace84dbf87ae249efa84

Android Studio can run on a variety of operating systems. Click the package for your operating system to proceed.

Note: This chapter assumes your computer is running macOS; however, because Android Studio supports Windows and Linux, this book will provide instructions where possible for those operating systems as well.

After making your selection, the Terms and Conditions screen appears.

Download Android Studio

Before downloading, you must agree to the following terms and conditions.

Terms and Conditions

This is the Android Software Development Kit License Agreement

1. Introduction

1.1 The Android Software Development Kit (referred to in the License Agreement as the "SDK" and specifically including the Android system files, packaged APIs, and Google APIs add-ons) is licensed to you subject to the terms of the License Agreement. The License Agreement forms a legally binding contract between you and Google in relation to your use of the SDK.

1.2 "Android" means the Android software stack for devices, as made available under the Android Open Source Project, which is located at the following URL: <http://source.android.com/>, as updated from time to time.

1.3 A "compatible implementation" means any Android device that (i) complies with the Android Compatibility Definition document, which can be found at the Android compatibility website (<http://source.android.com/compatibility>) and which may be updated from time to time; and (ii) successfully passes the Android Compatibility Test Suite (CTS).

1.4 "Google" means Google LLC, a Delaware corporation with principal place of business at 1600 Amphitheatre Parkway, Mountain View, CA 94043, United States.

2. Accepting this License Agreement

2.1 In order to use the SDK, you must first agree to the License Agreement. You may not use the SDK if you do not accept the License Agreement.

2.2 By clicking to accept, you hereby agree to the terms of the License Agreement.

2.3 You may not use the SDK and may not accept the License Agreement if you are a person barred from receiving the SDK under the laws of the United States or other

I have read and agree with the above terms and conditions

DOWNLOAD ANDROID STUDIO FOR MAC

android-studio-ide-181.5056338-mac.dmg

To agree to the terms and conditions, check the checkbox at the bottom of the screen, and then click **Download Android Studio**.

As your computer begins to download Android Studio, the browser will display a window with some suggestions on what to do next.

The screenshot shows a web page with a light gray background. At the top left, it says "Thank you for downloading Android Studio!". Below that, a message encourages new users to check out resources like "Build your first app", "Learn with Udacity", and "Explore Android Studio". Each suggestion has a small icon: a phone for the first app, a graduation cap for learning, and a compass for exploring. At the bottom left, there's a link to the "Install guide". On the right side, there's a section about the Android Dev Summit and links to Contents, Windows, Mac, and Linux versions.

Thank you for downloading Android Studio!

If you're new to Android development, check out the following resources to get started.

Build your first app

Start writing code in Android Studio by following the tutorial to [Build your first app](#).

Learn with Udacity

Learn Android with interactive video training in the [Android Fundamentals Udacity course](#).

Explore Android Studio

Introduce yourself to the Android Studio UI and its various tools in [Meet Android Studio](#).

For help installing Android Studio, see the [Install guide](#).

Click the **blue install guide link** at the bottom of the window, which opens a new page containing set up videos for Android Studio for each operating system.

The screenshot shows a page titled "Install Android Studio". It features a sidebar with navigation links for "Meet Android Studio" and other topics. The main content area has a heading "Install Android Studio" with a star rating of 5 stars. It includes instructions for Windows users, a video thumbnail showing a Windows desktop, and links for Contents, Windows, Mac, and Linux versions.

Join us on the livestream at [Android Dev Summit](#) on 7-8 November 2018, starting at 10AM PDT!

Meet Android Studio

- Overview
- [Install Android Studio](#)
- Migrate to Android Studio
- Configure the IDE
- Keyboard shortcuts
- Accessibility features
- Update the IDE and tools
- Workflow basics
- Manage your project
- Write your app
- Build and run your app
- Configure your build
- Debug your app
- Test your app
- Profile your app
- Publish your app
- Command line tools
- Troubleshoot
- Known issues
- Report a bug

Install Android Studio

Setting up Android Studio takes just a few clicks.

First, be sure you [download the latest version of Android Studio](#).

Windows

To install Android Studio on Windows, proceed as follows:

- If you downloaded an `.exe` file (recommended), double-click to launch it.
If you downloaded a `.zip` file, unpack the ZIP, copy the `android-studio` folder into your `Program Files` folder, and then open the `android-studio > bin` folder and launch `studio64.exe` (for 64-bit machines) or `studio.exe` (for 32-bit machines).
- Follow the setup wizard in Android Studio and install any SDK packages that it recommends.

That's it. The following video shows each step of the setup procedure when using the recommended `.exe` download.

Contents

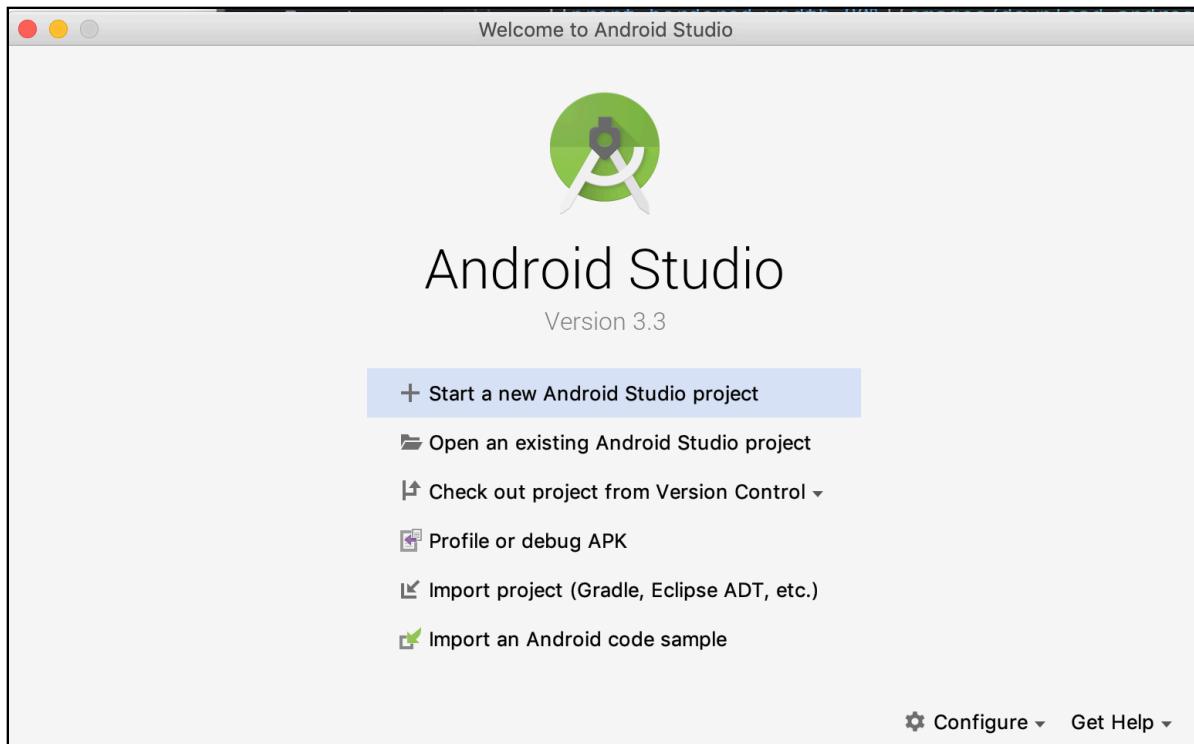
Windows

Mac

Linux

Follow the instructions and/or video that's relevant to your operating system until the “Welcome to Android Studio” window appears.

Note: If you don't have a high-speed internet connection, it can take a while for all of the components to download. Also, depending on how your system is set up, you may need to enter your password or an administrator's password to allow the installation to complete.



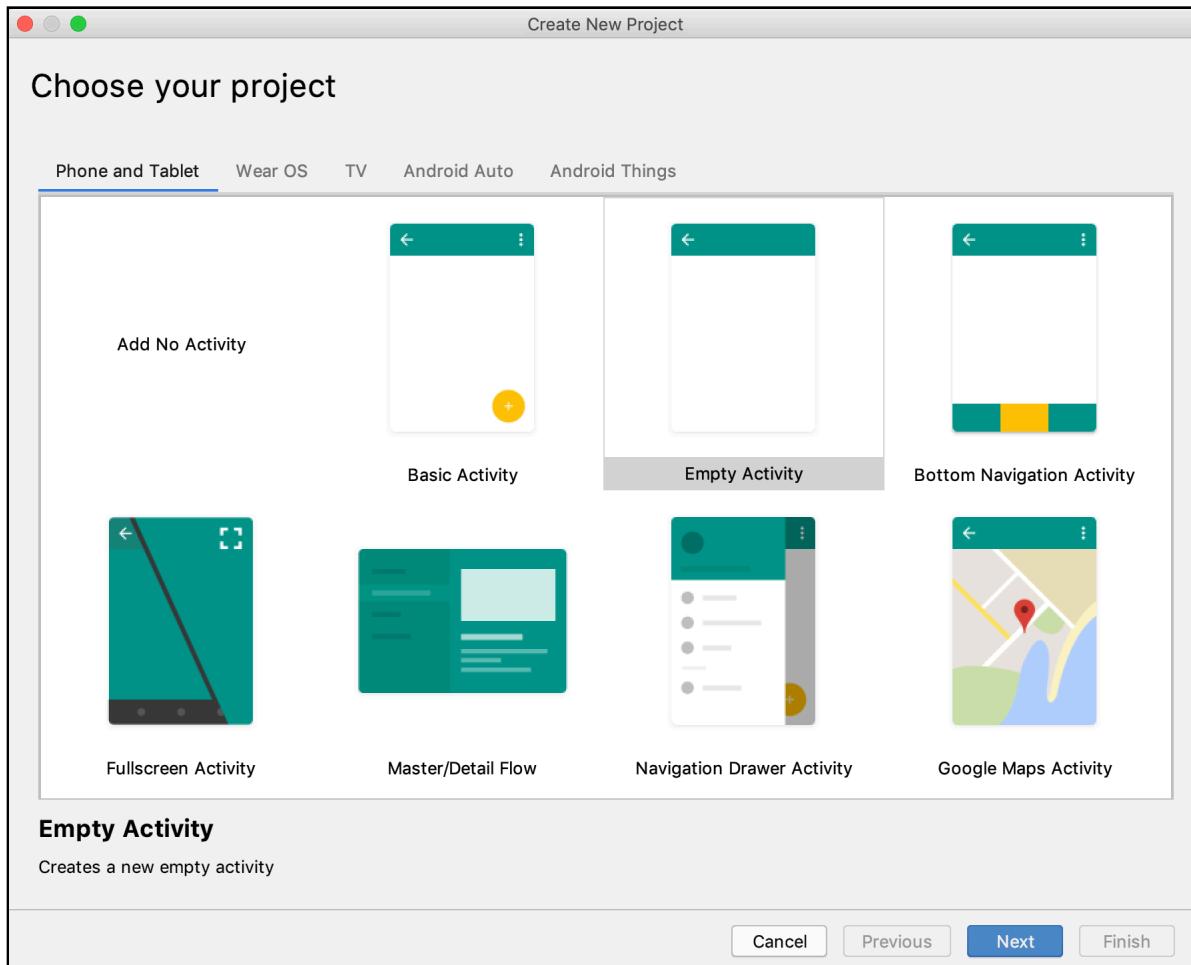
Your first Android project

Now that you have Android Studio installed, it's time to create your first project. It's important to note that this chapter focuses on getting your app running as quickly as possible. As such, you'll encounter a few screens that you won't understand; but don't worry, you'll get a chance to experience those screens in detail when you get to Chapter 6, "Creating a New Project".

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

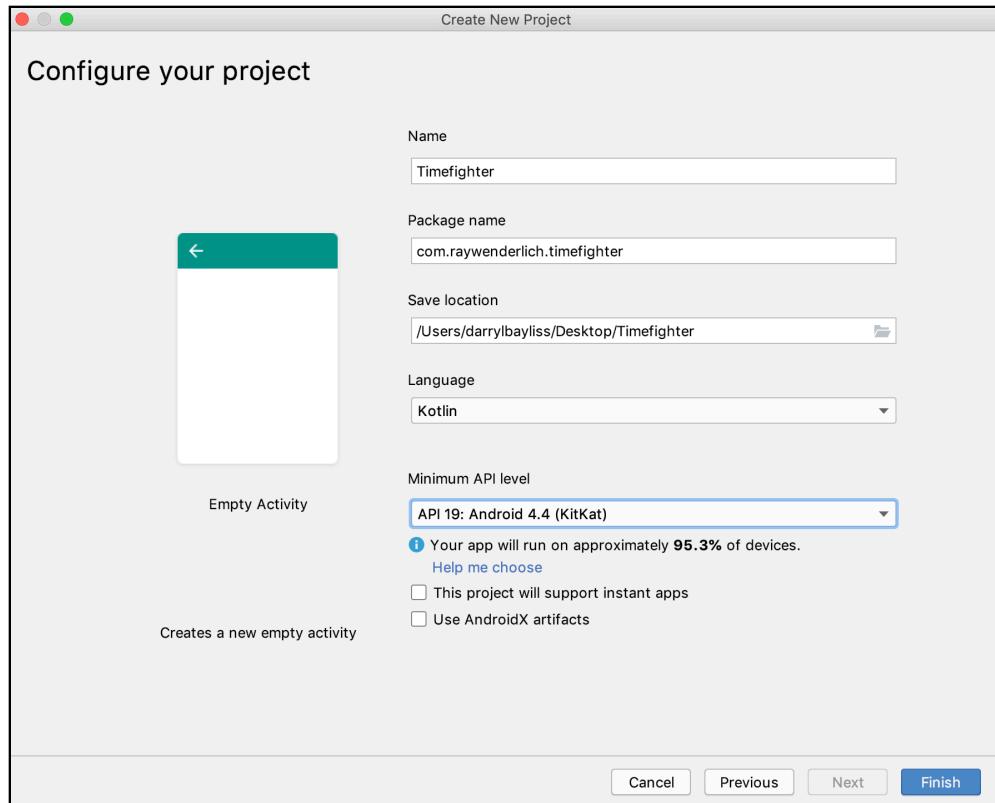
Start a new Android Studio project

The Welcome screen disappears, and a new window takes its place. This is where you choose the type of device you want your project to support.



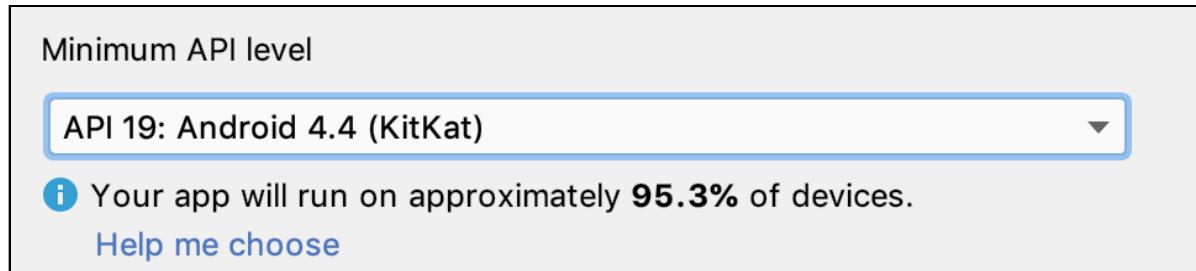
By default, **Phone and Tablet** is selected, which shows a variety of screen types. Click **Empty Activity** to choose it as your preferred project setup.

Click **Next** in the bottom-right of the window to move on to the next step.



This screen asks for some important information:

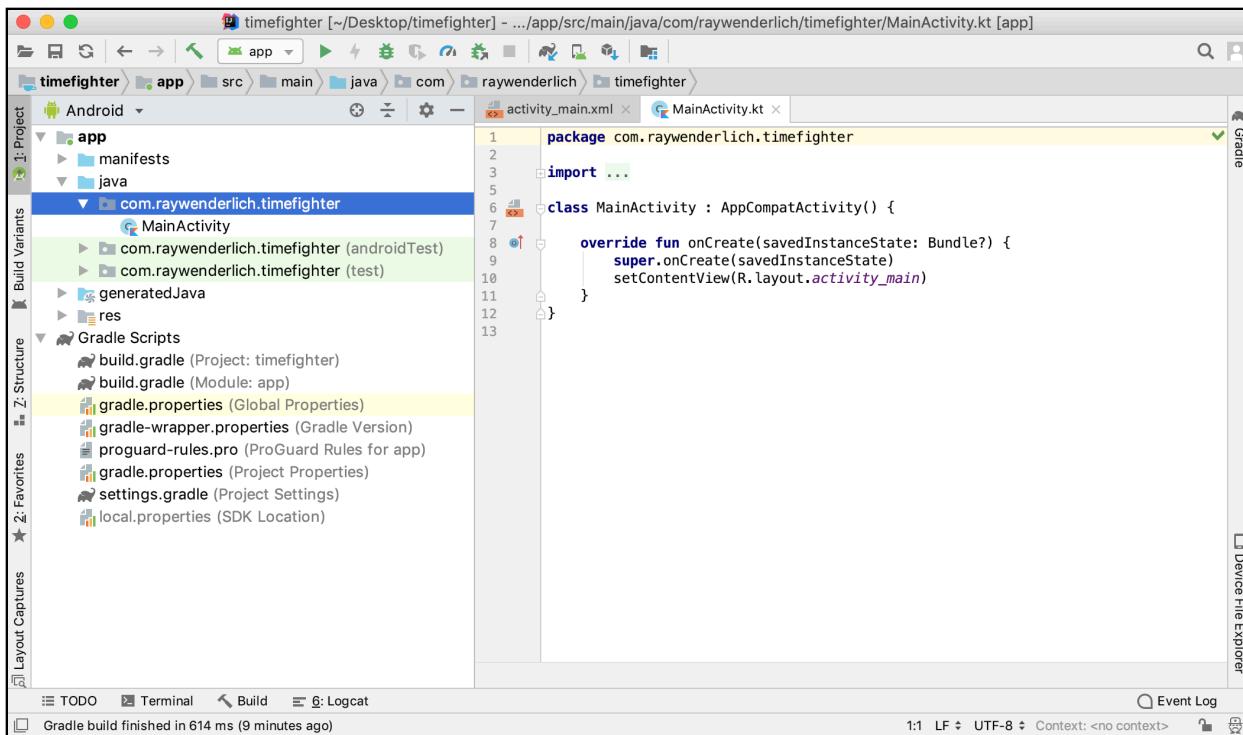
1. **Name** is where you enter the name of the app. For this project, name the app **Timefighter**.
2. **Package name** provides the app with a package name, a concept you might be familiar with from Java or Kotlin. Verify this field reads **com.raywenderlich.timefighter**.
3. **Save location** informs Android Studio of where to save the project. You can create your project anywhere you want. The folder button to the right of the field provides you with a system navigator to make your selection.
4. **Language** informs Android Studio of the language your project requires. Kotlin is already selected, so no action is needed here.
5. **Minimum API level** sets the minimum version of Android the app will support. This book requires apps to run API 19, or Android KitKat in English, so ensure **API 19: Android 4.4 (KitKat)** is selected.



6. **Finish**, the button in the bottom-right of the window, completes the project setup. Click this when you're ready to move on.

Android Studio uses this information and gathers the required libraries and resources to generate a new project. Depending on your internet connection, this phase may take some time.

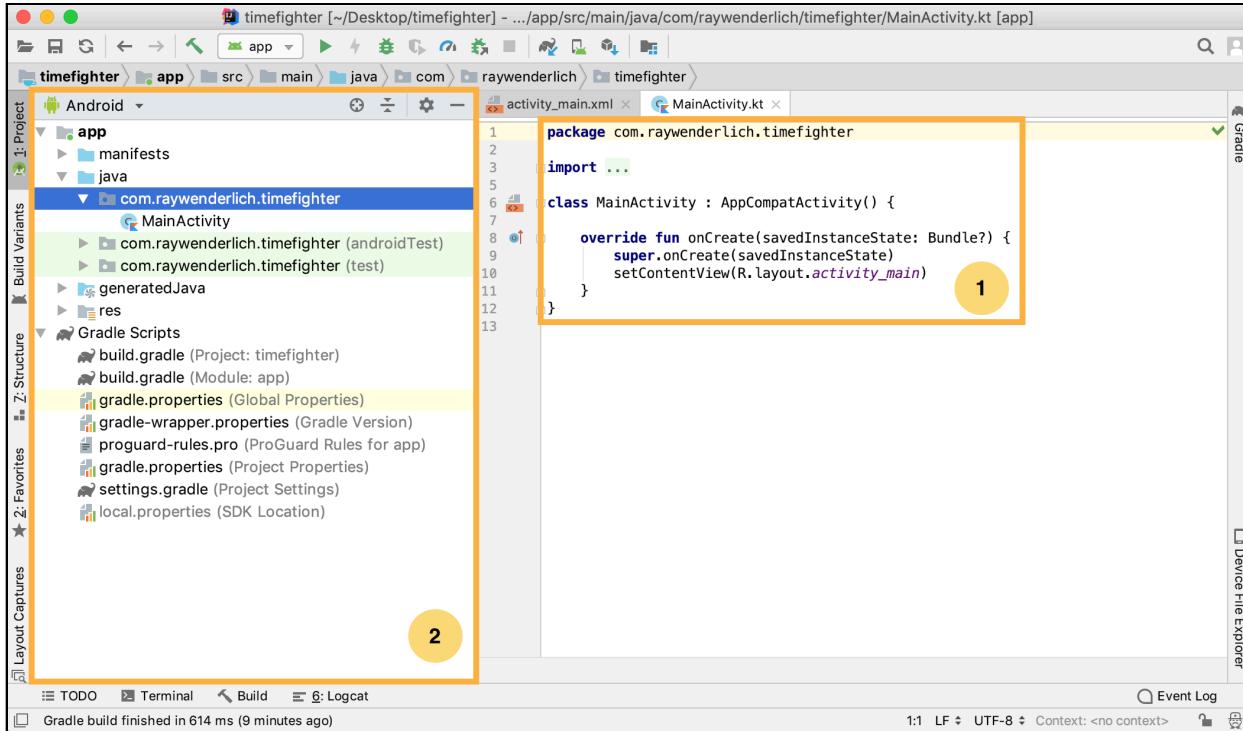
When that process finishes, Android Studio displays the newly created project with **MainActivity.kt** and **activity_main.xml** already open:



Android Studio

With the project created, you're ready to work on it. But don't jump in so quickly. Android Studio is a complex piece of software. If you dive in too fast, you may find yourself lost.

Before you start building your app, take a moment to review what Android Studio has to offer.



1. The first window that appears is the **Editor**.

This window provides a space to edit your app's source code. It includes tools like syntax highlighting, auto-completion for methods and objects as well as the ability to set breakpoints in 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.

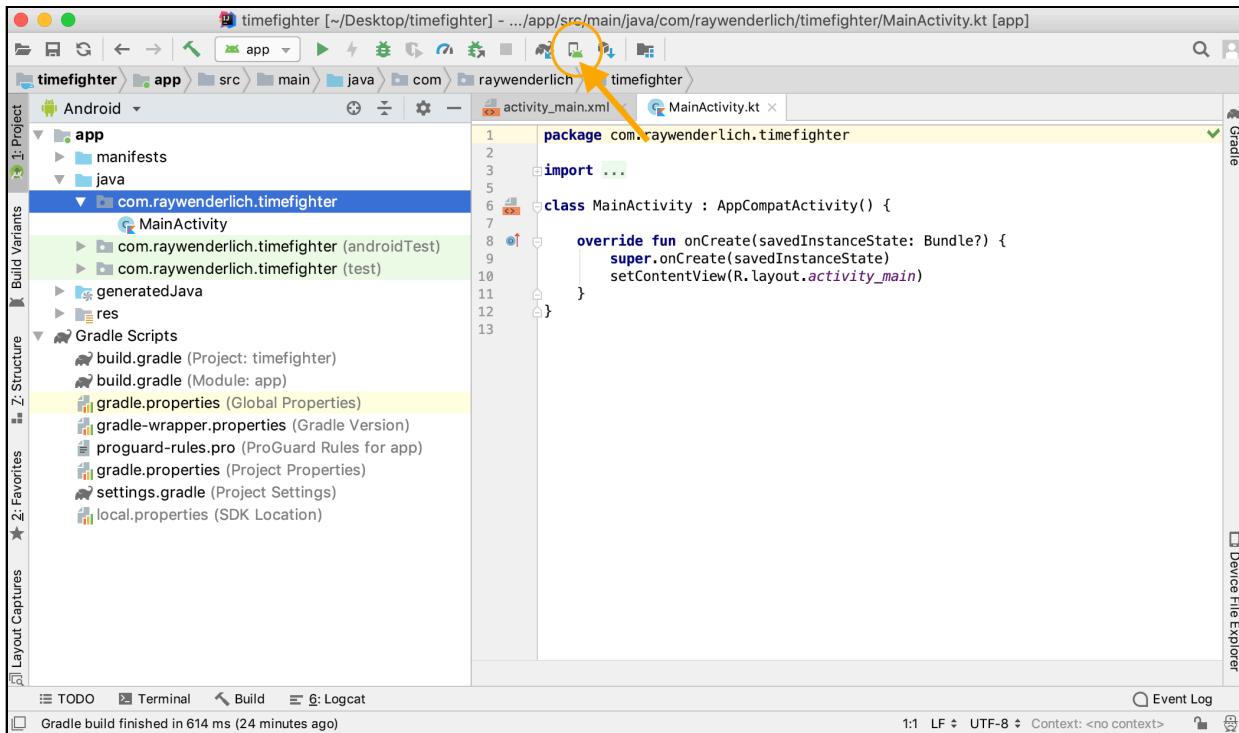
2. To the left of the Editor is the **Project navigator**.

This window shows everything your project contains, including code and image assets. Android Studio provides a lot; you can see what by clicking on the arrow to the left of the items in the Project navigator. For now, don't worry too much about these files; you'll get well-acquainted with them as you read through this book.

Creating an Android virtual device

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

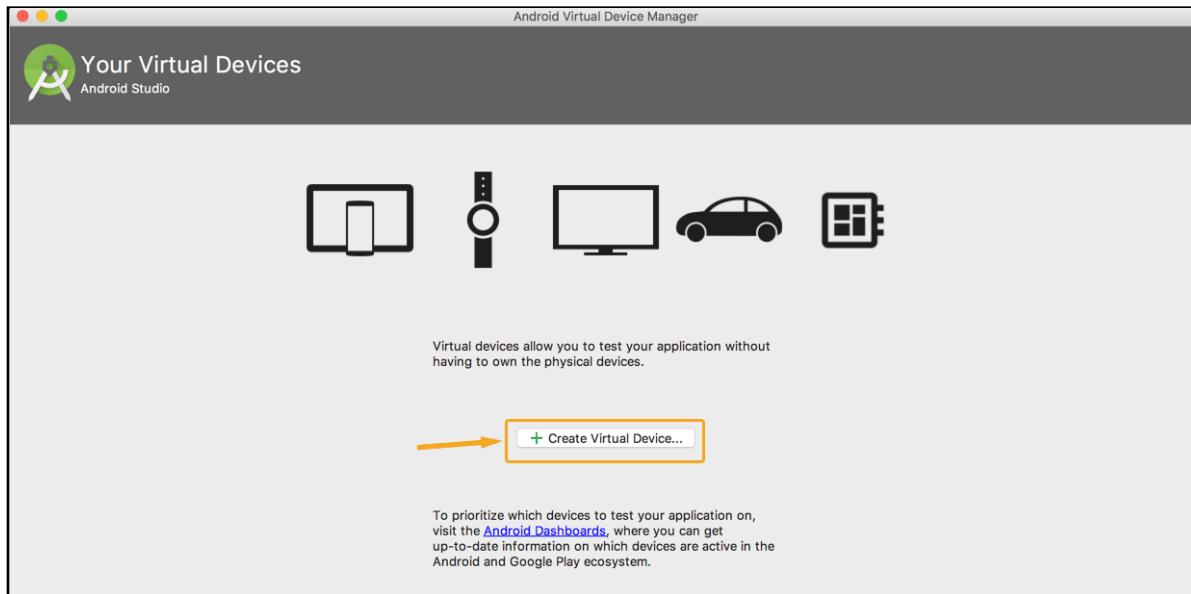
Looking at your editors and files is fine, but once you’re done writing code, you need to run the app to see it in action. Before you can do this, you need a device — whether it’s real or virtual. Look at the button highlighted in the following image.



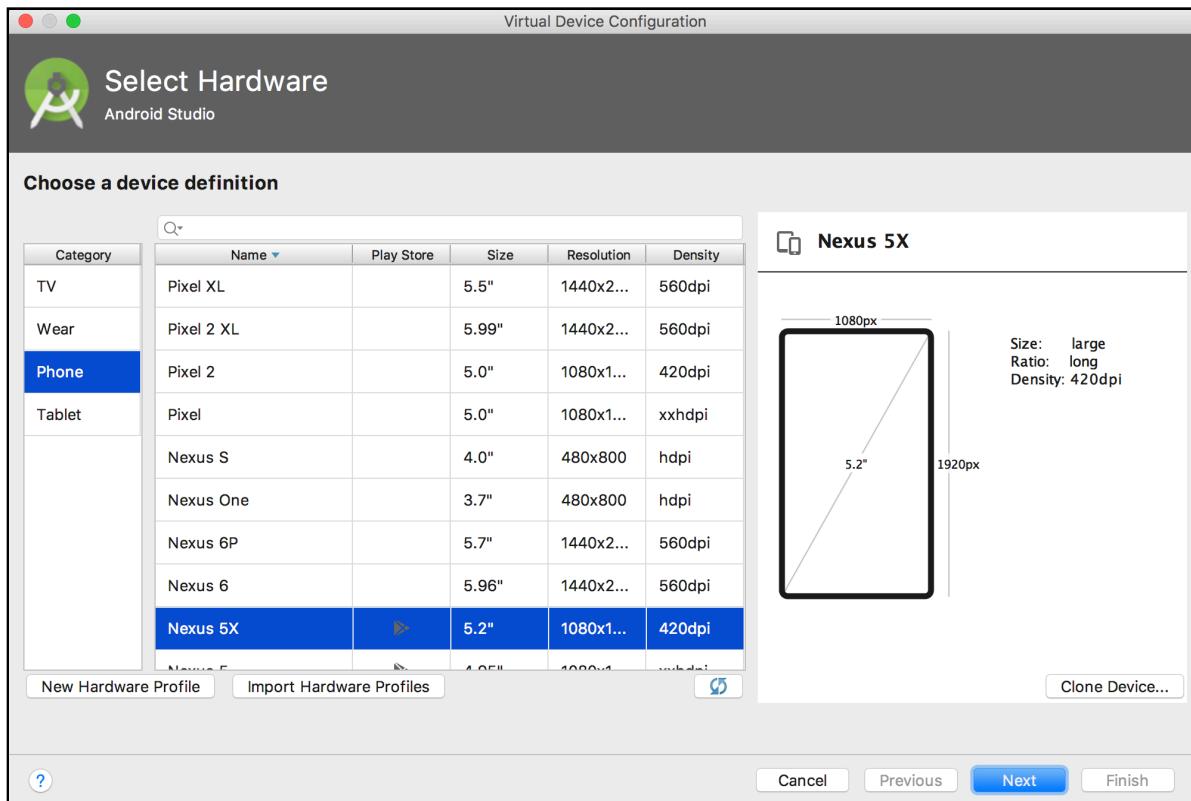
This button opens a new window that helps you create an **Android Virtual Device** (AVD). An AVD is an emulator that acts as a device on your computer. This allows you test your app without requiring a physical device. If you don’t have a physical device for testing, you’ll need to create a virtual device before you can run your app.

Click **Android Virtual Device**. A new window appears.

This window shows all of the available AVDs that exist on your machine. At the moment, none are available yet because you just installed Android Studio.

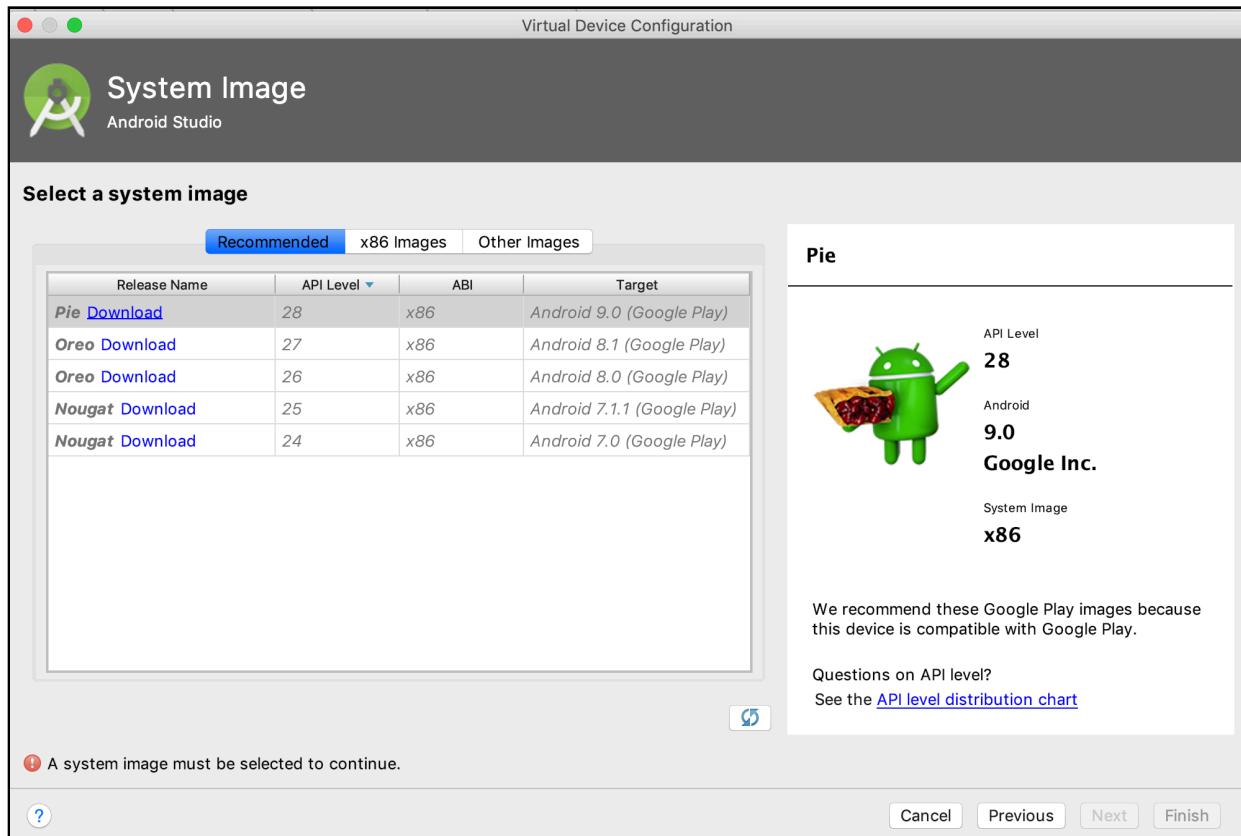


In the middle of the screen, click **Create Virtual Device** to show the **Select Hardware** window. In this window you can select the type of device you want your AVD to emulate.



A device is already selected for you, a Nexus 5X. This is an excellent choice because it closely emulates a real device that's used by a lot of people.

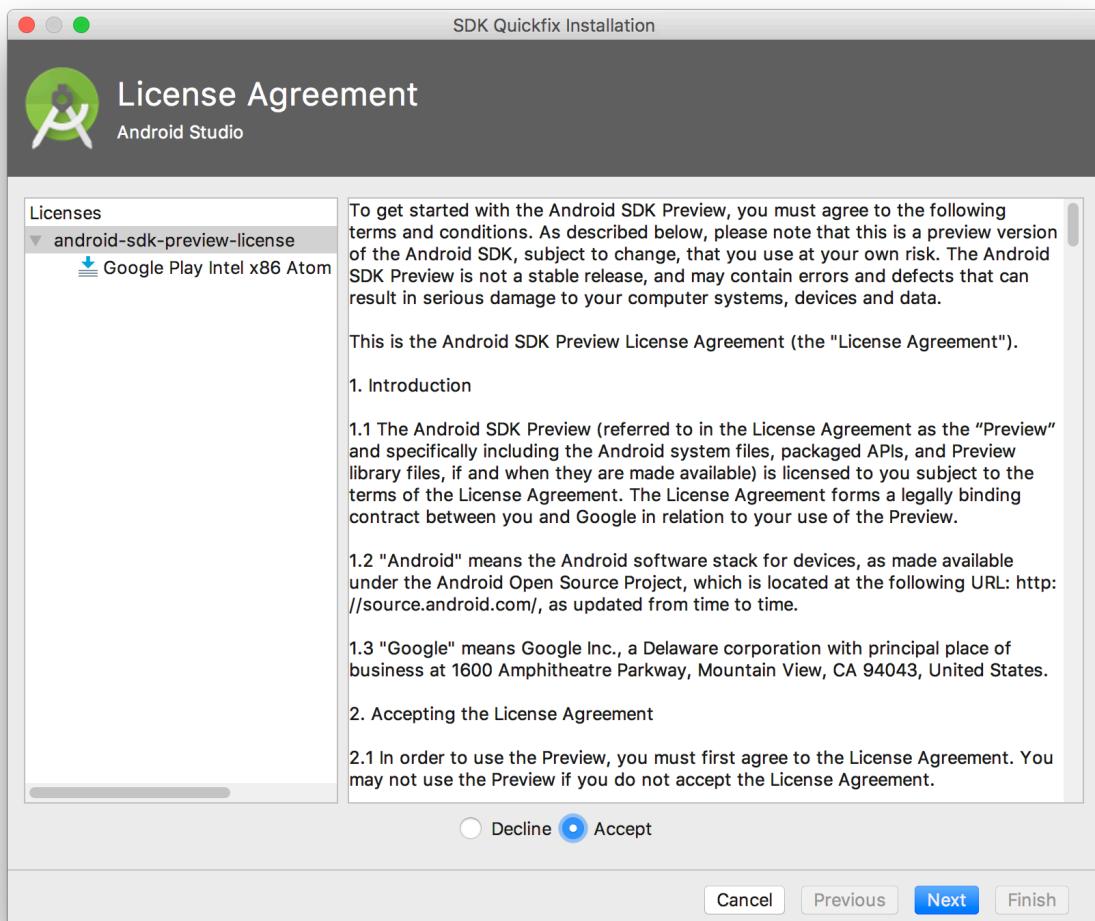
In the bottom-right of the window, click **Next** to progress to the **System Image** window. In this window, you can select the version of Android to run on your emulator.



There are a few tabs that run along the top of the list. The most interesting is **Recommended**, which shows a list of Android versions that Google recommends you use when testing your apps. Currently, those versions are grayed out because none of them are installed on your machine. You can change this by downloading the latest version of Android recommended by Android Studio.

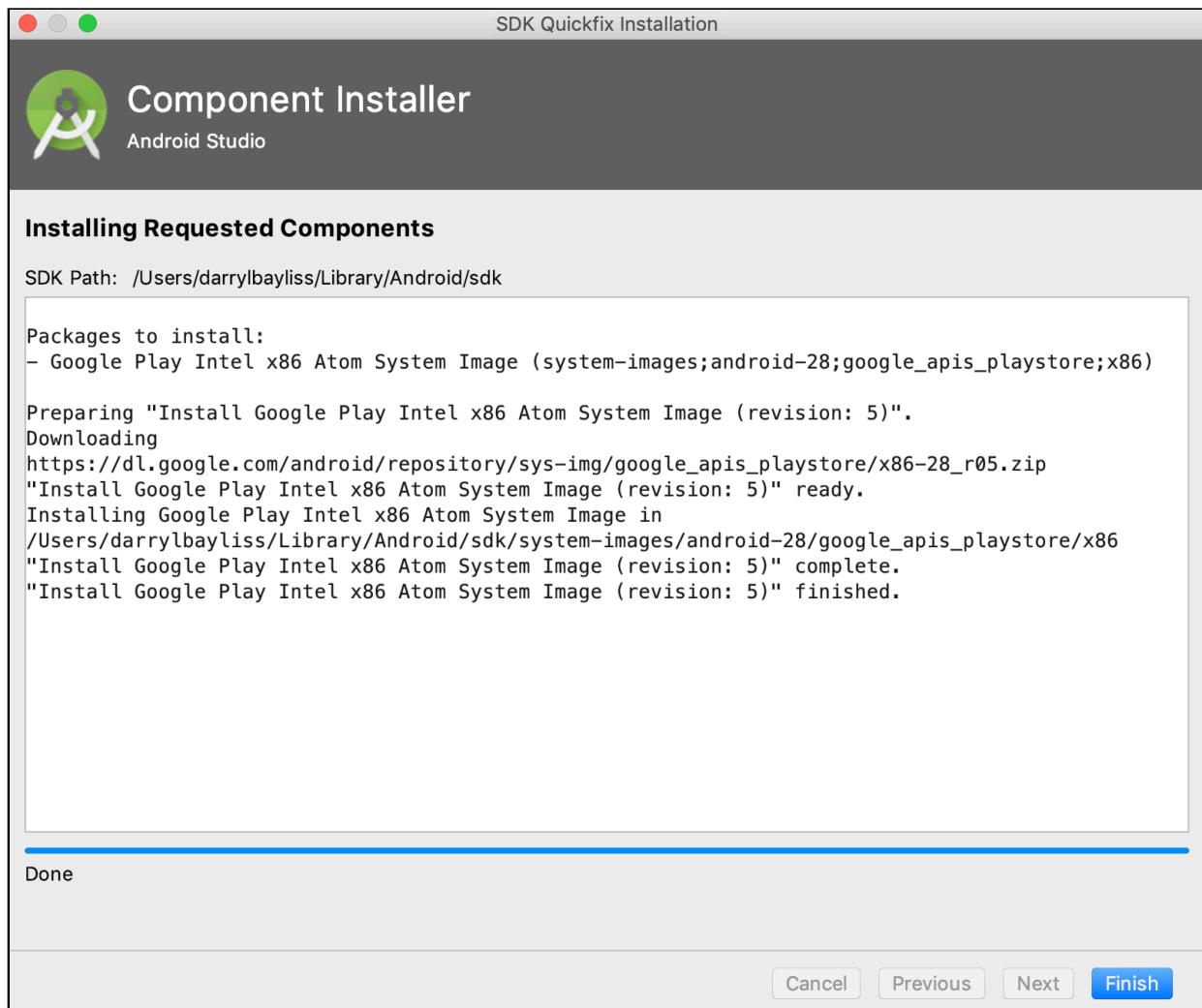
Select the top item in the table and click **Download** in the **Release Name** column.

A legal agreement appears describing the terms you need to agree to if you want to use Android in the emulator:



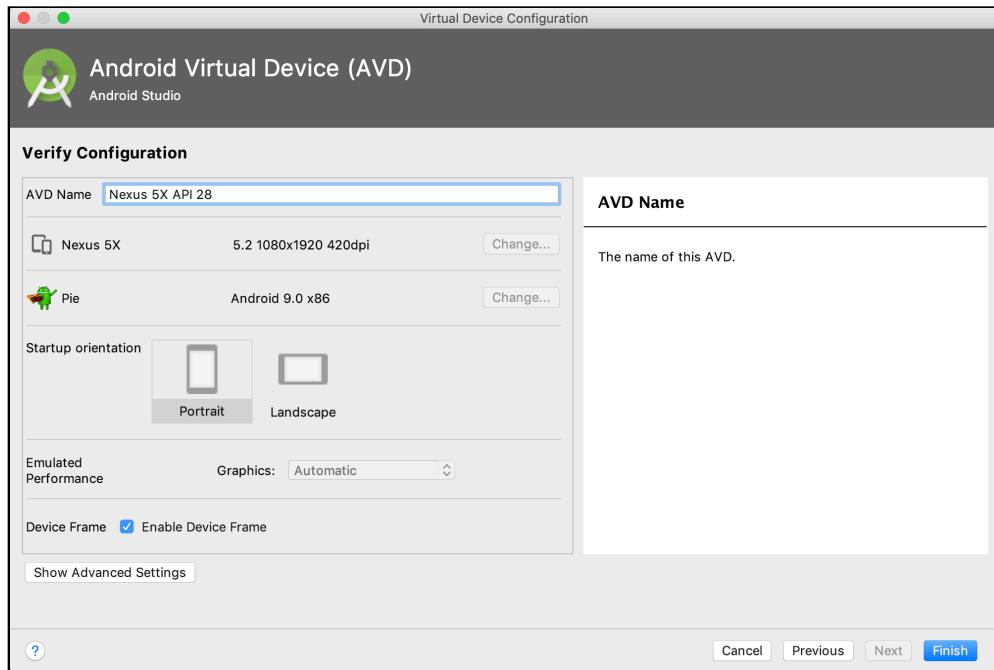
Select **Accept**, and then click **Next**.

The **Component Installer** window appears and automatically downloads the selected version of Android.



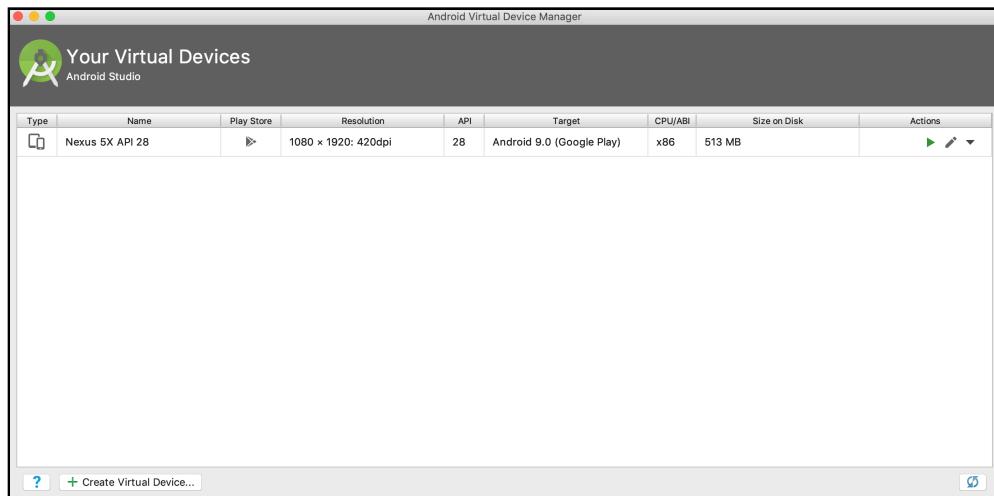
Once the download is complete, click **Finish** in the bottom-right. The Component Installer window disappears, and the System Image window appears once again. At this point, the latest version of Android is ready to use. To move on, click **Next** in the bottom-right of the window.

The next and final window for creating an emulated device is a summary of the characteristics the device will have.



This window gives you the opportunity to name the AVD and to confirm other aspects of the device such as the Android version. You don't need to change anything here, so click **Finish** at the bottom-right of the screen to create the AVD.

The current window disappears. In the AVD window that lists the available AVDs, your newly created AVD shows up and is ready for use:



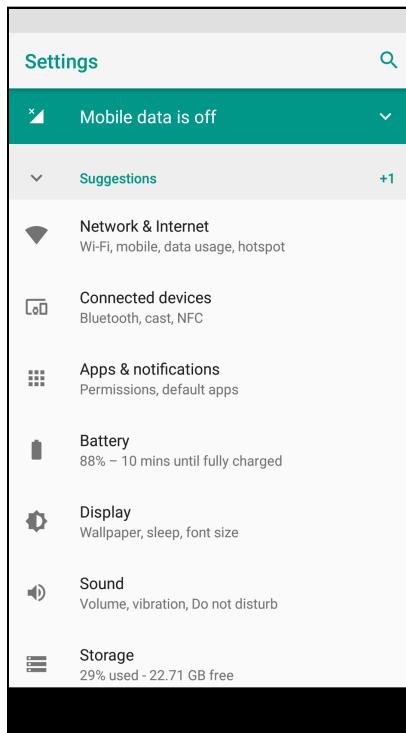
Setting up an Android device

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.

Before you can install Timefighter onto your device, you need to get your device set up for use with Android Studio. But first, you need to connect your Android device to your machine via a USB cable.

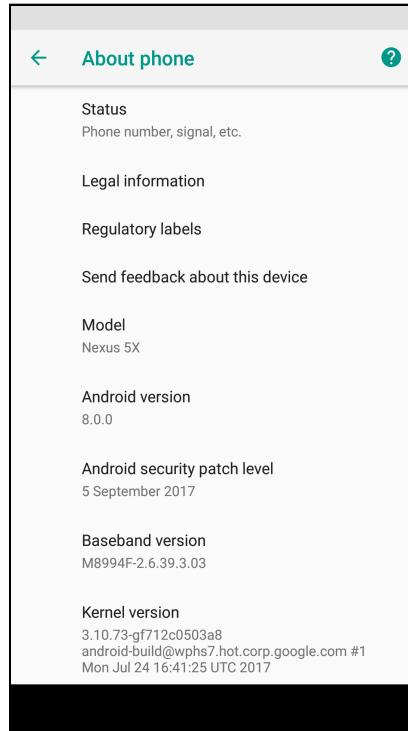
Note: If you're using a Windows machine, you need to download a USB driver for your device. You can download the driver and find instructions on installing it at <https://developer.android.com/studio/run/oem-usb.html>.

On your device, open the **Settings** app.

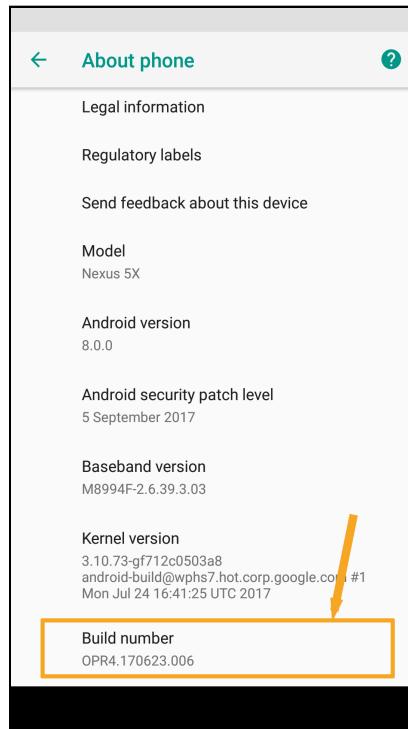


Note: If your device is running Android 8.0 (Oreo) or higher, tap **System** first to find the **About Phone** section.

Scroll through the settings until you find **About Phone** and tap it.



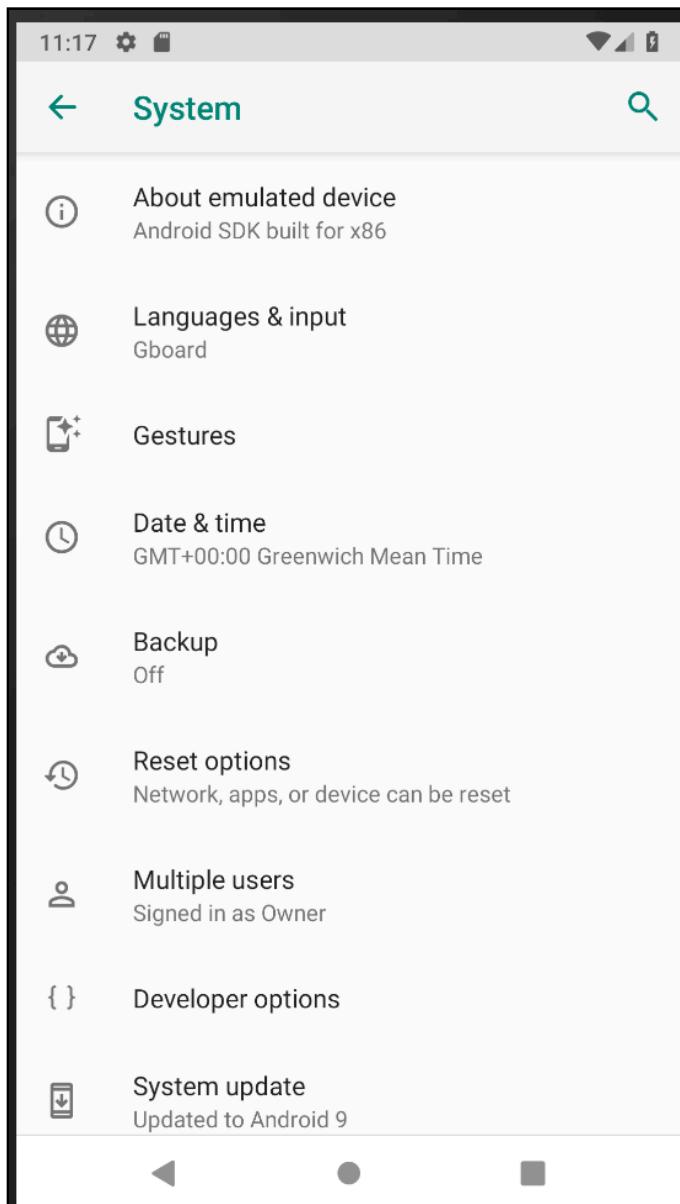
Now for the magical part: Scroll to the bottom of the **About Phone** screen, until the build number item appears:



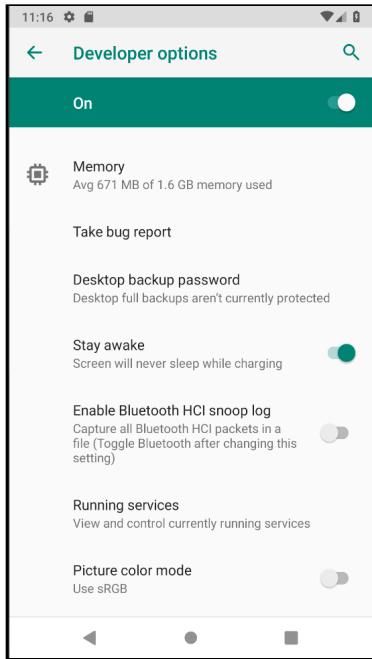
When you see the build number item, tap it several times until a message appears that informs you that you're a few steps away from being a developer. Keep tapping away until another message appears letting you know that you've become a developer.

Note: If your device is locked with a PIN, you'll need to enter it before you can reach this stage.

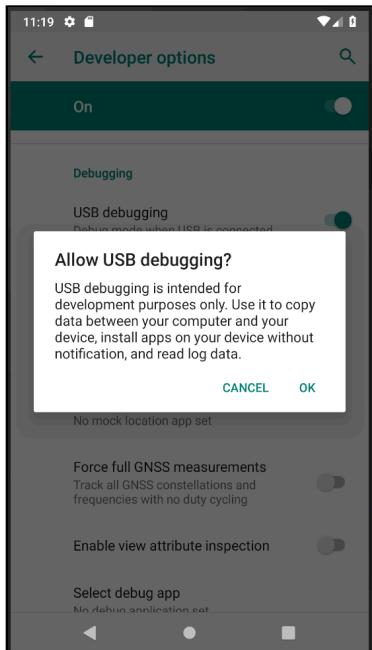
So what did this magical button do? Tap the back button to go to the previous Settings page. Do you notice anything different?



A new item shows up titled **Developer Options**. Tap this new option to review all of the developer features available.

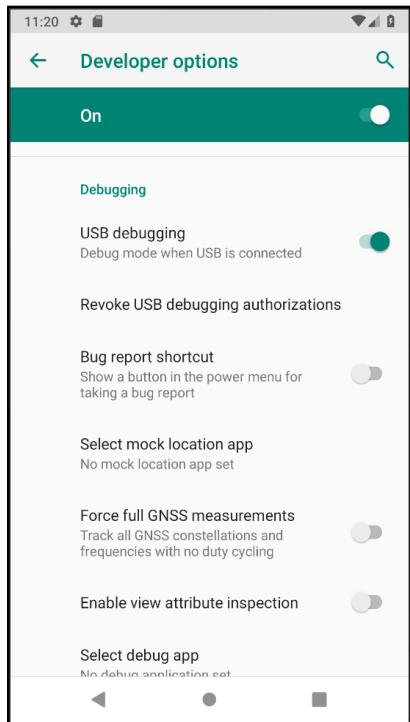


There's a lot to go through here. The only option you need right now is **USB Debugging**. Scroll down to this option and toggle it on. A dialog appears informing you of the intended usage of USB debugging.



Granting USB debugging privileges is a potential security hole, so most devices have this turned off by default. However, since you need to install apps over USB using Android Studio, you'll need to turn this on.

When you're ready, tap **OK** and the USB Debugging toggle is enabled.



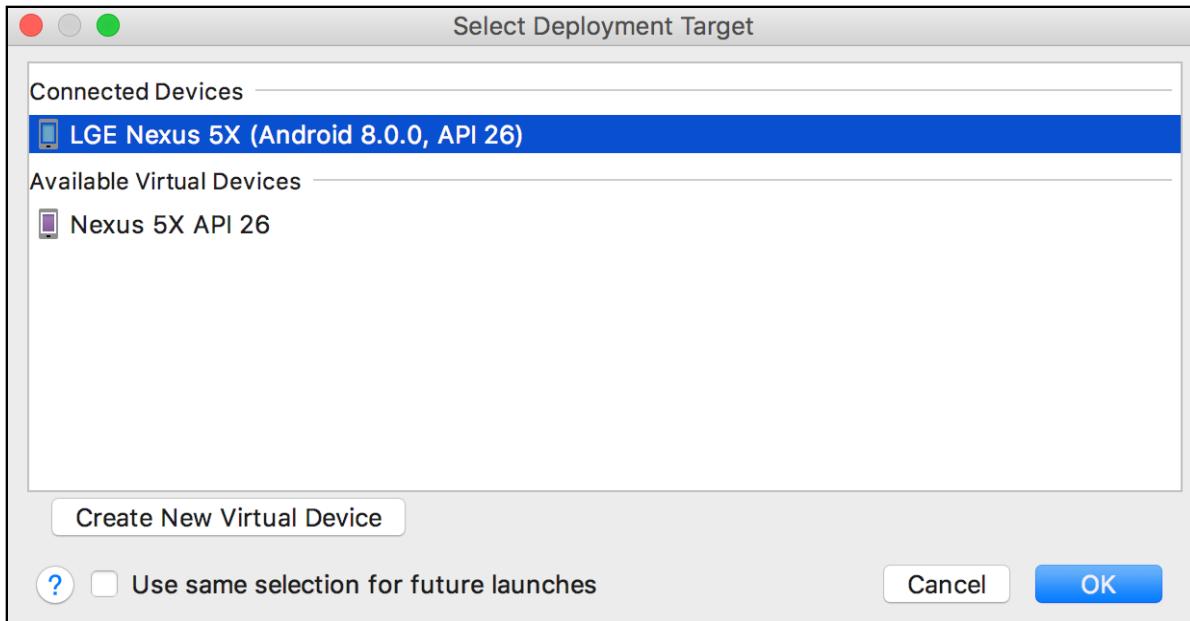
Congratulations, your device is now set up for development.

Running the app

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



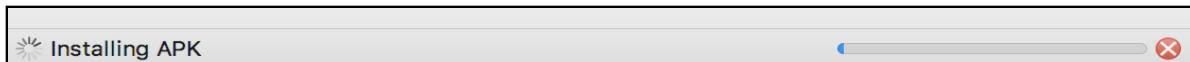
Click the **Play** button, and a new window appears.



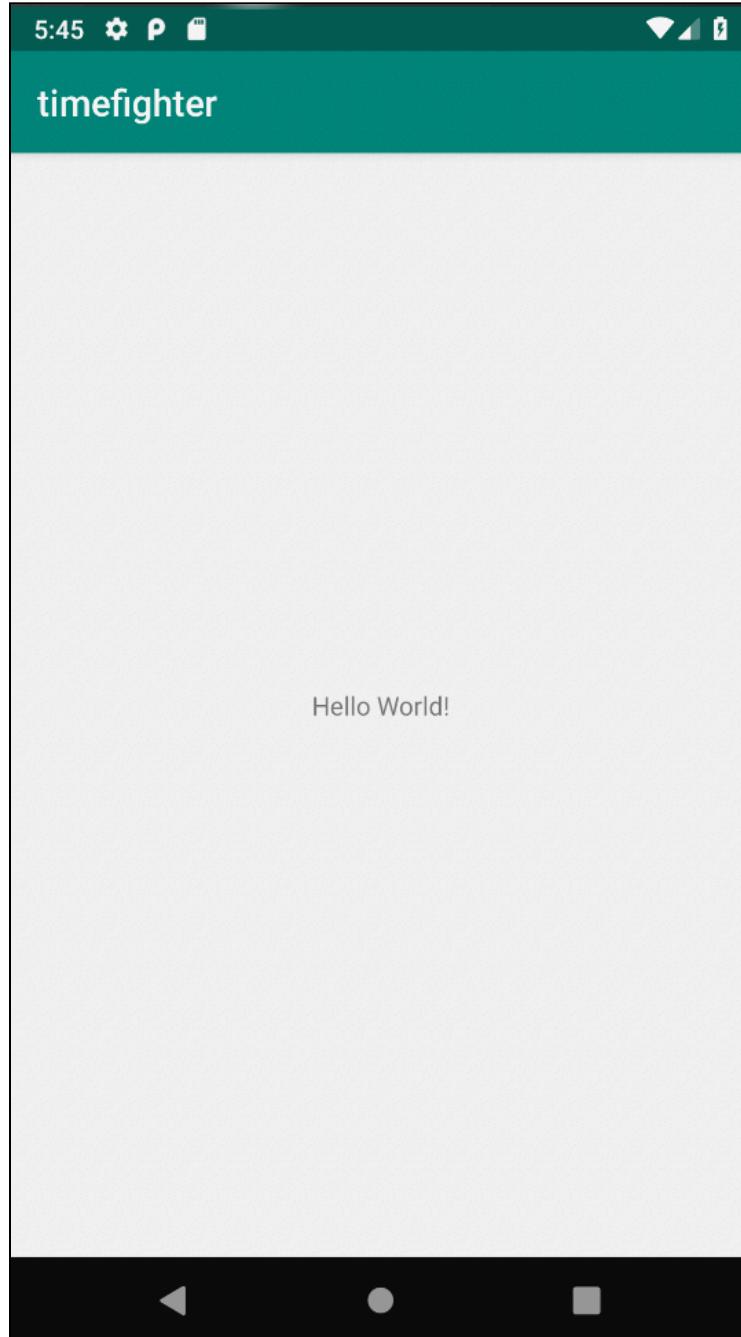
This is the **Select Deployment Target** window. It shows all of the available devices and emulators that you can use to run your app.

If you have a physical device with developer mode enabled, this appears in the **Connected Devices** section. If not, the emulator you set up is available in the **Available Virtual Devices** section.

Select either of the devices available and click **OK** in the bottom-right of the window. Android Studio begins building Timefighter and installs the app onto your device. You can see this happening at the bottom of Android Studio.

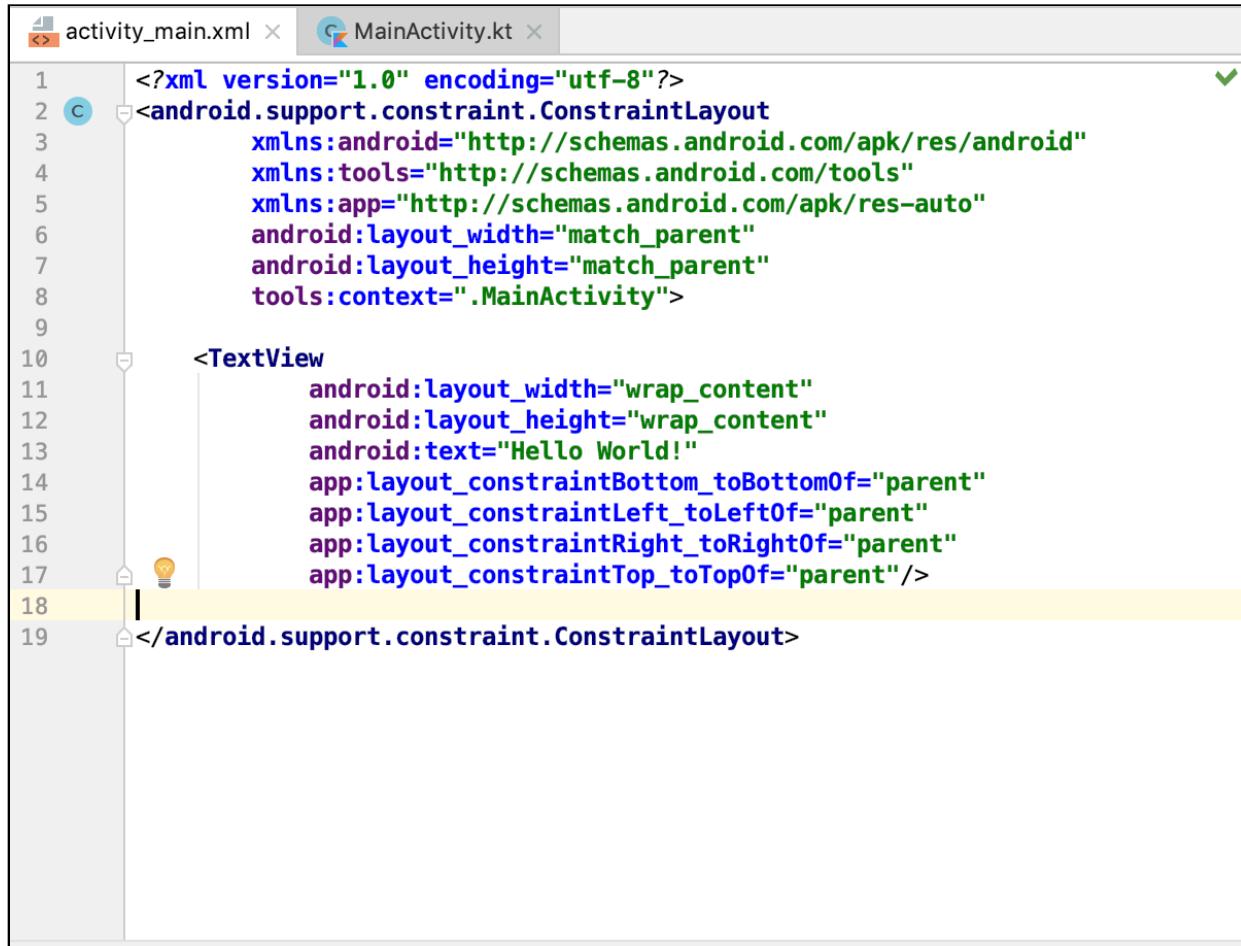


When Android Studio is finished building and installing, Timefighter will appear on your device:



Excellent, you just built your first Android App! Now it's time to make it a little more personal.

Back in Android Studio, in the Project navigator, open **app > res > layout > activity_main.xml**. Then, switch to the **Text** tab on the bottom.



```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    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:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"/>

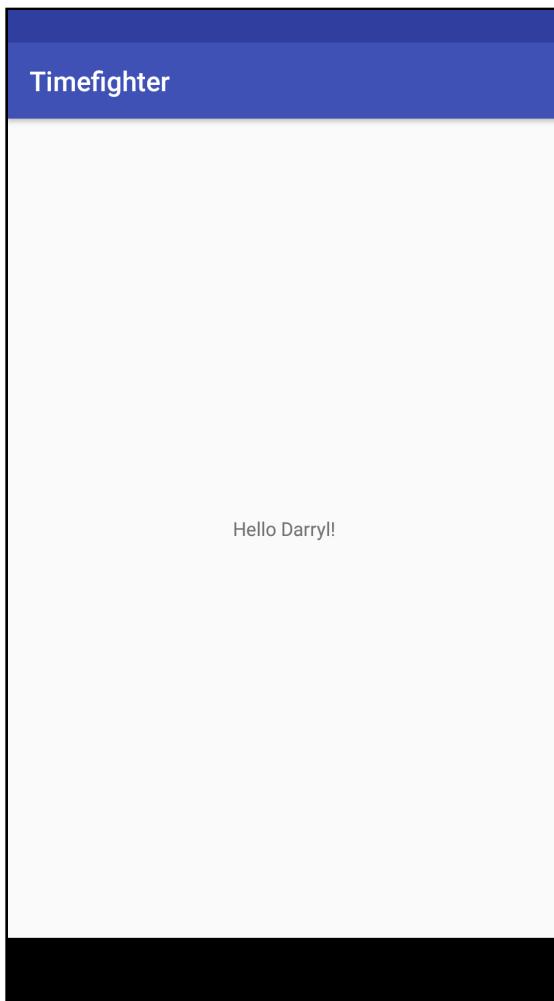
</android.support.constraint.ConstraintLayout>
```

Don't worry too much about what you see in this file. Right now, all you need to know is that it represents the app screen that appears on your device. You'll learn more about this in the next chapter.

Inside the **TextView** tag, update **android:text** from "Hello World!" to your own name:

android:text="Hello Darryl!"

To run your app again with the new setting, click the green **Play** button:



Installing new versions of Android Studio

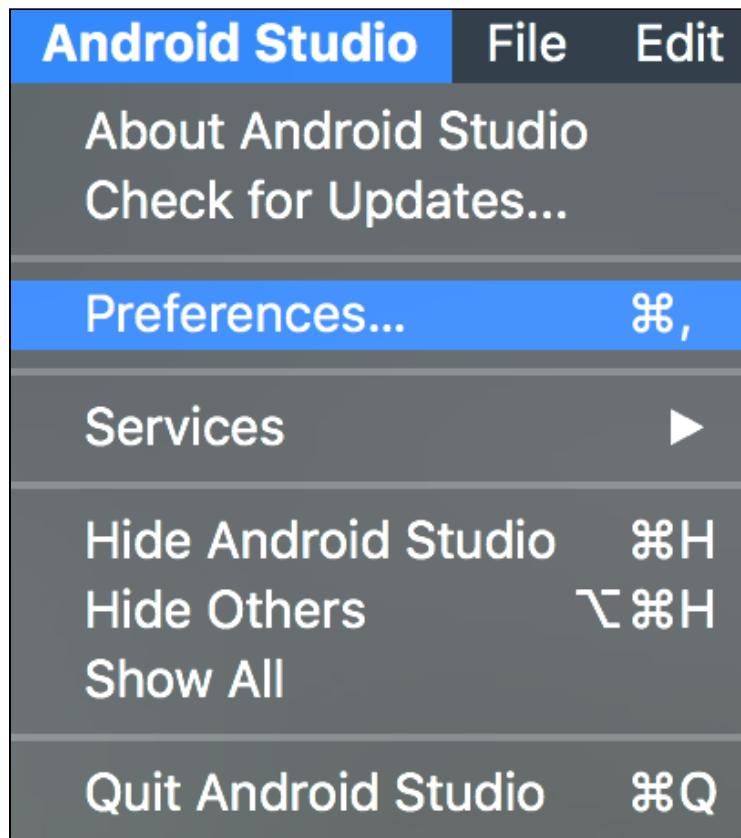
Note: This book assumes a specific version of the Android SDK is the most current: Android Pie API version 28. However, it's entirely possible that a new version exists. That said, you may want to install the latest versions of Android Studio and the Android SDK.

Google has decoupled Android Studio from versions of Android. This means you can build apps in Android Studio with any version of the Android operating system, including any future versions of the Android SDK.

Android Studio does its best to prompt you when new versions of either Android Studio or the Android SDK are available. But, you don't have to wait on Android Studio to update.

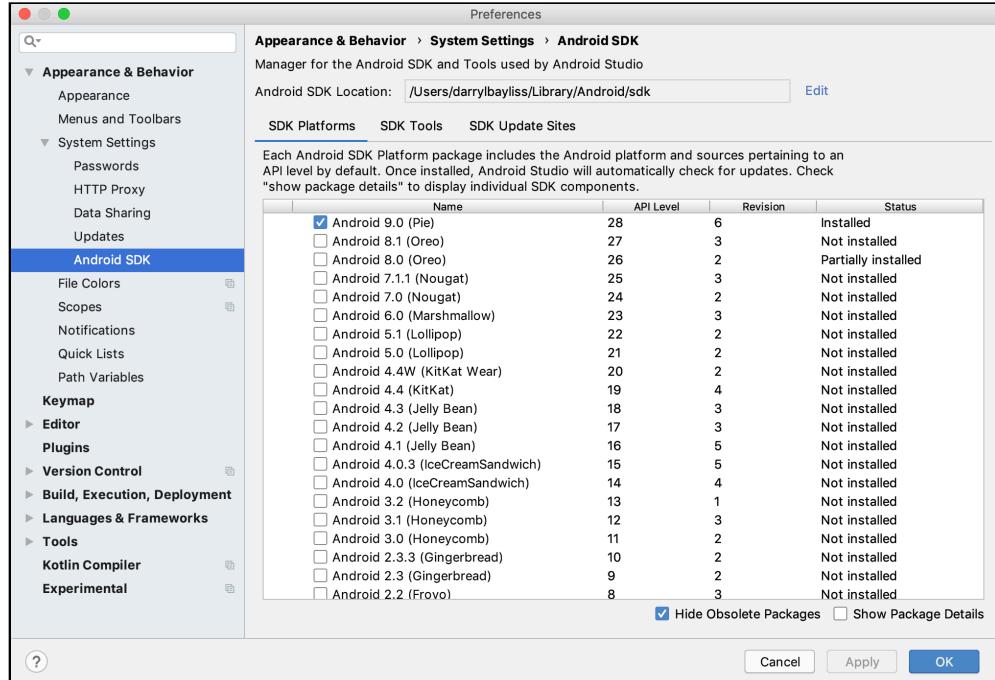
In the **Android Studio** menu, you can select **Check for Updates**, which gives you a dialog showing the items you can update. It also lets you know if things are already up-to-date.

If you want to download a newer — or even an older — version of Android SDK, select **Preferences...** from the same menu.



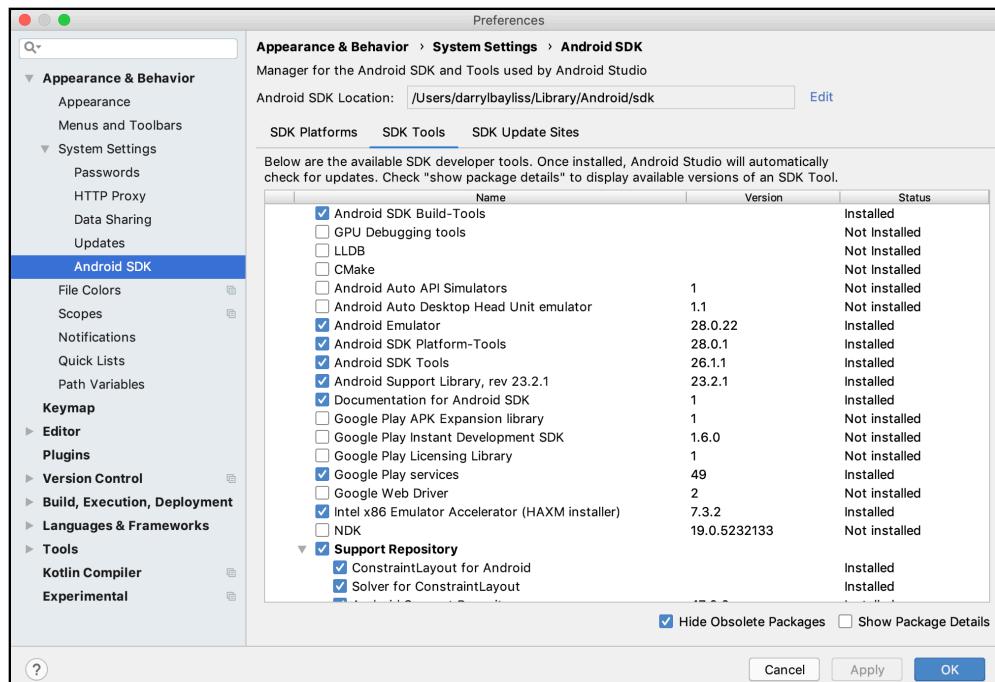
In the **Preferences** dialog, drill down through the menu items in the tree to **Appearance & Behavior ▶ System Settings ▶ Android SDK**.

In this window, there are two noteworthy tabs: **SDK Platforms** and **SDK Tools**. In **SDK Platforms**, you'll see a list of the available Android SDKs.



Check the box next to one or more of these SDKs, and then click **OK**; this will install the selected SDKs.

In **SDK Tools**, you'll see a list of the available build tools that Android Studio and your app can access.



Check the box next to one or more of the tools, and click **OK**; this will install that tool. You'll learn more about these tools later.

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 full-featured 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. They let you define how to present your app's user interface on the device. You can create Layouts in one of two ways:

1. Using an XML file to declare the user interface ahead of time.
2. Writing Kotlin code to create the layout at runtime programmatically.

In this book, you'll define your Layouts ahead of time using XML — that's 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 few moments to think about what makes up an app. Most often, an app is a self-contained program that lets its users perform one or more tasks. When you build an app, you want your users to accomplish those tasks quickly and intuitively, which is why having a well-thought-out user interface is so important.

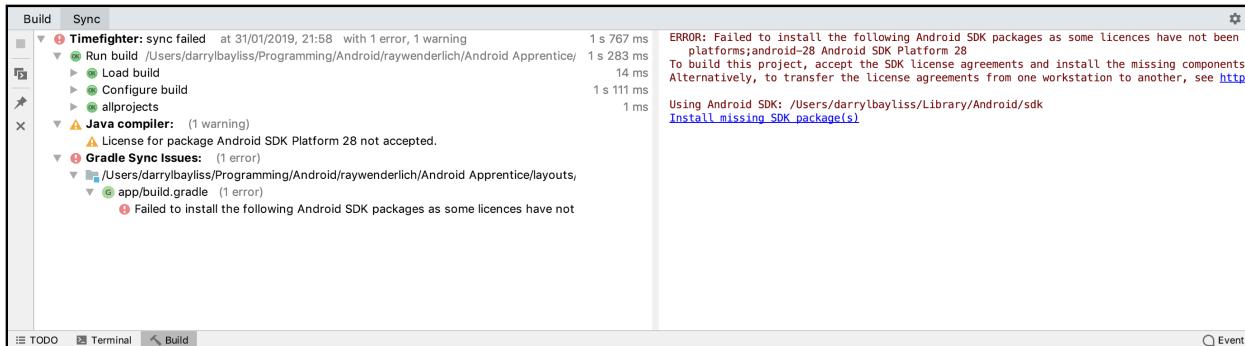
The app you'll build in this section, TimeFighter, is no different. It's minimal in its design, so usability isn't an issue.

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

Locate the **projects** folder for this chapter and open the TimeFighter app inside **starter**. The first time you open the project, Android Studio will take a few minutes to set up the environment and update its dependencies. After that, you're ready to rock and roll!

These are not the SDKs you're looking for

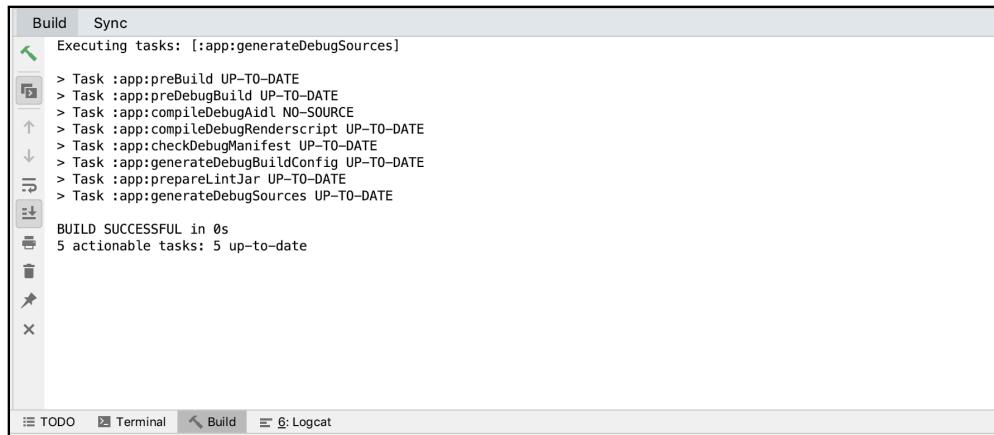
When you open the project, you might get the following error in the **Build** tab:



If you followed along in the previous chapter and installed a fresh version of Android Studio, you may not see this error. However, if you're already running Android Studio, it's possible that you don't have the version of the Android SDK that was used to create this project.

Do not fret young padawan learner; Android Studio will always do its best to help resolve these sorts of issues for you. On the right, Android Studio provides you with a convenient link, that when clicked, will install the required version of the Android SDK *and* rebuild your project.

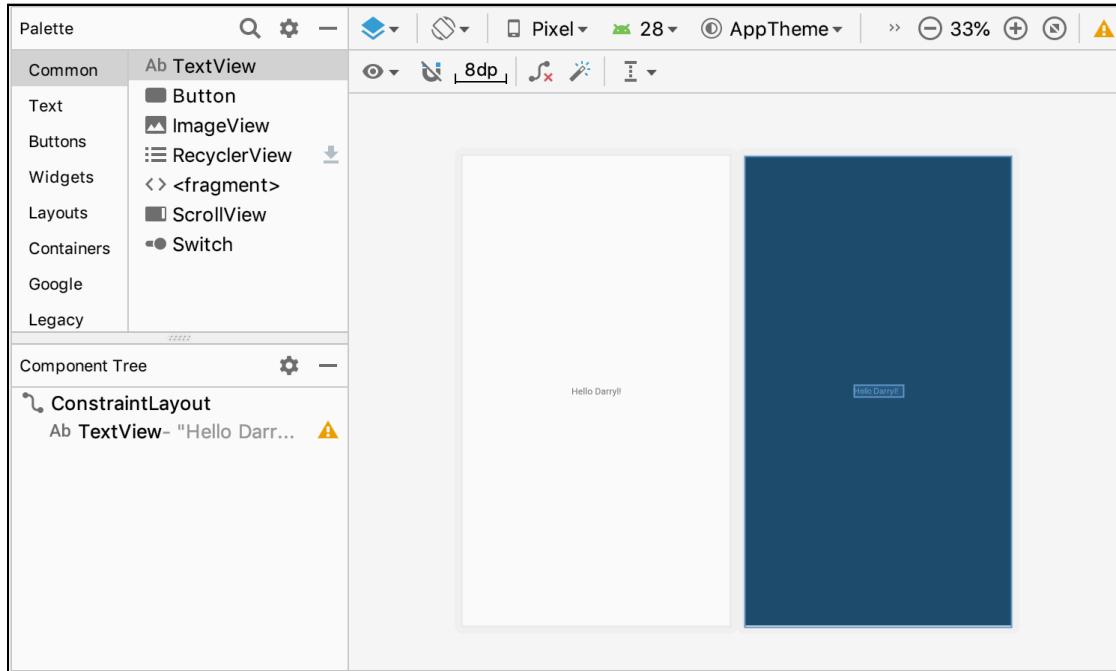
After this error disappears, you'll see the following in the **Build** tab:



Excellent! It's time to get comfy with everything Android Studio has to offer.

The Visual editor

In the project structure sidebar on the left of Android Studio, expand **app**, **res**, and **layout**. Then, double-click **activity_main.xml** and you'll see a screen that looks like this:



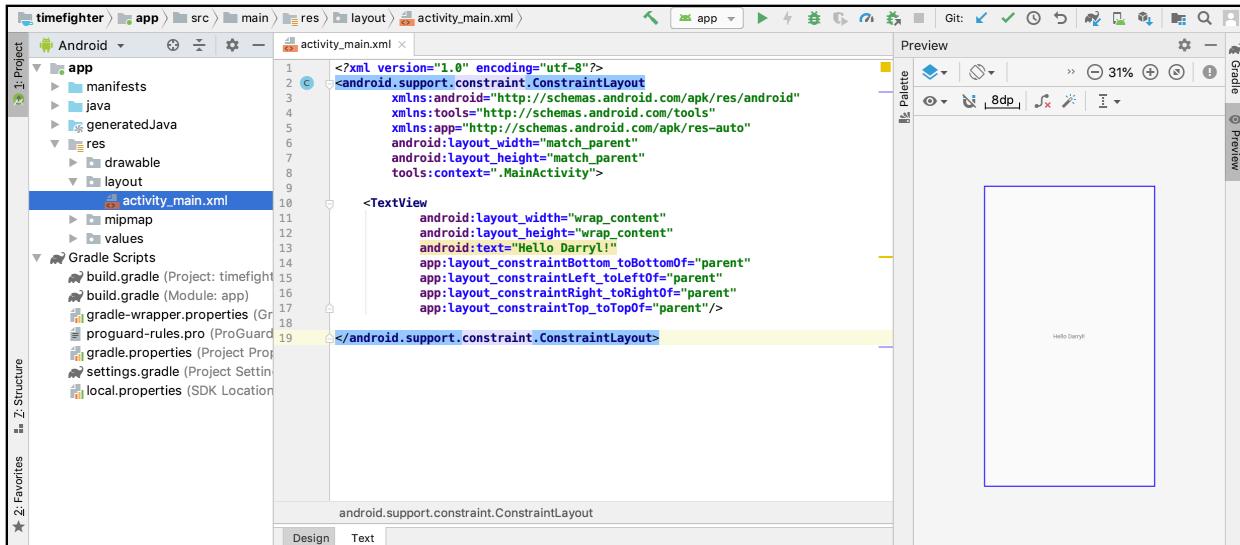
Editing activity_main.xml in Visual Editor

Behold the **Visual editor**!

In **design mode**, the middle of Android Studio shows a few different screens.

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

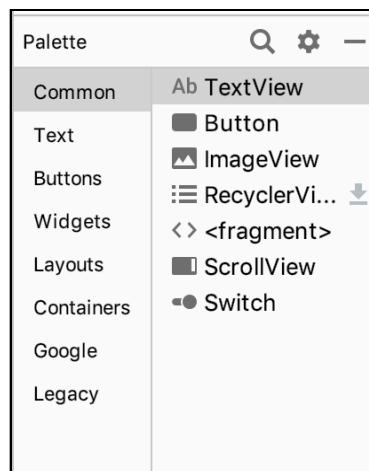
At the bottom of the Visual editor, you'll find two tabs: **Design** and **Text**. Click **Text**, and you'll see this:



Editing activity_main.xml in XML Editor

In the middle section of Android Studio, you'll see the **Text editor**. This shows the XML representation of the app's first screen. You can create the interface here if you like, but using the Design tab provides you with more visuals.

Click **Design** 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**:

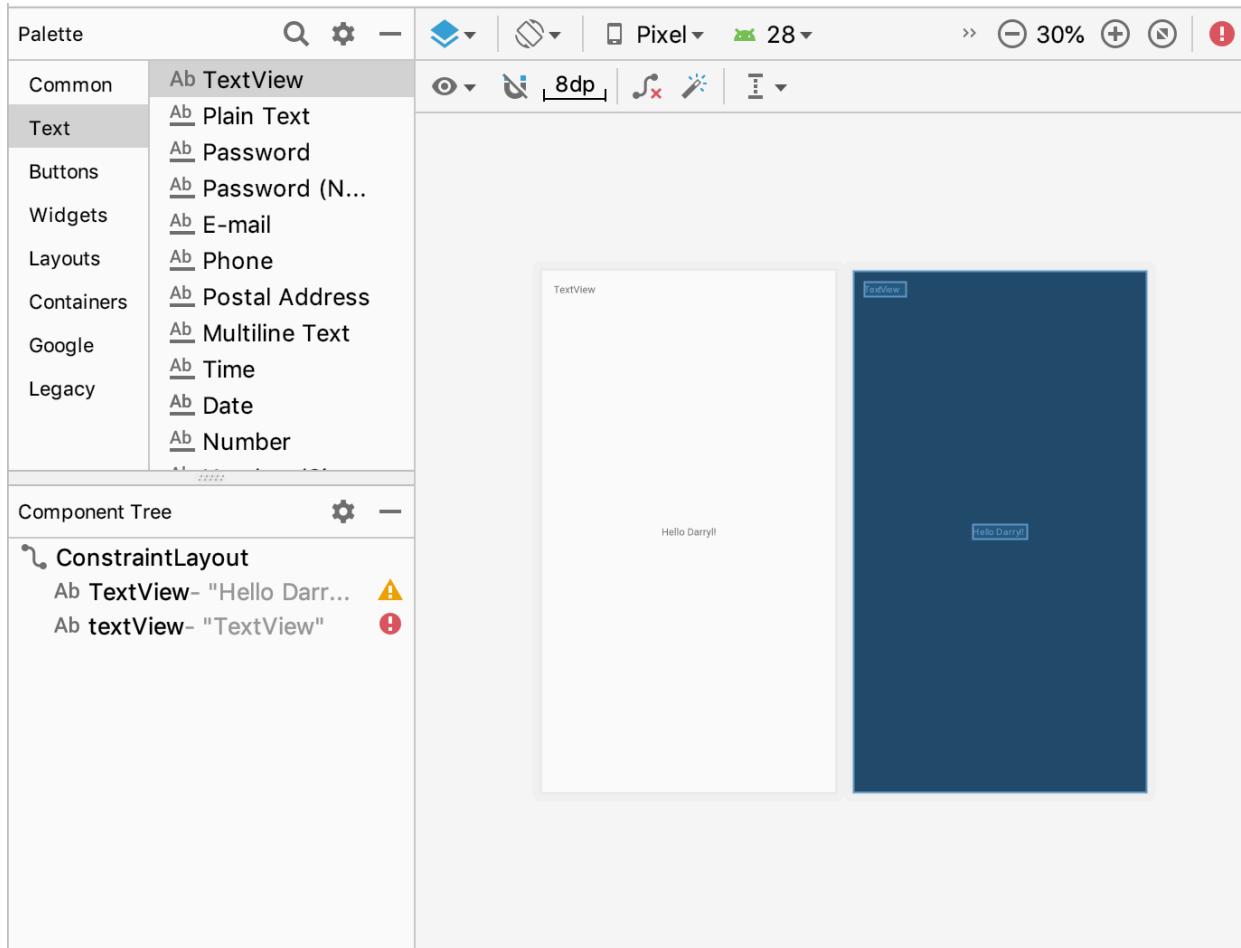


Palette of interface components

This contains all of the built-in user interface components that 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 directly into the Preview screen to add a component.

Open the **Palette** and select **Text**. The palette changes and shows everything text-related.

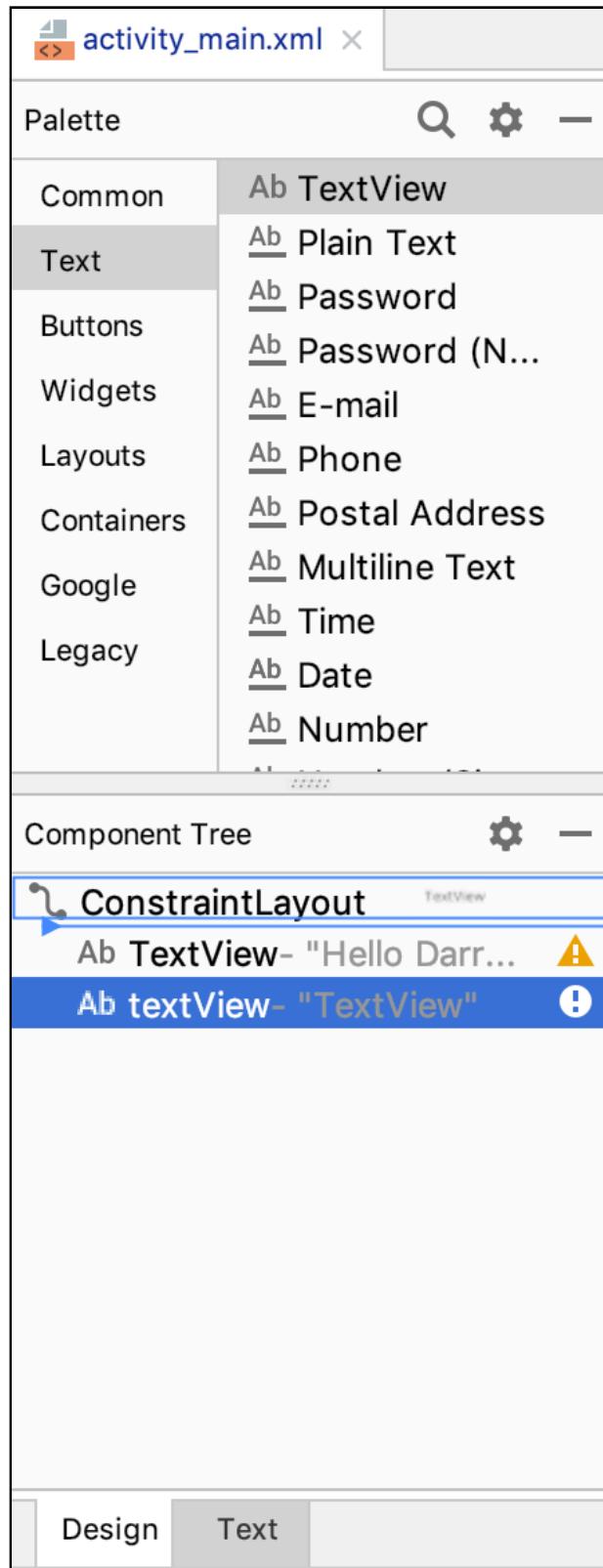
Next, drag a **TextView** from the Palette — this is for your score label — and drop it in the top-left of the Preview screen.



Component tree view

Before moving on, it's worth noting that although dragging and dropping components into the Preview area is relatively easy to do, it can be tricky to get things to show up in the right spot, especially when you're dealing with projects that have many views.

As an alternative, you can drag components from the Palette directly into a **Component Tree**, dropping it underneath the desired parent component.



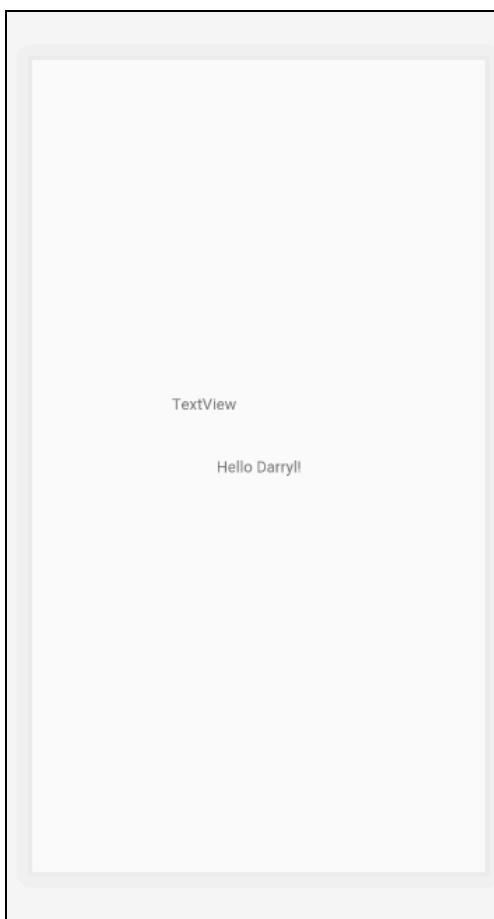
Keep that little feather in your cap as you progress through this book, because you may find it easier to drop components in this way, and then deal with positioning them later.

Positioning your views

At this point, you have the start of your app, with your `TextView` sitting in the top left-hand corner. Come to think of it, how does the device know where to position the `TextView`? What happens if someone rotates the device?

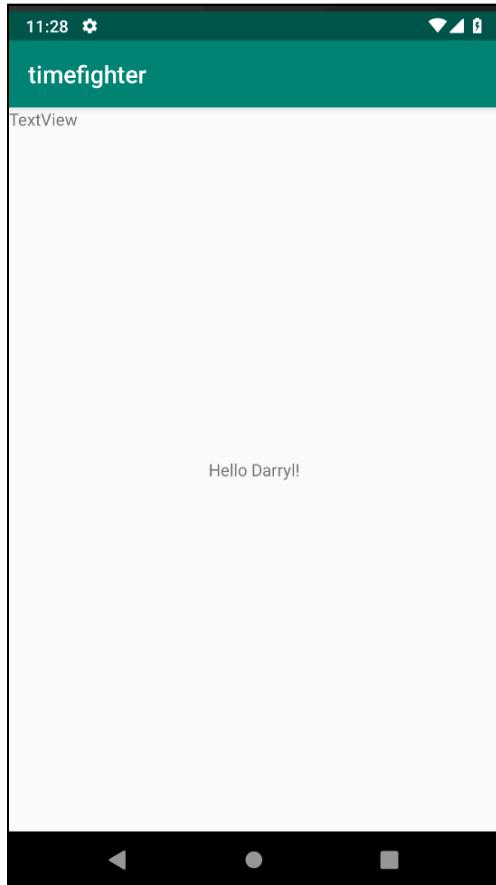
As it stands now, the app doesn't know where to place the `TextView` — and you can prove it!

In the Visual editor, drag the 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 the emulator.



Layout as seen on device

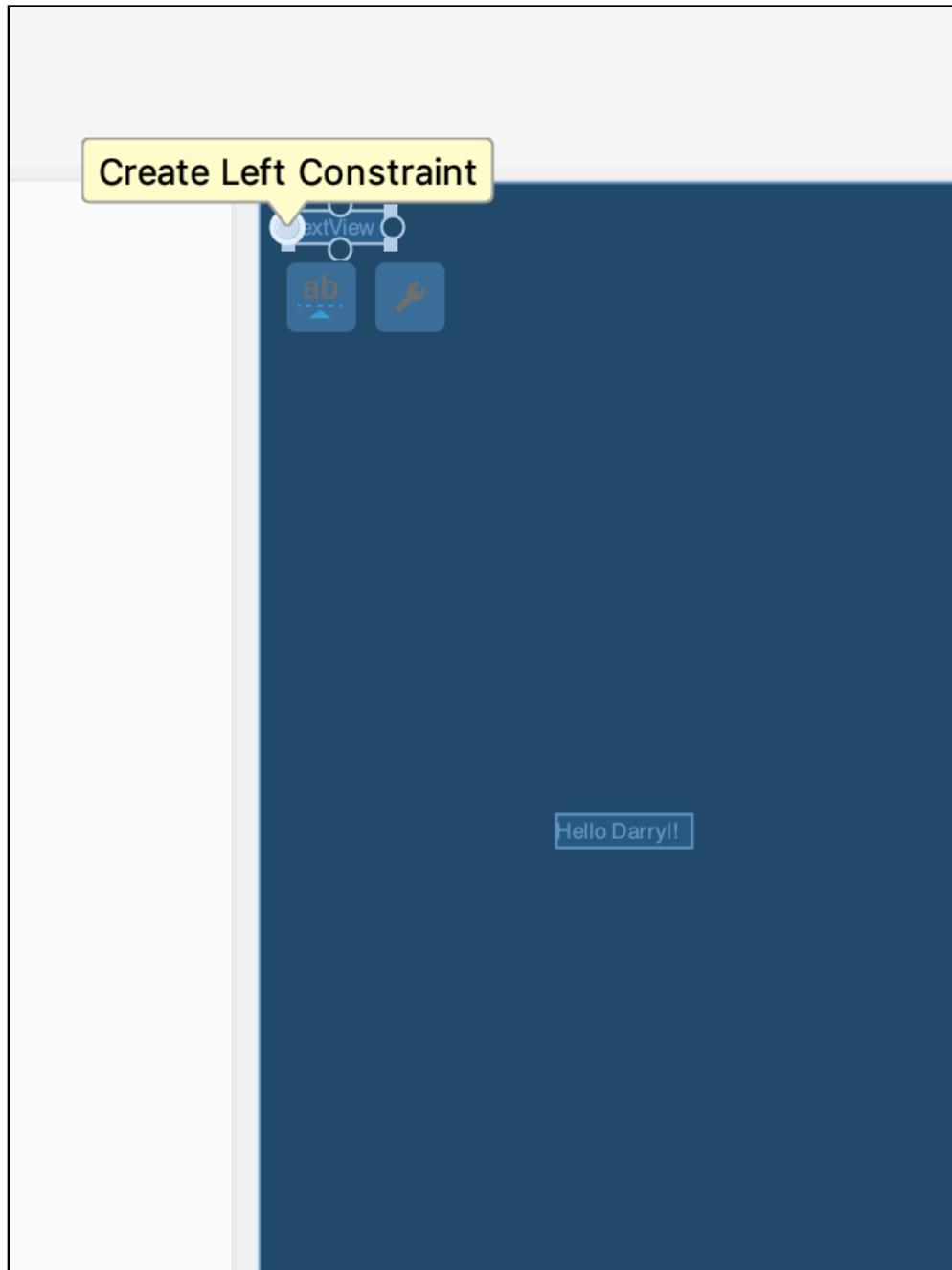
Hey, that's not where you placed the `TextView`! But don't worry; in the next section, you'll ensure the `TextView` stays put.

Adding rules to your position

There are millions of Android devices out there that come in all shapes and sizes. To ensure your app looks great on all of those different screens, you need to do a little Layout work and give the `TextView` some *rules* on where it should show up on the screen.

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

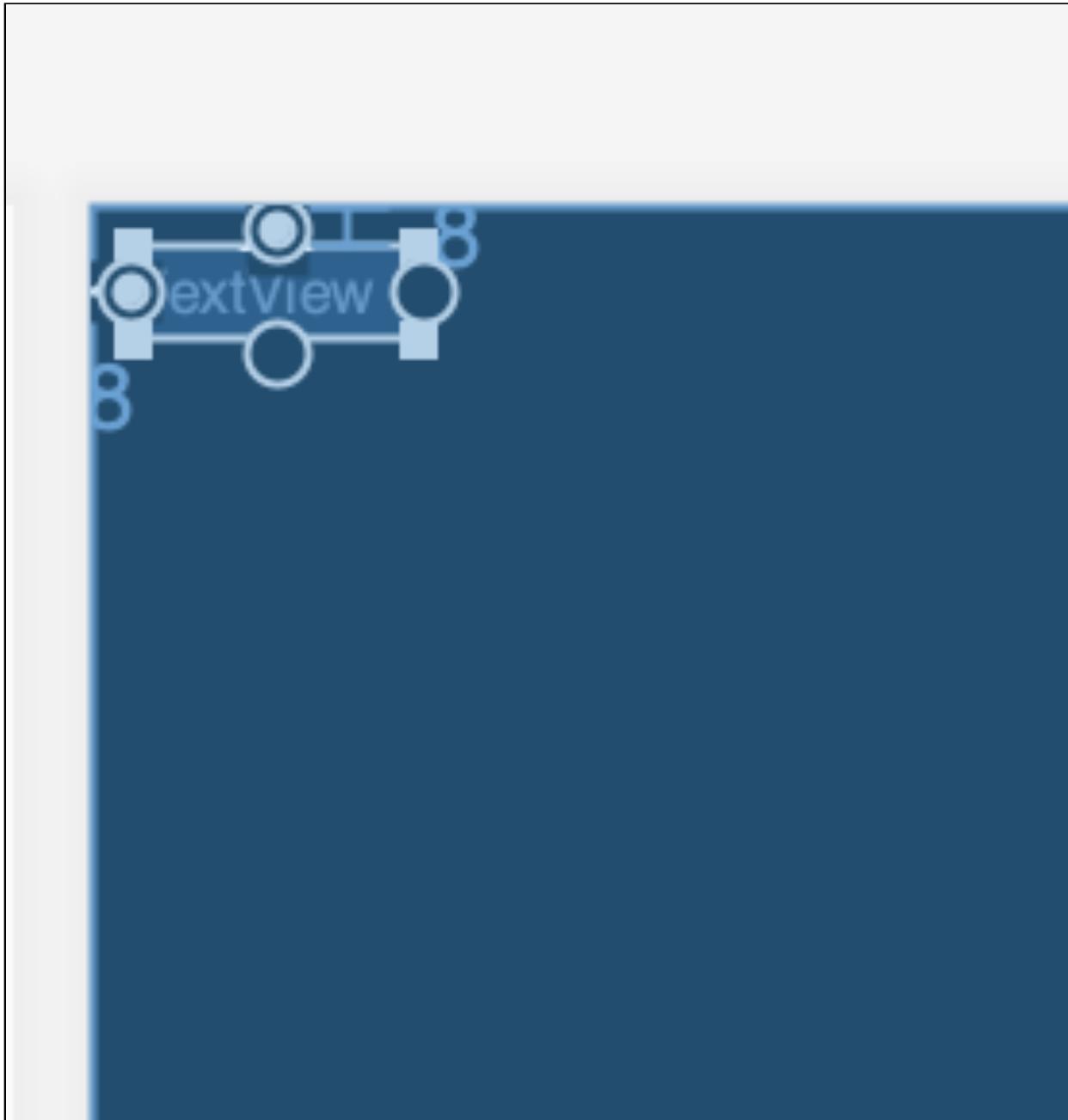
In the Preview screen, click and drag the TextView to the top-left corner of the screen. Now, hover your mouse over the left side of the newly placed TextView in the Blueprint screen. A circle with a white outline appears and a **Create Left Constraint** bubble pops up:



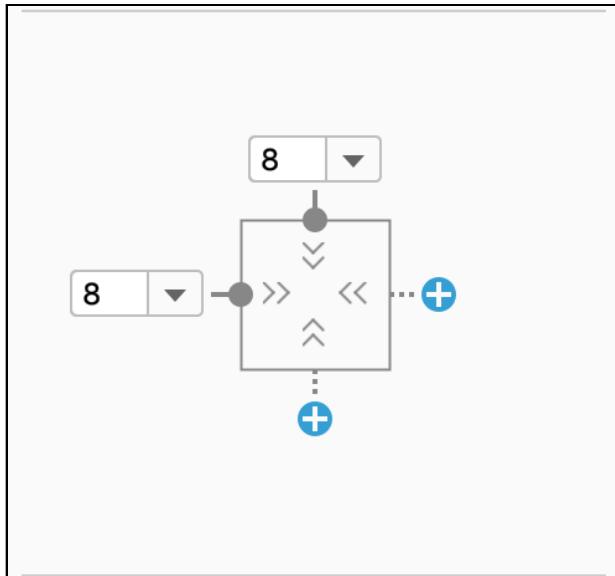
Click and drag toward the left edge of the Blueprint screen, and you'll see the TextView move slightly to the right. At this point, release the mouse button.

Congratulations, you just created your first layout rule!

Next, you 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 the second layout rule.



To see what's happening, select the TextView 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 the screen, and they instruct the TextView how to position itself relative to the screen's edge.

If you want to position this TextView with greater control, you can adjust the margins of the constraint by clicking the number beside the constraint line and selecting one of the preset numbers in the drop-down or entering your own.

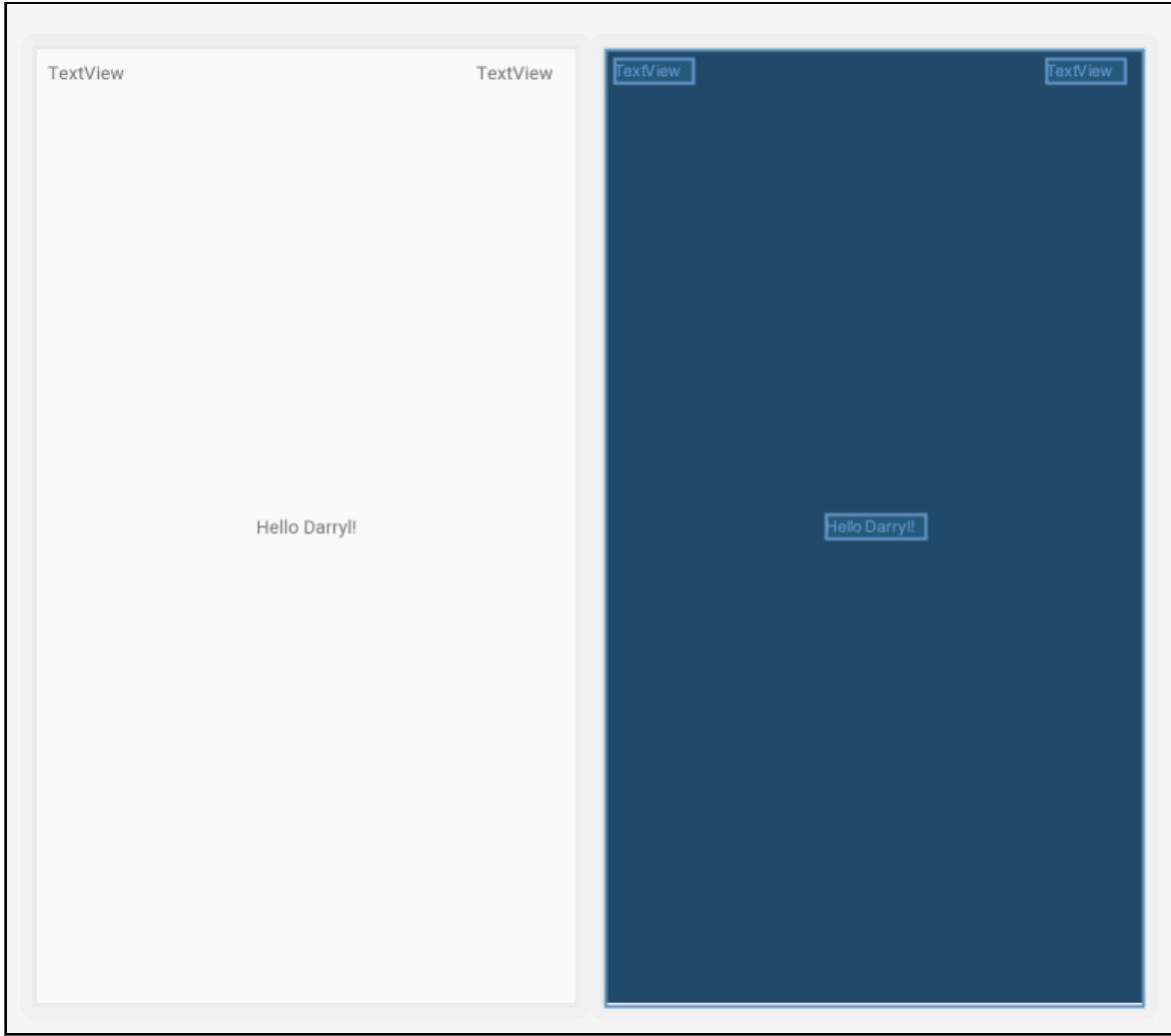
It's time to finish off the screen!

Finishing the screen

Go back 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 the new TextView and hover over the right edge of it until the **Create Right Constraint** bubble appears. Create a new constraint against the right side of the screen. Now, do the same for the top of the TextView against the top of the screen.

You'll end up with something like this:



That takes care of the two TextViews. All that's left is the **Tap Me!** button.

First, remove that TextView floating in the middle of the screen. Select the **Hello User!** **TextView** and press the **delete** key — the TextView disappears.

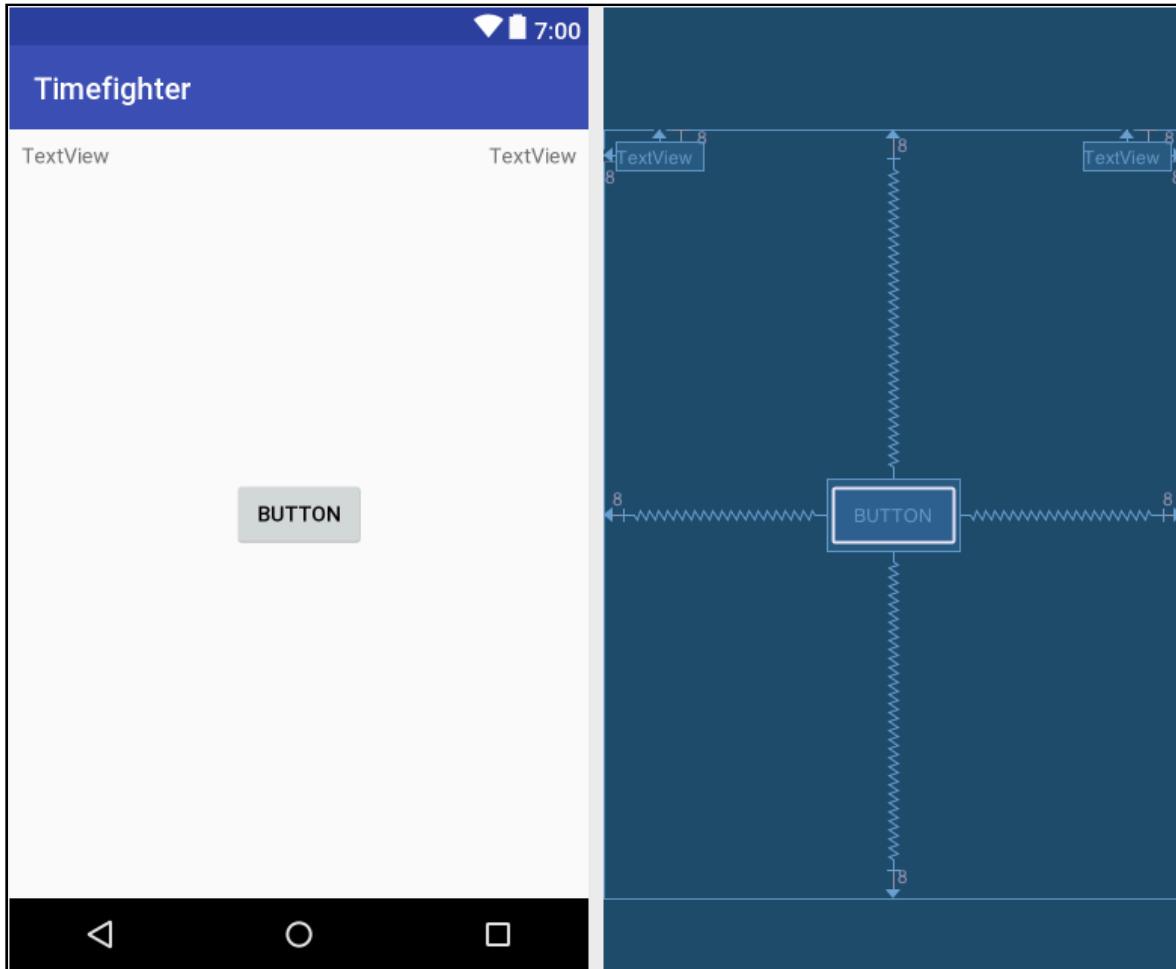
With the TextView removed, you can add the button.

Once again, in the Palette window, click the **Common** tab. When 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 position your Button right in the center of the screen.

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

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

Keep dragging each constraint to the edge of the screen.



Finally, click **Run 'app'** from the top menu in Android Studio. Your emulator or device loads the latest changes to your app, and all of your hard work is rewarded with an app that contains two correctly placed TextViews and one Button.



Where to go from here?

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

Other Layouts 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 of choice. If you want to learn more about it, review the documentation on ConstraintLayout on the Android Developer: <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, review 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!

3 Chapter 3: Activities

By Darryl Bayliss

A lifestyle of various activity — like cardio, strength training and endurance — can keep you healthy. Although they're all different, they each have a specific purpose or goal in mind.

Android apps are similar — they're built around a set of screens. Each screen is known as an **Activity** and is built around a single task. For example, you might have a Settings screen where users can adjust the app's settings, or a Sign-in screen where users can log in with a username and password.

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

Getting started

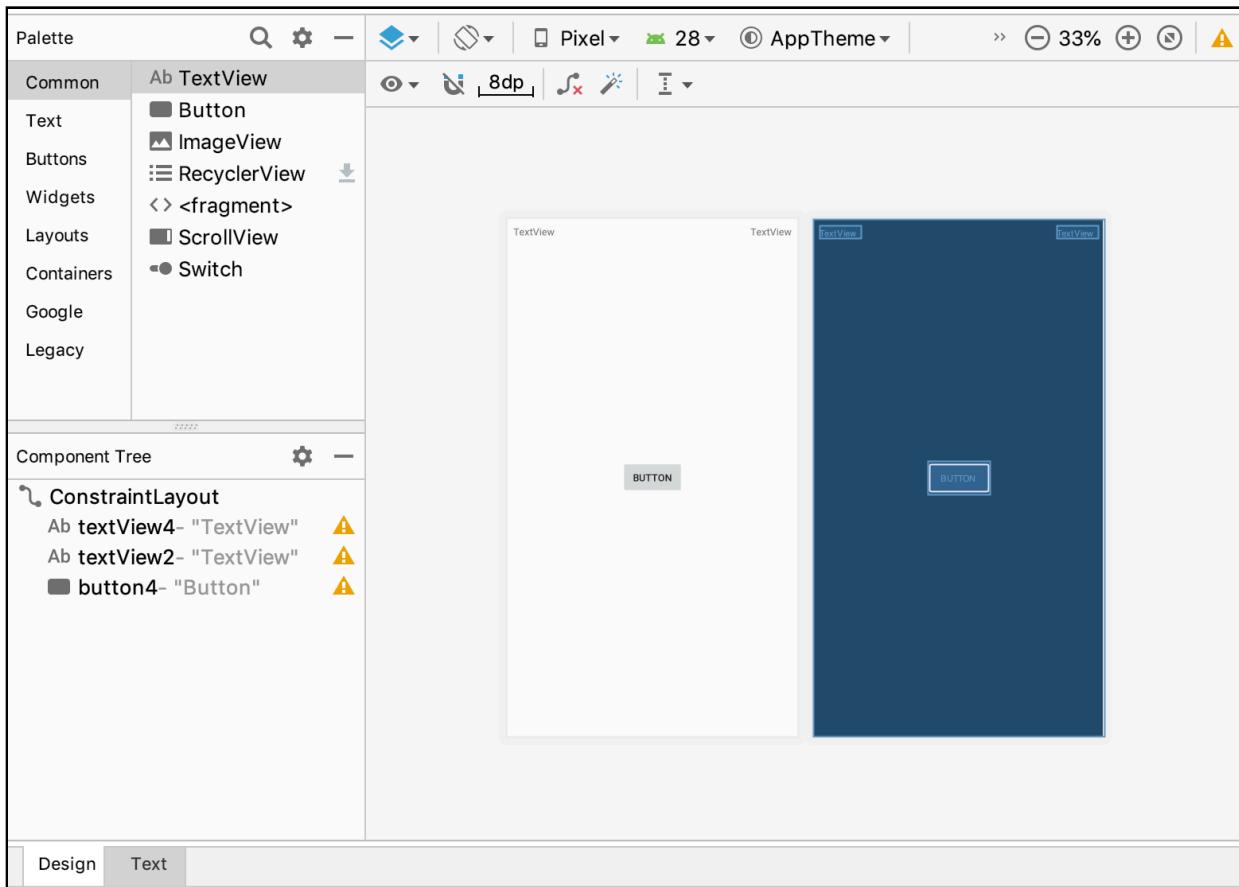
Before you jump head first into writing code, you first need to understand how IDs work. In Android, IDs play a fundamental role in connecting things, for example, connecting Views to your code.

In the previous chapter, you positioned Views and established that the top-left TextView will show the score, the top-right TextView will show the time and the Button, when pressed, will increment the score. Connecting your code to these Views will require each to have its own ID.

If you were following along making your app, open it and keep using it for this chapter. If not, don't worry — locate the **projects** folder for this chapter and open the **TimeFighter** app inside the **starter** folder.

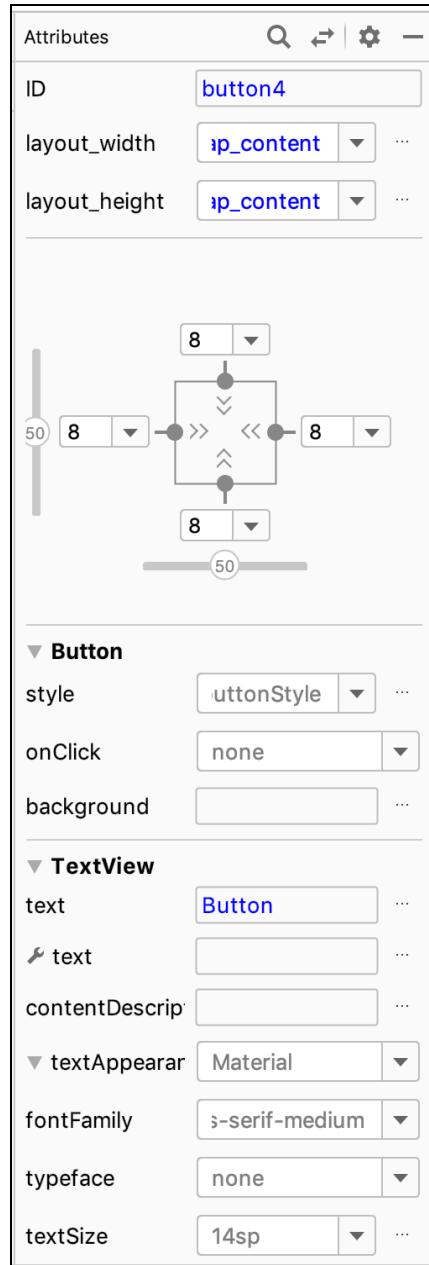
The first time you open the project, Android Studio takes a few minutes to set up your environment and update its dependencies.

Open **activity_main.xml** where you built your Layout and make sure you've selected **Visual editor**. Next to the Palette tab, you'll see a window called the **Component Tree**:



This window provides you with an overview of the Views available in your Layout and their relationship relative to one another.

In the Component Tree, click on the row labeled **button**, or **buttonX**, where X is a number. This action highlights the Button in the middle of the screen and updates the Properties window on the right with details about the Button.



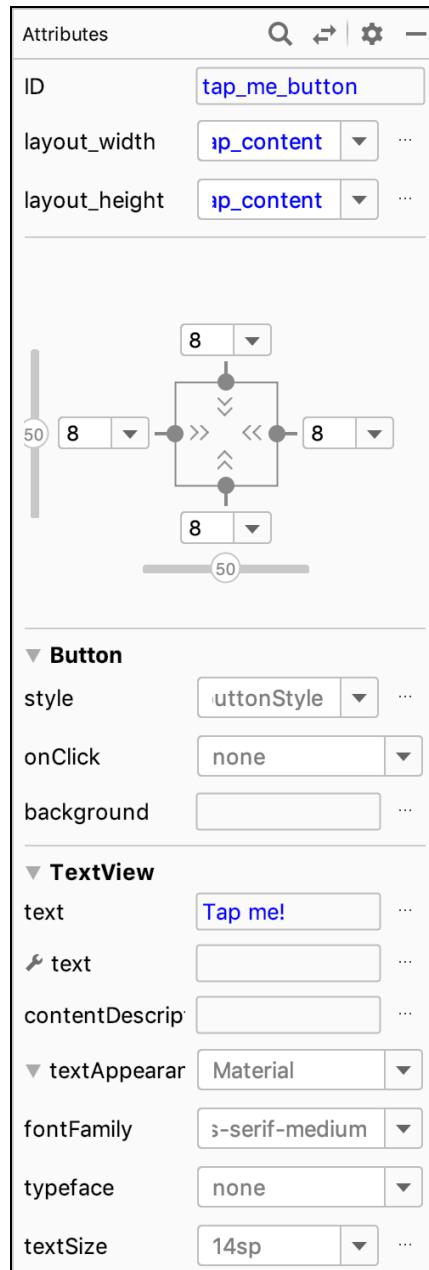
The Button in the screen above already has an ID of **button4**, but this isn't very descriptive.

Note: In your project, it might have a different string value.

Theoretically, you could leave the ID as **button4**, but it's unlikely that in a year that'll mean anything to you. Using descriptive IDs makes it easier to know which IDs refers to which Views.

Change the value of the **ID** field from **button** to **tap_me_button**.

It's also a good idea to give the Button a more descriptive name too. Change the value of **text** in the **TextView** section of the Properties window to **Tap me!**



Select the **TextView** on the top-left from the Component Tree. Set its ID to **game_score_text_view** and change the text to **Your Score: %1\$d**. Finally, select the

TextView you added to the top-right and change its ID to **time_left_text_view** and its text to **Time left: %1\$d**.

So, what's the deal with the “%1\$d”? That's a placeholder for any integer you want to inject into your text values. You'll fill in those placeholders later.

At build time, Android Studio takes these IDs and turns them into constants that your code can access through what's known as the **R** file.

You'll see more about R files in the upcoming sections, but for now, know that Android takes an ID such as **game_score_text_view** that you assigned to your View in your Layout and creates a constant named **R.id.game_score_text_view**, which you can then access in your code.

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



Now that all of 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 that the app folder is expanded. Navigate to **MainActivity.kt**, which is located in **src/main/java/com.raywenderlich.timefighter**. Open the file, and you'll see the following contents:

```
package com.raywenderlich.timefighter

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

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

This is where the logic for your game screen goes. Take a moment to explore what it does:

1. `MainActivity` is declared as extending `AppCompatActivity`. It's your first and only Activity in this app. What `AppCompatActivity` does isn't important right now; all you need to know is that subclassing it is required to deal with content on the screen.
2. `onCreate()` is the entry point to this Activity. 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 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 named **really_good_looking_screen**, then the identifier generated would be `R.layout.really_good_looking_screen`.

These four lines are key ingredients in creating Activities in Android. You'll see them in every Activity you create. In the most general sense, any logic you add must come after calling `setContentView`.

Note: `onCreate()` isn't the only entry point available for Activities, but it is the one you should be most familiar with. `onCreate()` also works in conjunction with other methods you can override that make up an Activity's *lifecycle*.

This book covers a number of those lifecycle methods, but if you're curious, you can find out more at <https://developer.android.com/guide/components/activities/activity-lifecycle.html>.

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

```
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 MainActivity : AppCompatActivity() {
    private lateinit var gameScoreTextView: TextView
    private lateinit var timeLeftTextView: TextView
    private lateinit var tapMeButton: Button

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

        // 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 app.

Note: Sometimes, when using new objects in your classes, Android Studio won't recognize them until you import the class definition. This is shown by Android Studio highlighting the object in red.

To import the class definition:

- macOS: Click the object and press Alt-Return
- Windows: Click the object and press Alt-Enter.

You can also choose to let Android Studio handle imports automatically for you when pasting code.

On macOS, select **Android Studio** ▶ **Preferences** ▶ **Editor** ▶ **General** ▶ **Auto Import** from the top menu. Set **Insert imports on paste** 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 **Insert imports on paste** to **All**. Finally, tick the **Add unambiguous imports on the fly** checkbox.

Hooking up Views

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

In **MainActivity**, you added three variables: `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 immediately after `setContentView`:

```
// 1
gameScoreTextView = findViewById(R.id.game_score_text_view)
timeLeftTextView = findViewById(R.id.time_left_text_view)
```

```
tapMeButton = findViewById(R.id.tap_me_button)  
// 2  
tapMeButton.setOnClickListener { incrementScore() }
```

Going through the code:

1. `findViewById` searches through the `activity_game` Layout to find the View with the corresponding ID and provides a reference to it you can store as a variable.
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's clicked, you increment the score.

You're nearly there now. Add a new variable to the top of `MainActivity` and initialize it to 0:

```
private var score = 0
```

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

```
private fun incrementScore() {  
    score++  
  
    val newScore = "Your Score: $score"  
    gameScoreTextView.text = newScore  
}
```

You increment the new score variable to the next number and use that number in a string to use with your score text view.

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

Ready to see things in action? Run the app and tap the button a few times. The score in the top-left corner of the screen increments with each tap.



You just hit a milestone in your Android app development: You created a View, gave it an ID, accessed it in code and reacted to user input. These are the fundamental tasks of app development, and you'll repeat this cycle many times in your career. Take a moment to appreciate this significant accomplishment.

Managing strings in your app

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

One of the most important elements of any app is the text, or strings, displayed on the screen. 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, and it's a feature you should seriously consider when putting your app on the Google Play store.

In the previous section, you set `gameScoreTextView` to use the string "Your Score: \$score"). This works well if you're only targeting English-speaking users. But how would you support one, two or even a dozen other languages?

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

In the Project navigator, expand **res/values** 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 of the strings used in your app. This helps to keep strings from being sprinkled throughout your code. This also makes it easy to add support for another language. Rather than hunting through the entire project to change all of the strings, you copy the file and change it to hold the language translations of your choice.

You'll use this file to keep your English text in a separate location. Add the following lines under the `app_name` string:

```
<resources>
    <string name="app_name">Timefighter</string>
    <string name="tap_me">Tap me!</string>
    <string name="your_score">Your Score: %1$d</string>
    <string name="time_left">Time left: %1$d</string>
    <string name="game_over_message">Times up! Your score was: %1$d</
    string>
</resources>
```

Now, go back to `incrementScore()` in `MainActivity.kt` and replace the contents of that method with the following:

```
private fun incrementScore() {  
    score++  
  
    val newScore = getString(R.string.your_score, score)  
    gameScoreTextView.text = newScore  
}
```

`getString` is an Activity-provided method that allows you to reference strings from the R file name or ID. In this case, you're retrieving the strings you added earlier to **strings.xml**. You also pass in an `int` for the placeholder `%1$d` you added way back at the beginning of this chapter.

Note: To learn more about String Resources in Android, review 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. Sprinkling strings throughout your app is one of the worst types of technical debt to incur (Technical debt reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution).

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 time fighting going on. In this section, you'll add a countdown timer that limits the amount of time you have to increase your score. `CountDownTimer` is an Android class that starts with a value in milliseconds and counts down until finished.

At the top of `MainActivity`, add the following new properties underneath the View properties:

```
private var gameStarted = false  
  
private lateinit var countDownTimer: CountDownTimer  
private var initialCountDown: Long = 60000
```

```
private var countDownInterval: Long = 1000
private var timeLeft = 60
```

Here, you declare new properties for your game: 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 set the rate at which the countdown decrements and finally a variable to hold how many seconds are left in the countdown.

Finally, replace `resetGame()` with the following method:

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

    val initialScore = getString(R.string.your_score, score)
    gameScoreTextView.text = initialScore

    val initialTimeLeft = getString(R.string.time_left, 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, 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 oddities like symbols appearing next to the time left TextView or the score TextView before you started the game. This method ensures that your game always has a default state to begin.

Take a closer look:

1. You first set the score to 0, then create a variable to store the score as a string, using the `getString` method to insert the score value into your string stored in **strings.xml**. You then initialize `gameScoreTextView` with this value. You repeat the process for the amount of time left and assign it to `timeLeftTextView`.
2. You create a new **CountDownTimer** object and pass it into `initialCountDown` and `countDownInterval`, set to 60000 and 1000. 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 call `onFinish` when `CountDownTimer` has finished counting down. You'll add some code to this later.
4. 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. You can do this in `onCreate()`.

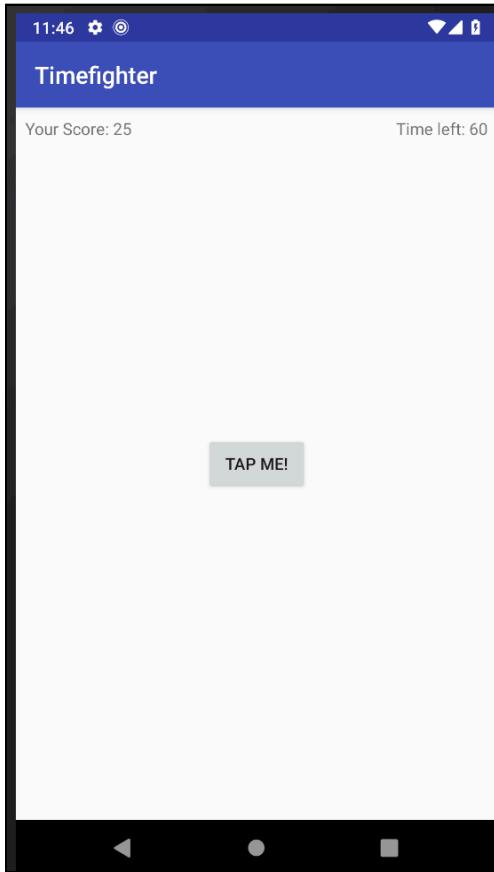
Add the following line to the bottom of `onCreate()`:

```
resetGame()
```

Starting the game

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

Click the Tap me 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  
}
```

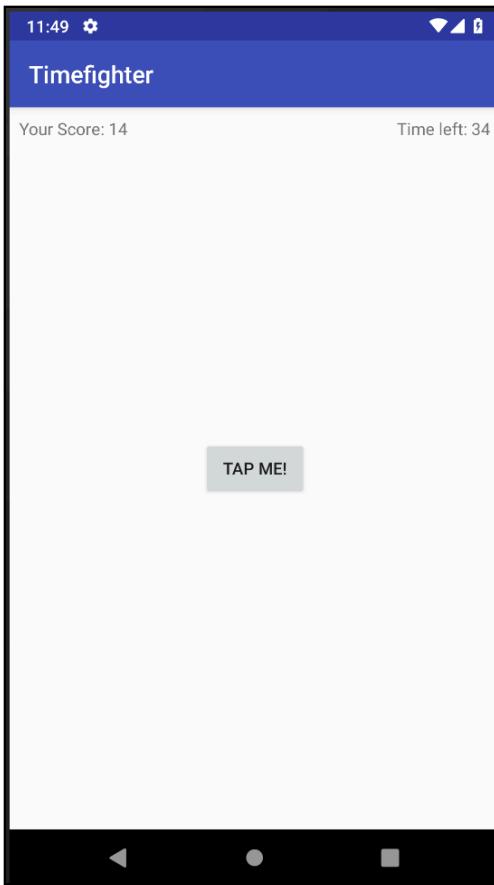
You inform the countdown timer to start. You also set `gameStarted` to `true` to say the game has indeed started.

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

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

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

Run the app to see what's 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. Time to fix that!

In `MainActivity`, replace `endGame()` with the following code:

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

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.

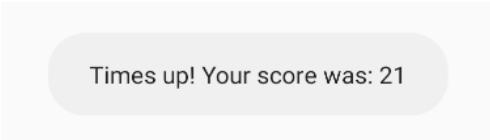
You pass into the Toast the Activity you want the Toast to appear on and the message to display. The end game state is a good time to display the score along with the game over message you put into **strings.xml**.

You also inform the Toast to display for a long time with `Toast.LENGTH_LONG`, which is a few seconds, and then show the Toast. Once that's done, you reset the game. You need to call `endGame()` from somewhere. The best time to call this is when `countDownTimer` finishes counting.

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

```
endGame()
```

Run your app again, and keep clicking the button. The countdown will continue to decrement until it hits 0. Once it does, you'll see the Toast with your score and game over message, at which point the game resets.



Times up! Your score was: 21

Where to go from here?

With a small amount of code, you created a functional game while learning some of the foundational elements of building an Android app. Although this Activity is small, they can get complicated as you add more Views. However, no matter its size, creating an Activity has the same flow:

1. Create a Layout for the Activity.
2. Give the Views in your Layout valid IDs.
3. Create properties in the Activity code and reference those IDs.
4. Manipulate your Views as needed or required.

If you find using `findViewById` cumbersome, you can leverage Kotlin to find your Views for you using the **Kotlin Android Extensions** (KAE) library. This library binds your Views to your code automatically, and provides many more benefits. You can learn how to use KAE over at: <https://www.raywenderlich.com/84-kotlin-android-extensions>.

Next, you'll learn how to fix potential problems in your app using Android debugging techniques.

Chapter 4: Debugging

By Darryl Bayliss

In the previous two chapters, you developed TimeFighter into a full-fledged app. In this chapter, you'll focus on debugging it.

All apps have bugs. Some are subtle, such as glitches within the UI, while others are obvious, such as outright crashes. As a developer, it's your job to keep your app bug-free.

Android Studio provides developers with some tools to help track down and fix bugs. In this chapter, you'll learn how to:

1. Debug your app using Android Studio's debug tools.
2. Add landscape support to TimeFighter.

Getting started

If you've been following along, open your project in Android Studio and keep using it for this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **TimeFighter** app inside the **starter** folder.

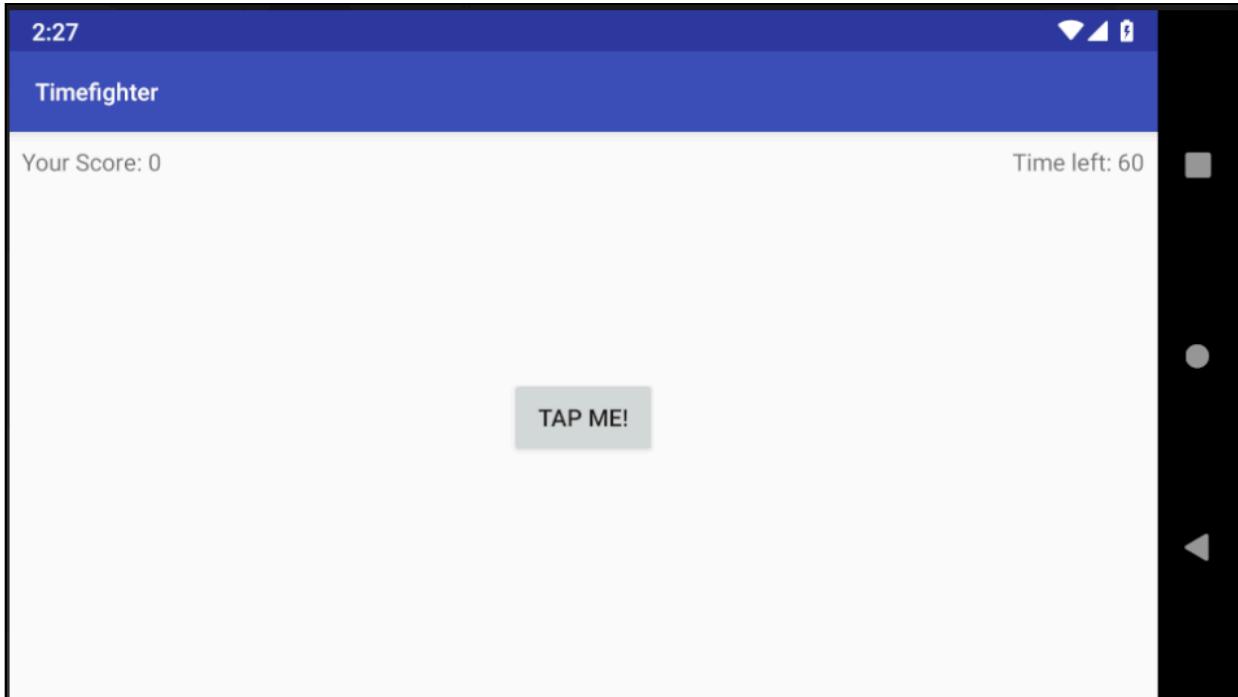
The first time you open the project, Android Studio takes a few minutes to set up your environment and update its dependencies.

You might not have noticed, but TimeFighter has a bug. Start the app in the emulator or on your device. Push **TAP ME** a few times, and then change the orientation of the device to landscape.

Note: For Android Pie devices. You may need to enable auto-rotate on your device

or emulator if the screen doesn't rotate automatically.

To do this, swipe the notification drawer down to reveal the quick settings and ensure the auto-rotate button is colored green to signify it's enabled.



Notice anything strange? TimeFighter resets the game when you rotate the device. Whoops! To understand why this happens, you need to put on your debugging hat and analyze the code.

Add some logging

The first debugging approach is to add logging to your app. With logging, you can find out what's happening at certain points within your code. You can even log and check the values of your variables at runtime.

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

```
// 1  
private val TAG = MainActivity::class.java.simpleName
```

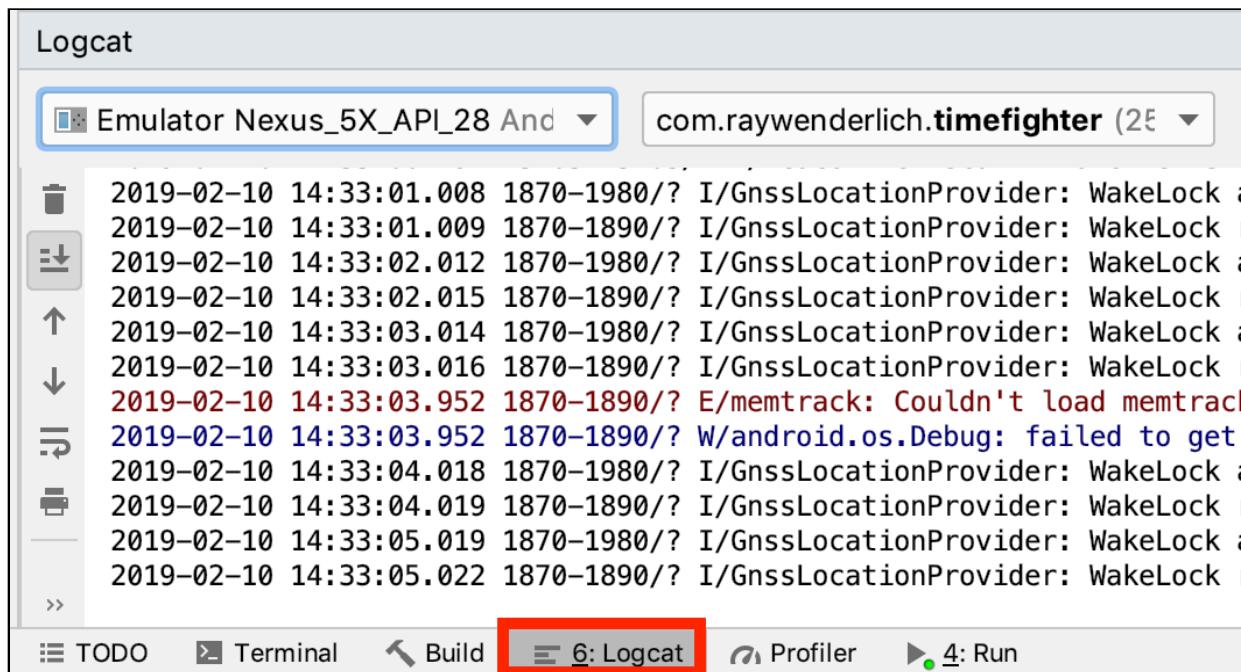
Then, add the following line below the call to `setContentView` in `onCreate()`:

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

Having a look at the code:

1. You assign the name of your class to TAG. The convention is to use the class name in log messages. This makes it easier to see which class the message is coming from.
2. You Log a message when the Activity is created. Your app informs you when `onCreate()` is called and informs you of the current value in `score`. Injecting `$score` into the message is an example of **string interpolation** in Kotlin. At runtime, Kotlin looks for `score` and replaces it in the log message.

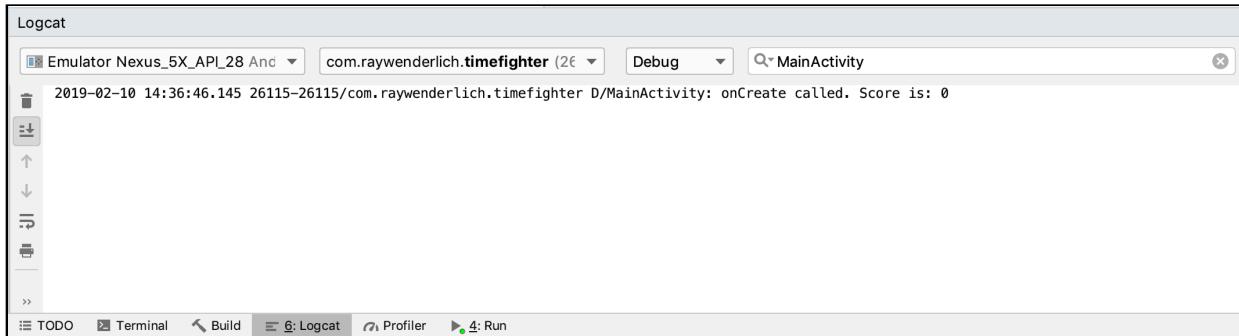
Run the app again. After it's loaded, go to Android Studio. At the bottom of the window there's a button labeled **Logcat**. Click that button, and Android Studio displays a console-like window at the bottom:



With Logcat, you can see everything your emulator or device is doing via log messages, including messages coming from outside of your app. For now, you can ignore most messages and filter down to only the ones you've added yourself.

In the Logcat window, there's a search bar with a magnifying glass. The text you enter here filters the log messages so that you'll only see log messages that match that text.

In the Logcat search bar, type the name of your Activity — **MainActivity** — and watch as the filter gets applied.



Excellent, you can now 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.



That's strange! Why is the score reset to 0? You'll work that out in the next section.

Note: You'll only scratch the surface of Logcat in this chapter. For more information about Logcat and everything it can do, read the Android developer documentation: <https://developer.android.com/studio/command-line/logcat.html>.

Orientation changes

From your log messages, you can establish that score is reset to 0 whenever you rotate the device. But why? The reason for this relates to how Android handles device orientation changes.

When Android detects a change in orientation, it does three things:

1. Attempts to save any properties for the Activity specified by the developer.
2. Destroys the Activity.
3. Recreates the Activity for the new orientation by calling `onCreate()`, which resets any properties specified by the developer.

But it's more than just orientation changes. Android performs these steps any time there's a change to the **configuration** of a device. A configuration change can happen for many reasons, including changes to the orientation or the selected device language. In fact, your Activity can get destroyed and recreated several times while the user is using the app, so it's incredibly important that you develop your app so it can recover from these changes.

Back in **MainActivity.kt**, add the following companion object at the bottom of the properties you declared earlier:

```
// 1
companion object {

    private const val SCORE_KEY = "SCORE_KEY"
    private const val TIME_LEFT_KEY = "TIME_LEFT_KEY"
}
```

Next, add the following methods below `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.")
}
```

Here's what's happening:

1. You create a companion object containing 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`, which is a hashmap 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()`, a method used by the Activity to clean itself up when it is being destroyed. You call `super` so your Activity can perform any essential cleanup, and you add a final log to track when `onDestroy()` is called.

Run your app again. Play the game for a few seconds, change the orientation, and then look at the Logcat output:



```
Logcat
Emulator Nexus_5X_API_28 Anc com.raywenderlich.timefighter (2€) Debug MainActivity
2019-02-10 14:48:08.428 26615-26615/com.raywenderlich.timefighter D/MainActivity: onCreate called. Score is: 0
2019-02-10 14:48:17.031 26615-26615/com.raywenderlich.timefighter D/MainActivity: onSaveInstanceState: Saving Score: 9 & Time Left: 57
2019-02-10 14:48:17.031 26615-26615/com.raywenderlich.timefighter D/MainActivity: onDestroy called.
2019-02-10 14:48:17.096 26615-26615/com.raywenderlich.timefighter D/MainActivity: onCreate called. Score is: 0
```

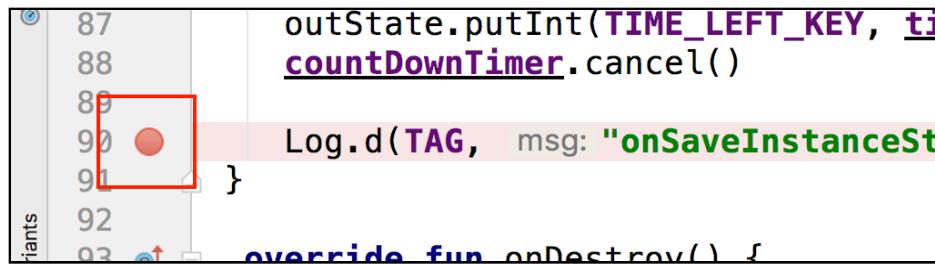
The Activity is still resetting the score back to 0. However, the log statement in `onSaveInstanceState()` is informing you that the score and the amount of time left are saved. How can you verify this? Breakpoints!

Breakpoints

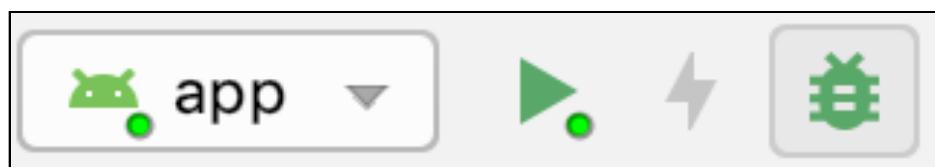
Logging is an effective way of understanding what your app is doing, but it can be tedious to write a log message, recompile, rerun your app and attempt to reproduce the bug. But don't worry, there's another way!

Android Studio provides **breakpoints**. With breakpoints, you can pause the execution of your app to inspect its current state.

In `MainActivity.kt`, scroll to `onSaveInstanceState()` and find the log line at the bottom of the function. Click on the grey border (also known as the gutter) to the left of the line.



This adds a red dot to the gutter to indicate where the breakpoint will trigger. Click the **Debug** button at the top of the window, it looks like a green bug.



The app loads in the same way it did when using the run button, except this time, it attaches the debugger.

Once the app reloads, rotate the screen. Android Studio changes windows and highlights the breakpoint.



```

76     override fun onSaveInstanceState(outState: Bundle) { outState = "Bundle{TIME_LEFT_KEY=60, android:viewHierarchyState=Bundle{android:views=[16908290=android.view.AbsSavedState$1@209b347,
77     super.onSaveInstanceState(outState)
78
79     outState.putInt(SCORE_KEY, score)
80     outState.putInt(TIME_LEFT_KEY, timeLeft) outState = "Bundle{TIME_LEFT_KEY=60, android:viewHierarchyState=Bundle{android:views=[16908290=android.view.AbsSavedState$1@209b347, 2131165190
81     countDownTimer.cancel() countDownTimer: MainActivity$resetGames$1@11126
82
83     Log.d(TAG, "onSaveInstanceState: Saving Score: $score & Time Left: $timeLeft") TAG: "MainActivity" score: 0 timeLeft: 60
84     }
85
86     // 3
87 }

```

MainActivity > onSaveInstanceState()

Debug: app

Debugger Console Variables

this = (MainActivity@11121)

outState = (Bundle@11122) "Bundle{TIME_LEFT_KEY=60, android:viewHierarchyState=Bundle{android:views=[16908290=android.view.AbsSavedState\$1@209b347, 2131165190=android.support.v7.widget...

Your app is paused at the line that has the breakpoint. In this case, it's the log message you added earlier where you save the 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 this information in the **Debug** window below your code. Move to the debugger view and click the arrow next to `this = {MainActivity}`.



this = (MainActivity@11121)
↳ TAG = "MainActivity"
↳ _\$_findViewCache = null
↳ countDownInterval = 1000
↳ countDownTimer = (MainActivity\$resetGame\$1@11126)
↳ gameScoreTextView = (AppCompatTextView@11127) "android.support.v7.widget.AppCompatTextView(20c532a V.ED..... 21,21-239,72 #7f07003d app:id/game_score_text_view)"
↳ gameStarted = false
↳ initialCountDown = 60000
↳ score = 0
↳ tapMeButton = (AppCompatButton@11128) "android.support.v7.widget.AppCompatButton(d12651b VFED..C. 425,729-656,855 #7f070084 app:id/tap_me_button)"
↳ timeLeft = 60
↳ timeLeftTextView = (AppCompatTextView@11129) "android.support.v7.widget.AppCompatTextView(98576b8 V.ED..... 853,21-1059,72 #7f07008a app:id/time_left_text_view)"
↳ mDelegate = (AppCompatDelegateImpl@11130)
↳ mResources = null

The number postfixing your `MainActivity` is likely different since this number indicates where your Activity is allocated in memory.

You might recognize some of the values as your own. However, there are also other values that may be unfamiliar to you. These are values specific to an Activity and give you an appreciation of how much work the `Activity` class does behind the scenes.

Also, when Android Studio hits a breakpoint, it inlines some debugging information within your code, which makes it even easier to inspect things.

Time to put this knowledge to use. Close this in the debugger, expand `outState`, and then expand `mMap`. You'll see some familiar values.

```
▶ └─ this = {MainActivity@11121}
└─ outState = {Bundle@11122} "Bundle[{TIME_LEFT_KEY=60, android:viewHierarchyState=Bundle[{android:views={}]}, mClassLoader={BootClassLoader@11162}, mFlags=0, mmap={ArrayMap@11163} ArrayMap@11163, size=5, mParcelledByNative=false, mParcelledData=null, shadow$klass={Class@3832} "class android.os.Bundle" ... Navigate, shadow$monitor=0}]"
  └─ mmap = {ArrayMap@11163} ArrayMap@11163, size = 5
    └─ value[0] = {Integer@11165} 60
    └─ value[1] = {Bundle@11166} "Bundle[{android:views={16908290=android.view.AbsSavedState$1@209b347, ...}...}"
    └─ value[2] = {Integer@11167} 0
    └─ value[3] = {Integer@11168} 1073741823
    └─ value[4] = {FragmentManagerState@11169}
      └─ mParcelledByNative = false
      └─ mParcelledData = null
      └─ shadow$klass = {Class@3832} "class android.os.Bundle" ... Navigate
      └─ shadow$monitor = 0
```

Compare those numbers with the values of `score` and `timeLeft` — they should match. This informs you that those values are now safely stored in the Bundle. In the next section, you'll see how to restore those numbers when the device orientation changes.

Restarting the game

So far, you've only used `onCreate()` to set up your Activity. You've yet to use the `savedInstanceState` object passed in as a parameter. Are you ready?

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 check to see if `savedInstanceState` contains a value. If it does, you attempt to get the values of `score` and `timeLeft` from the Bundle that you passed in earlier from `onSaveInstanceState`. You then assign those values to the properties and restore the game. If, however, `savedInstanceState` does not contain a value, you reset the game.

Next, implement the following method below `resetGame()`:

```
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

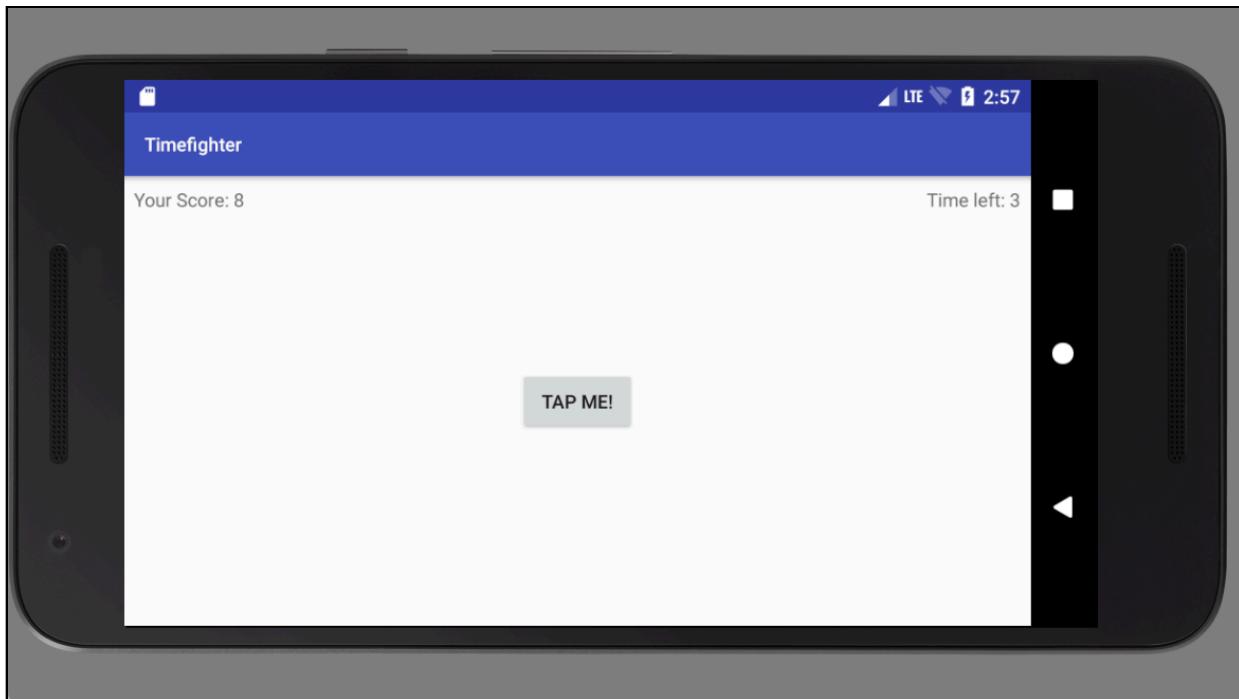
            val timeLeftString = getString(R.string.time_left,
Integer.toString(timeLeft))
            timeLeftTextView.text = timeLeftString
        }

        override fun onFinish() {
            endGame()
        }
    }

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

restoreGame() sets up the TextViews and countDownTimer properties using the values inserted into the Bundle before the change in orientation.

Run the app and play the game for a few seconds. Then, rotate the device to see what happens:



Woohoo! The score and time remaining stayed the same — bug fixed.

Where to go from here?

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

Android Studio contains many debugging tools that are beyond the scope of this chapter. To find out more, read the Android developer documentation: <https://developer.android.com/studio/debug/index.html>.

Note: 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. If you find yourself in this situation, inform the developers who maintain that code via their bug reporting channels.

For now, you're armed with enough tools and techniques to debug potential problems in your own apps. In the next chapter, you'll finish up TimeFighter so that it looks and feels more in place in the Android ecosystem.

Chapter 5: Prettifying the App

By Darryl Bayliss

Take a moment to congratulate yourself and recognize what you've accomplished so far: You have a working Android app that lets users fight the clock and score as many points as possible.

You also fixed a few undiscovered bugs and added support for portrait and landscape mode, regardless of their device. By all accounts, your app is ready to entertain people for years to come!



There's one problem though: It's not *visually exciting*.



Nothing special here...

An app that looks visually appealing tends to stick out when compared to similar apps. While it's not integral to the functionality of your app, it *does* give it that “wow!” factor.

In this final chapter for the section, you'll learn how to:

1. Adjust your app to adhere to the Material Design Guidelines.
2. Add small touches to give your app a polished look and feel.
3. Add a simple animation to your app to give it some life.

Getting started

If you've been following along, open your project and keep using it for this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **TimeFighter** app inside the **starter** folder.

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

With TimeFighter open, run the app and consider some things you can do to improve the way it looks and feels. Perhaps you can change the color of the app bar or the white screen? Maybe the button feels a little lifeless when tapped? And why is the app so silent — maybe it needs sound effects?

The important thing to remember is that you don't need to do *everything*. You only need to make changes that add to the essential elements on the screen. If you add too much, you run the risk of cluttering the screen and confusing the user.

Changing the app bar color

In the Project navigator, on the left side of Android Studio, open **colors.xml**; it's located 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>
```

colors.xml stores the color values used in your app. Like **strings.xml**, it's an excellent place to store the color-related values and keep them in one location, which makes it easier to change things later.

To define colors, you use a `<color>` tag along with a name attribute that you can use as a reference when it's compiled into **R.java**. The reference is available for use in your XML Layouts and also at runtime in your code.

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

With the theory out of the way, you're ready to update the file. In **colors.xml**, change the values to match the following:

```
<?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>
```

Are you wondering how this changes the color of the app bar at the top? The answer lies in **styles.xml**. This file is located in **app > res > values**.

Open **styles.xml** and review its contents:

```
<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>
```

Notice the **<item>** tags. These tags define specific items within your app which adhere to a particular color. In this case, these colors are the colors you updated in **colors.xml**.

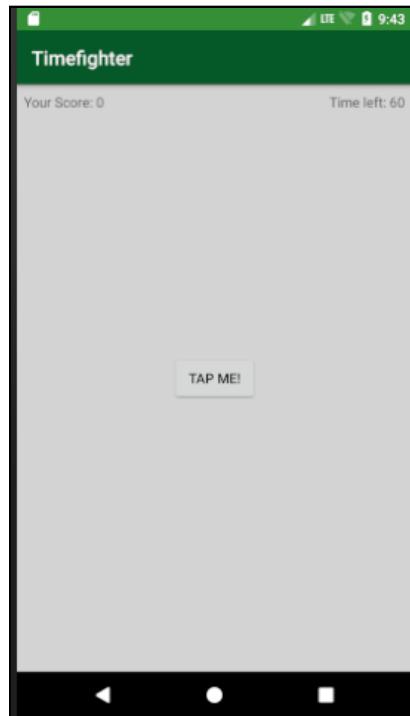
Note: Your app is adhering to a **Style** set within this file. This is used to set the presentation of Views and screens. You can override items that are inherited from other themes provided by Android or other developers. For more information, visit: <https://developer.android.com/guide/topics/ui/themes.html>.

One final tweak! Open **activity_main.xml**, located in **app > res > layout**. Switch from **Design** to **Text**, and update **ConstraintLayout** to change the color of the background, like so:

```
<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.MainActivity">
```

You now reference the **colorBackground** color added in **colors.xml**.

With that done, run the app and see if you can still recognize it.



That's more interesting!

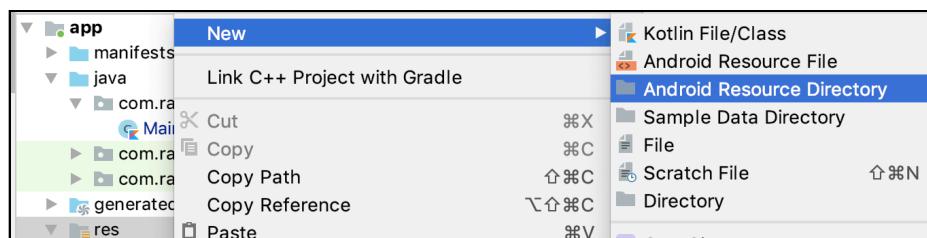
With a few lines of code, you've managed to transform the app and make it more visually appealing.

Animations

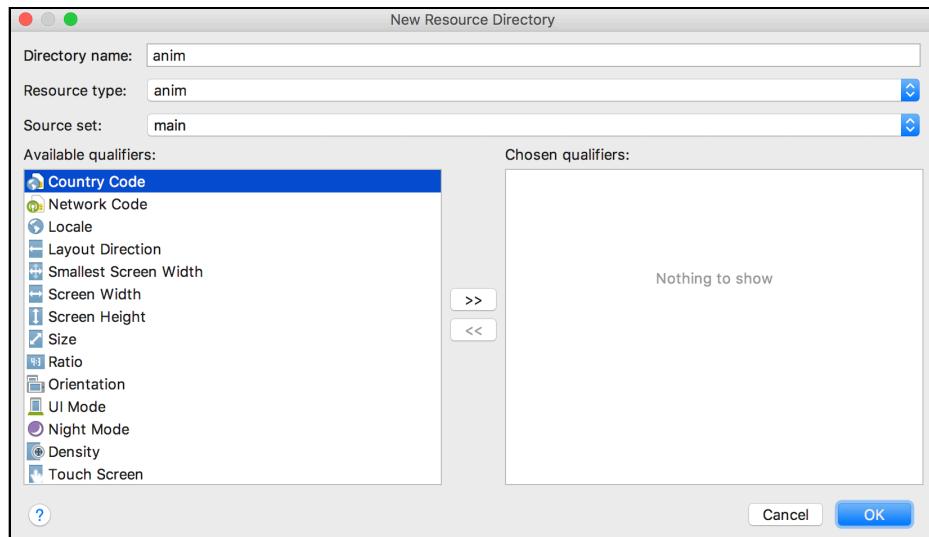
Animations give visual emphasis to elements and help direct the users' attention. When it comes to animation, the most important rule is to use it where and when it matters — not simply because you *can*.

One of the most heavily-used components in TimeFighter is the “Hit Me” button — because that’s what earns the user points. So, adding an animation here makes sense!

In the Project navigator, right-click on **res**. In the drop-down window, navigate to **New** and click **Android resource directory**.



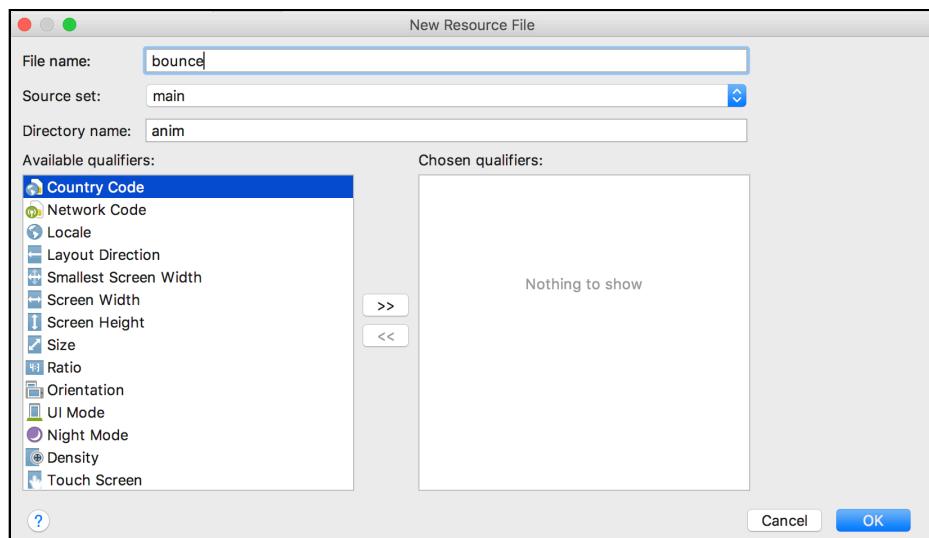
In the **New Resource Directory** window, click the drop-down button next to **Resource type** and change it to **anim** — which automatically changes the name of the directory — and click **OK**.



In the Project navigator, you now have a new folder inside **res** named **anim**.

Next, you need to create the file defining the animation for your button. Right-click on **anim**, navigate to **New**, and click **Animation resource file** on the right-most dropdown.

You're presented with a window similar to the one you saw when creating the **anim** folder. This time though, you need to enter the name of the file. For the **File name**, enter **bounce**, then click **OK**.

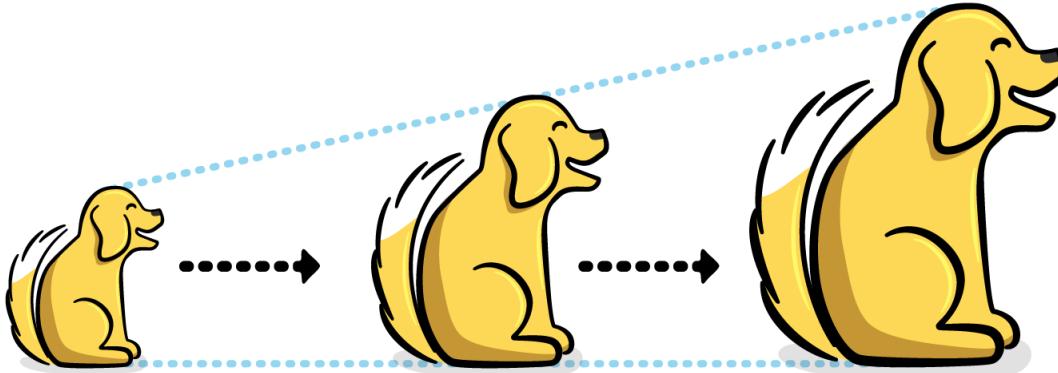


Android Studio creates the file and automatically opens it for you:

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

Notice the **set** attribute. This is a container that holds all of the transformations that occur throughout your animation. You can bundle more than one transformation in the same animation and have them all run concurrently.

Think of a transformation as something that happens over time. Imagine a dog moving from the left of the screen to the right: As the dog walks along the screen, his position changes; he may even grow larger as he moves.



This dog is performing two transformations. Moving from left to right and also growing in size!

For this animation, however, you only need one transformation.

Edit **bounce.xml** to match the following:

```
<?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's happening:

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

The `set` is declared and `fillAfter` is set to `true`, which means the animation won't reset the View to its original position once it's complete. Instead, the View remains wherever it is when the animation ends.

This `set` is also told to use the `bounce_interpolator` from Android. Interpolators affect the rate the entire animation is performed over time, independent of durations set within the transformations.

Android provides many built-in interpolators. It also lets you create your own if you don't find one that suits your needs. For now, the `bounce_interpolator` included with Android works nicely.

Time for the next few lines:

```
<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" />
```

You declare a `scale` attribute. This informs the animation to resize the View. You also declare that scaling occurs over 2000 milliseconds (2 seconds), via the `duration` attribute.

In addition, you set the width and height of the View as `2.0`, twice the original size when the animation starts via the `fromXScale` and `fromYScale` attributes.

The `pivotX` and `pivotY` attributes specify the center point where the animation occurs. In this case, it occurs from the center of the View, expressed in percentages as `50%`: halfway across the X-axis and halfway across the Y-axis.

Finally, you set the size of the View at the end of the animation as `1.0`, via the `toXScale` and `toYScale` attributes. This sets the View back to its original size.

In summary, the animation you 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, review the Android Developer documentation (<https://developer.android.com/guide/topics/resources/animation-resource.html>) for an in-depth review.

That's it for the bounce animation!

Open **MainActivity.kt** and modify the `tapMeButton.setOnClickListener` callback in `onCreate()` to use this animation:

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

Every time you click the button, `tapMeButton.setOnClickListener` loads the bounce animation inside **anim** and instructs the button to use that animation. Run the app and click the button.

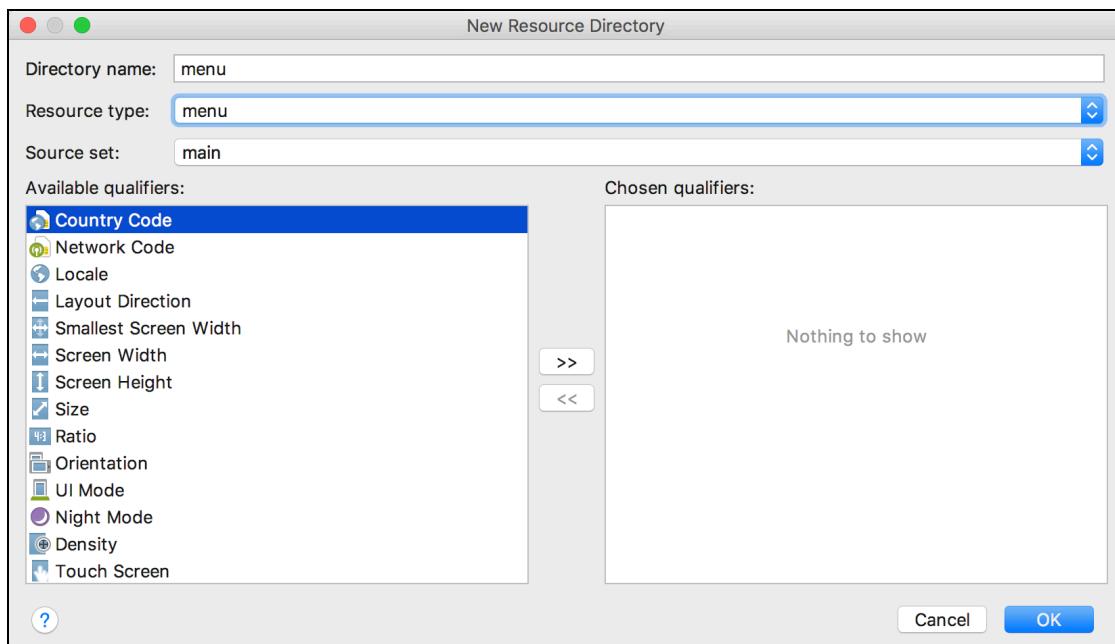
Adding a Dialog

Making an app if fun, and at some point you need to let the users know who created it. But at the same time, you don't want to distract your users while they're playing your game. So what can you do? One option is to use a **Dialog**.

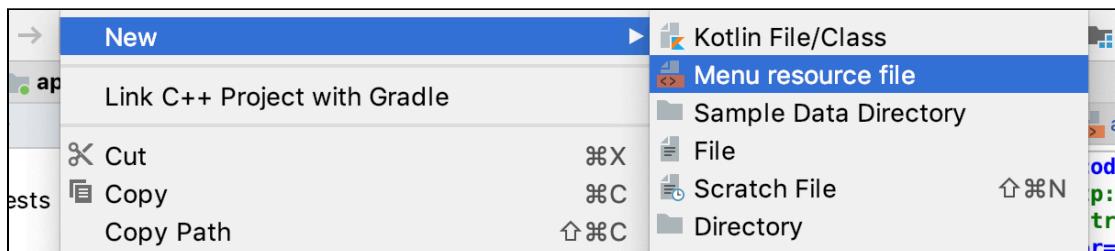
A Dialog is an excellent way to provide a snippet of information without moving away from the main content on the screen. Dialogs come in all shapes and sizes, but in this case, you want to let your users know about the creator of the app and what version of TimeFighter they're running.

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

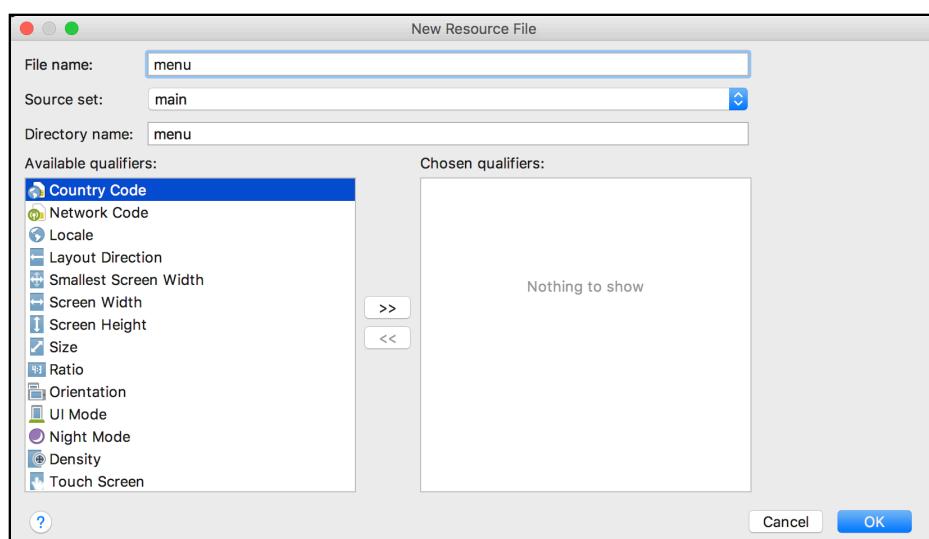
In the Project navigator, locate **res**. Right-click on the folder and select **Android resource directory**. In the new window, click the resource type drop-down and change it to **menu**. Then, click **OK**.



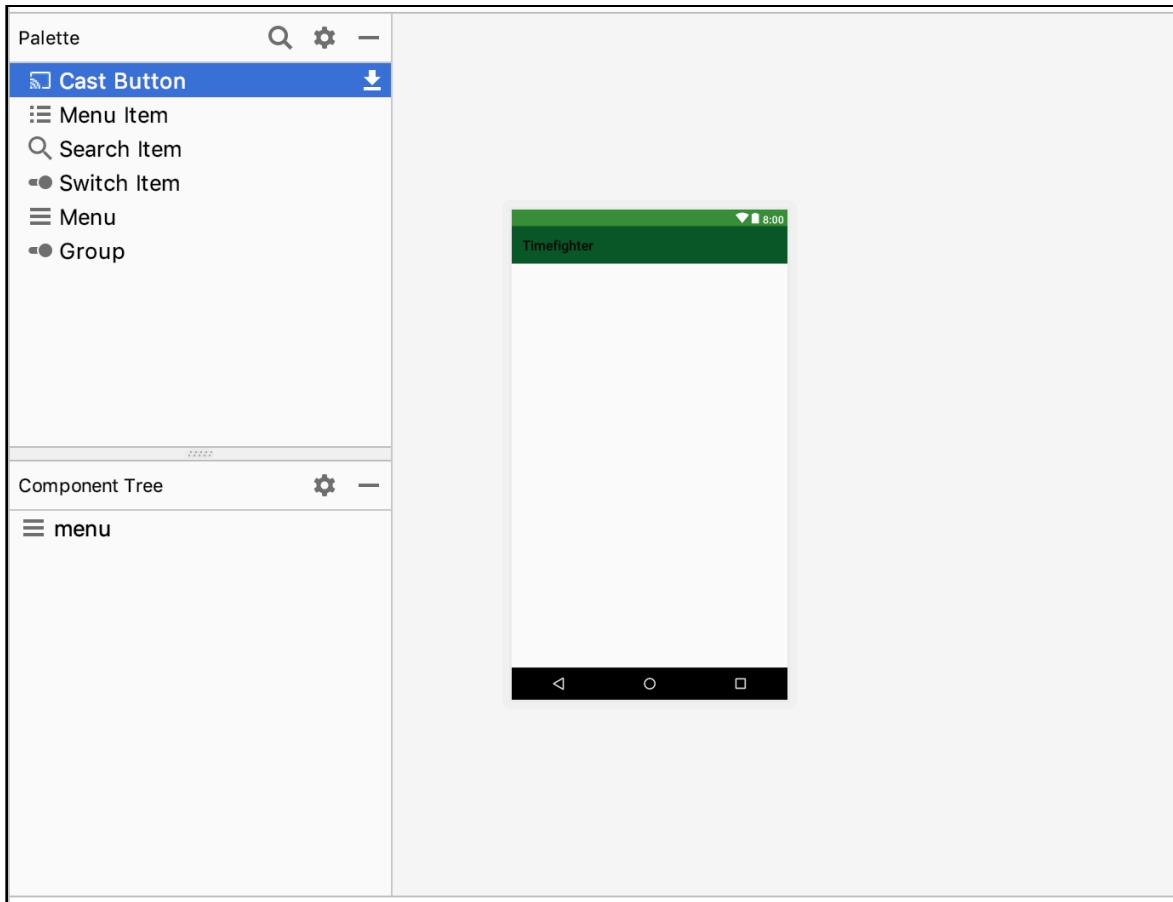
Right-click on the newly created **menu** resource folder. In the pop-up menu, hover over **New**, and then click on **Menu resource file**.



In the **New Resource File** window, enter the file name as **menu** and click **OK**:



Android Studio changes over to the Layout window and shows you a similar setup to what you've seen when editing Layout files:

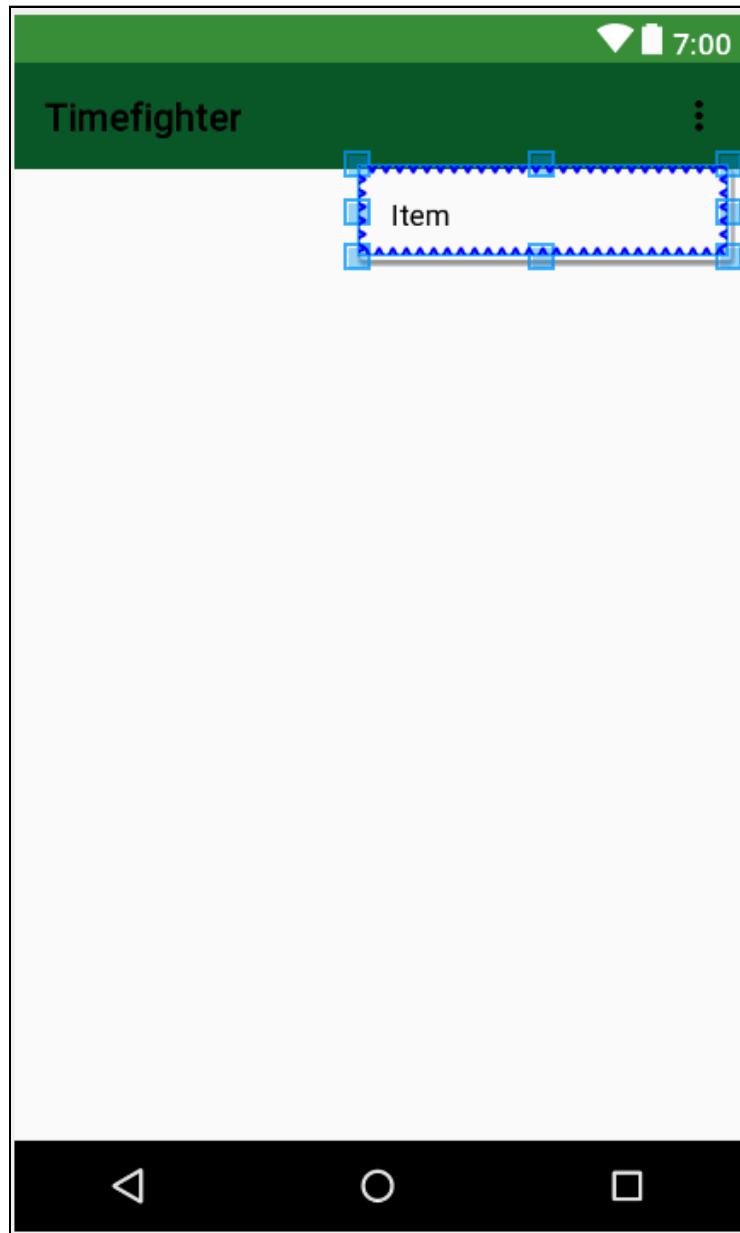


Note: Android Studio may open the Text editor. If this happens, click **Design** at the bottom of the Layout window.

Here, you have the usual windows. The **Palette** in the top-left changes to show only menu-specific items, and the **Component Tree** gives you an overview of the hierarchy for your menu.

You want a single 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'll end up with something like this:

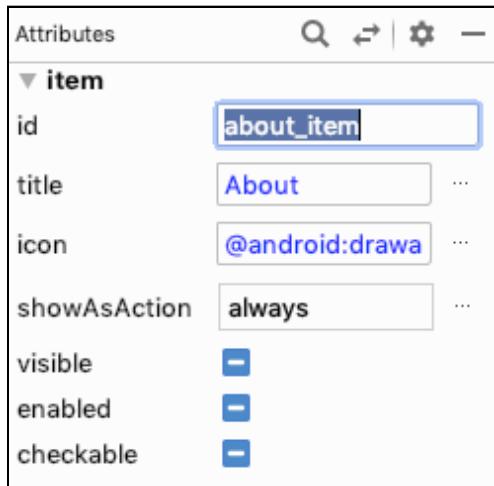


So far, so good. Your newly-placed menu item is now highlighted, and the **Attributes** window is shown on the right side. Time to edit some of these attributes!

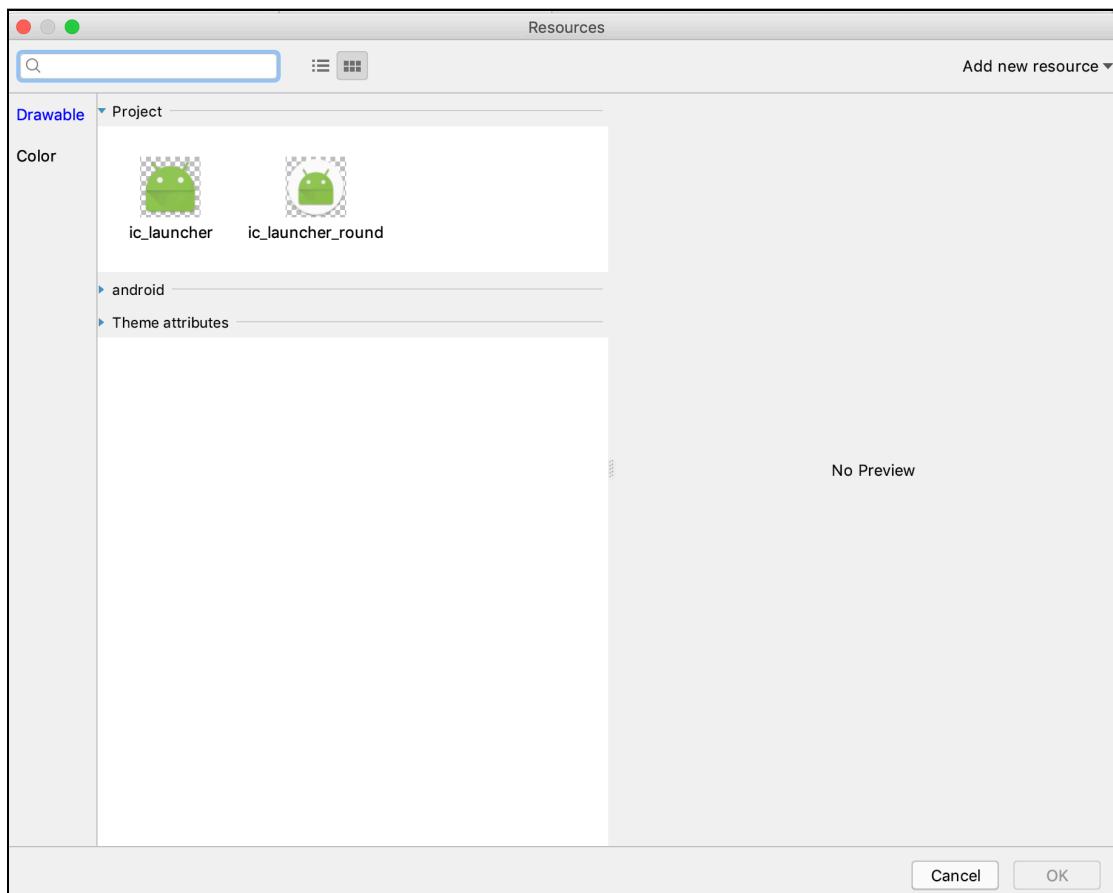
First, set the **id** for the menu item and name it **about_item**. Next, move on to the **title** attribute and name your menu item **About**.

You now need to decide what icon to use for the menu item. Android includes plenty of embedded images from which to choose, so you can use one of those.

Click the small dots next to the **icon** text field. They appear like so:



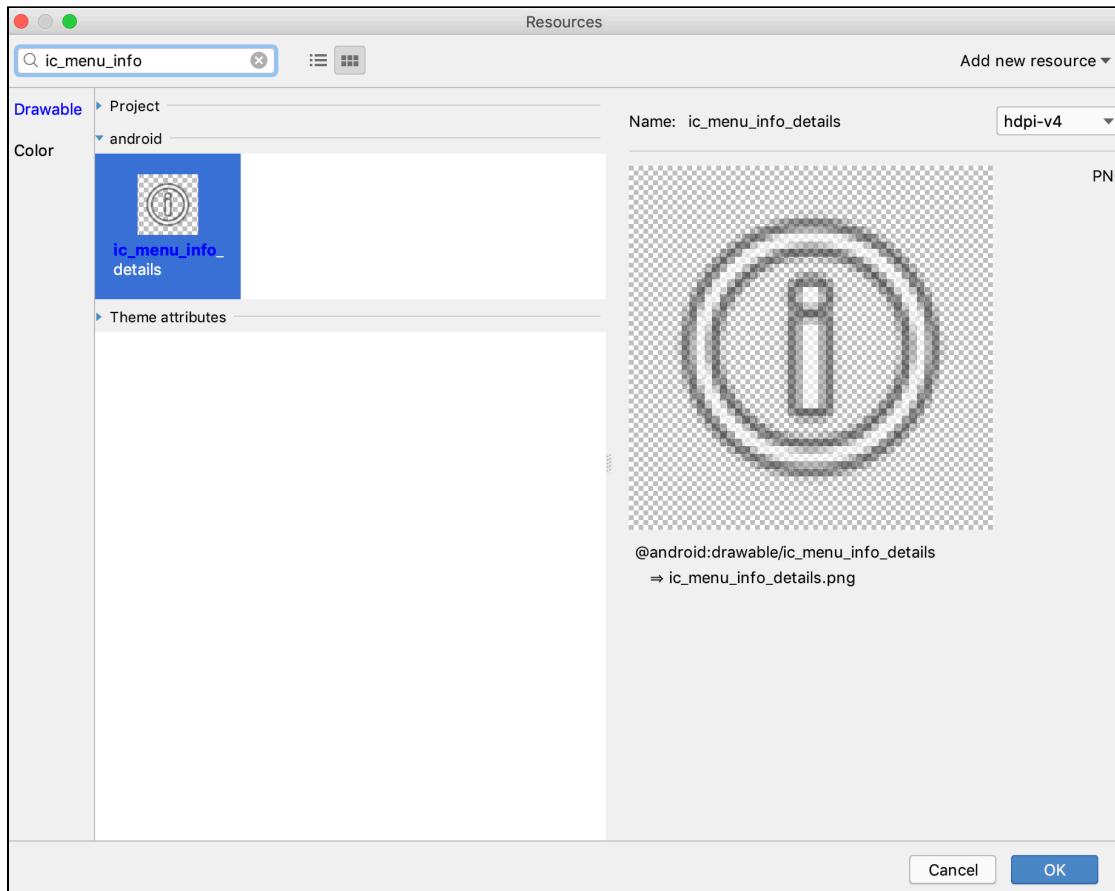
You're presented with the resources window.



This window displays the resources available for 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_info**.

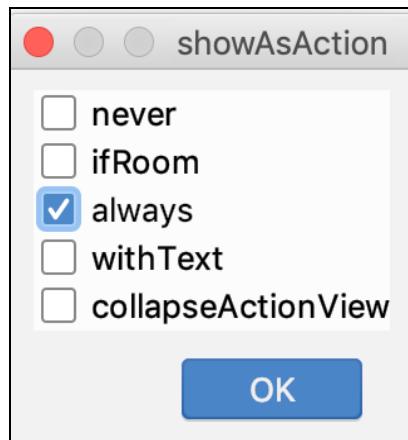
As you type, the list of resources filters down to match any resources that contain the characters you entered. In this case, there's only one.



Click the resource under the Android drop-down to select it, and then click **OK**. The Resource window closes and takes you back to the Layout window. The icon text field is populated with the resource you chose.



Finally, to make sure the button is always visible, you need to set the **showAsAction** attribute. Click the small dots next to **showAsAction**. In the dialog that appears, check **Always**, and then click **OK**.



showAsAction affects how your menu item is presented and can have multiple choices depending on the number of items your menu contains and the screen size of your device. You want the menu item to always show up regardless of the circumstances. To do that, you need to set the value to **Always**.

Looking good! Now for some Kotlin code.

In **MainActivity.kt**, add the following method below **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
}
```

This overrides the Activity callback when it attempts to create the menu. You make a call to **super** to give any superclasses of your Activity a chance to set themselves up. You then use the Activity's **menuInflater** to programmatically set up your menu layout for the Activity. Finally, you return **true** to let the Activity know that the menu is set up.

Below **onCreateOptionsMenu(menu: Menu)**, add this method:

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

This method is called whenever a user selects a menu item. You check to see if the ID of the selected menu item is equal to the ID of the item you set up earlier; if so, you call `showInfo()`. Once again, you return `true` to let the Activity know that the event was processed.

Nearly there, just a few more lines!

You still need to create `showInfo()`. Add the following method to **MainActivity.kt**, anywhere inside of the class. Don't worry about editor errors; you'll work on that next.

```
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()
}
```

`showInfo()` handles the setting up of a dialog View for you. It creates two strings to use in the dialog, one for the title and one for the message.

These strings are created using a mixture of the strings stored in **strings.xml** and 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 a moment.

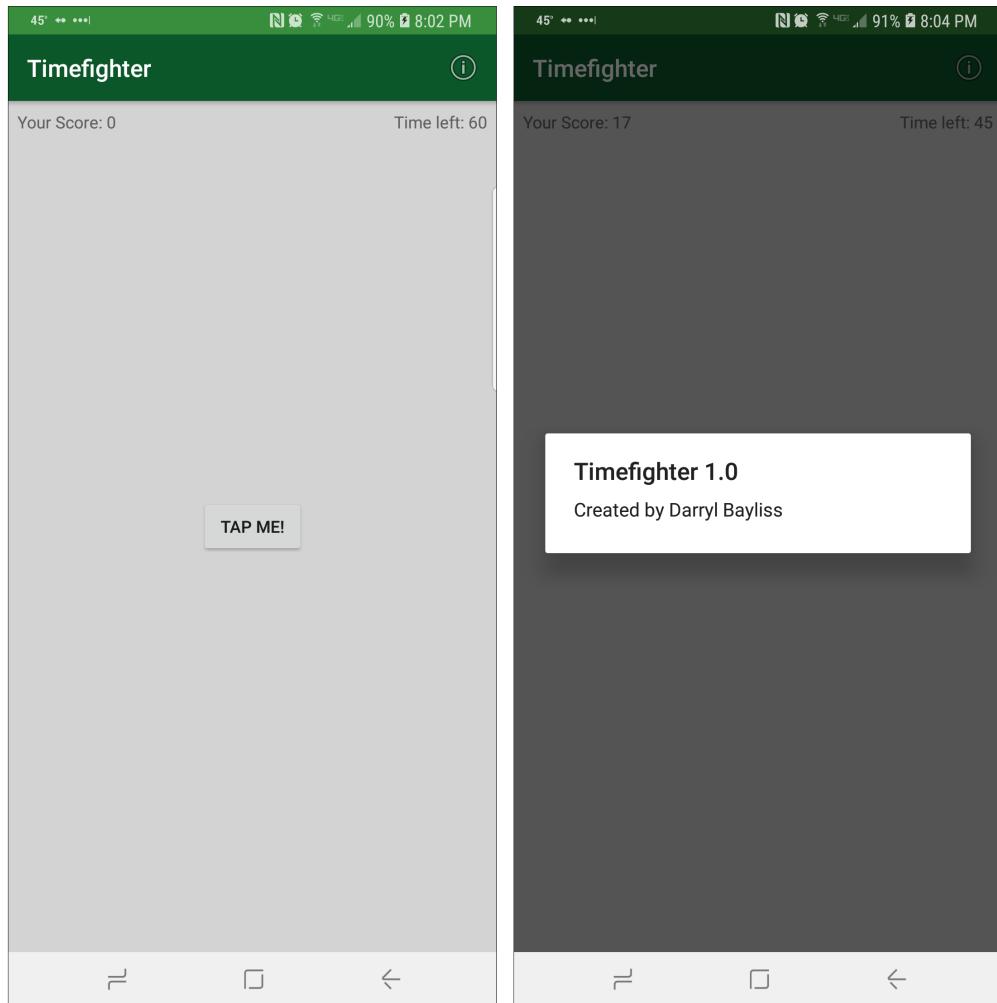
Next, you create an `AlertDialog.Builder` and pass in a Context instance to let the Dialog know where to appear. You pass in the title and message, create the Dialog, and finally display it.

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

Open **strings.xml** and add the following strings, substituting your name in `about_message`:

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

Finally, run the app and check out the new menu sitting in the top-right of the screen. Tap the info button in the menu, and the dialog shows up in the middle of the screen.



Fantastic! You now have a place for people to find out what version of your app they're using and who created it. Well done.

Where to go from here?

Congratulations on completing the first section of the book. You learned a lot over the last few chapters, and you now know how to create a simple game.

In the next section, you'll stop working on TimeFighter and move on to a different app that builds upon the skills and concepts you've learned in this first section.

Section II: Building a List 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.

You'll also learn how to persist data to your app using **SharedPreferences** and create different screens dedicated to different tasks. Towards the end of the section, you'll learn how to change your App to adapt to different screen sizes using **Fragments**.

Finally, you'll give ListMaker a design overhaul by enhancing it to follow **Material Design**. The recommended design language for Android apps.

[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

It's time to say goodbye to TimeFighter and move on to your next app. This new app, **ListMaker**, will allow you to create handy lists that you can look at later.

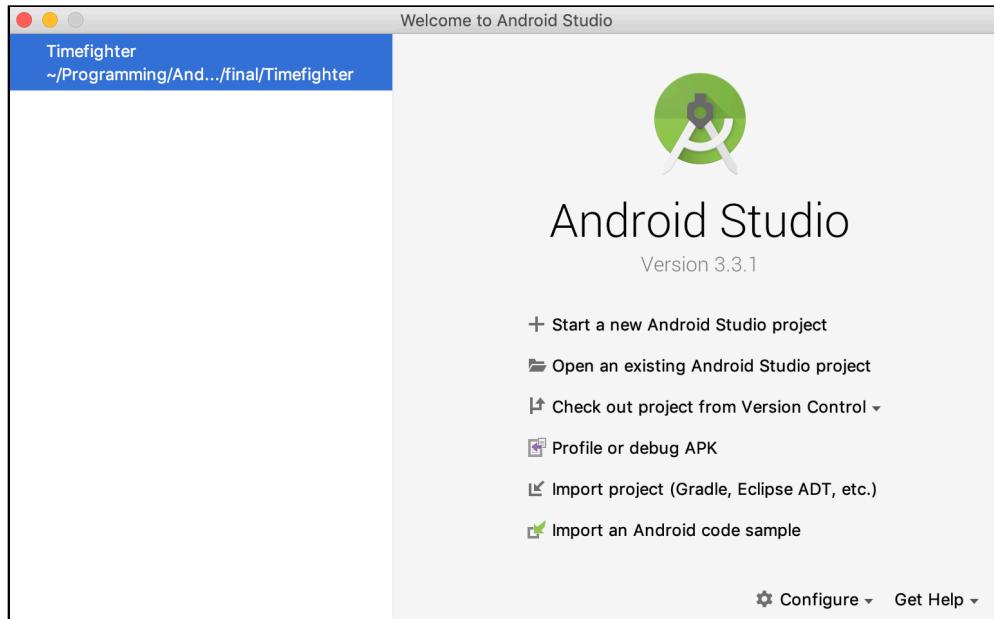
In this chapter, you'll:

1. Give your project an appropriate name and initial package structure.
2. Learn about each step of the project set up process and the associated screens.
3. Set up your new project.



Getting started

Open Android Studio, and you'll see a screen like this:

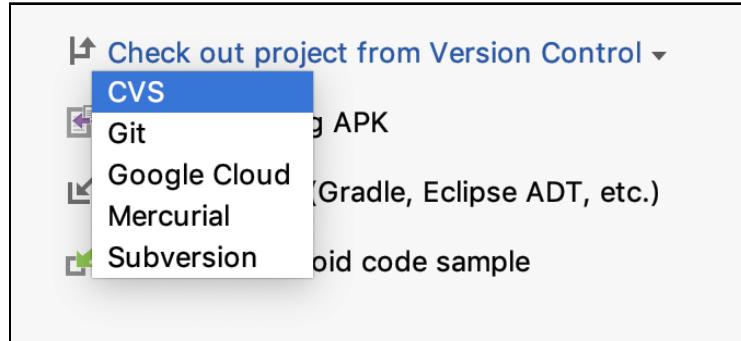


Before you begin working through the steps of creating a new app, there are some useful features in Android Studio that are worth pointing out for each of the options in this screen.

- **Start a new Android Studio project:** Starts creating a new project for you to build your app. You'll use this later.
- **Open an existing Android Studio project:** Lets you navigate through your computer's folders to find and open an existing Android Studio project.
- **Check out the project from Version Control:** Opens an Android Studio repository that's pulled from the internet and onto your computer.

Because Android Studio is built on IntelliJ, an IDE from the company JetBrains, you get access to powerful version control tools directly inside Android Studio.

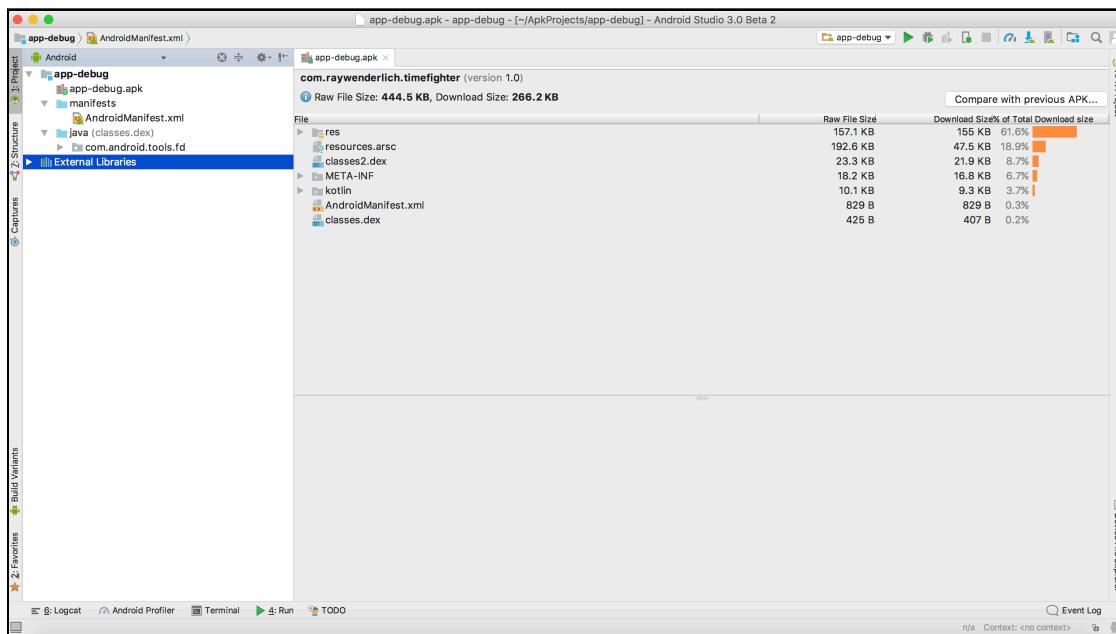
Clicking **Check out project Version Control** presents some version control systems that Android Studio supports, including Git and Mercurial. Android Studio also includes built-in support for Google Cloud.



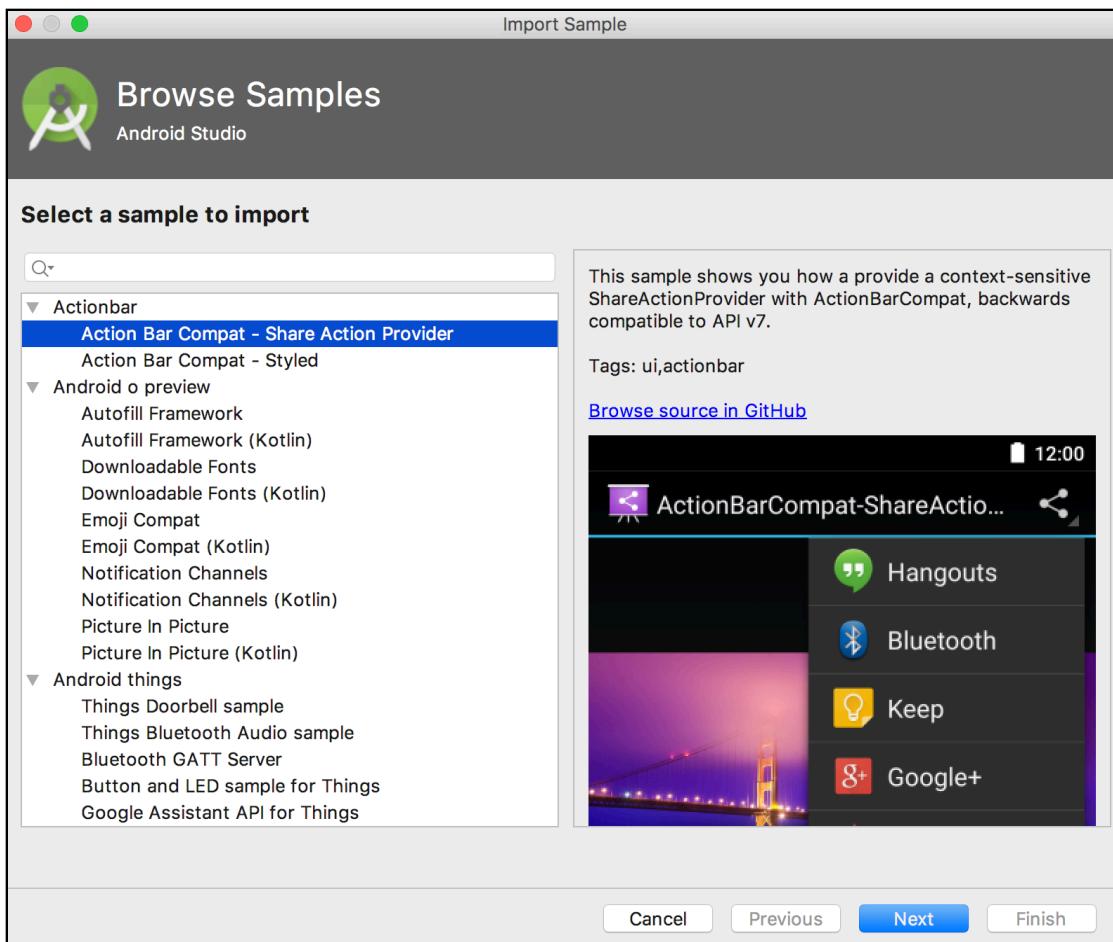
If you don't already use another version control system, you might consider using the tools within Android Studio for versioning control.

- **Profile or debug APK:** Gives you the option to select an **.apk** file from your computer's file system and run it on a device or emulator. This is helpful for gathering useful information about the app. **.apk** is the Android Package and the file where the app is stored.

The information you can gather ranges from the size of the app and its contents, to more sophisticated information gathered during runtime, like memory usage and network activity.



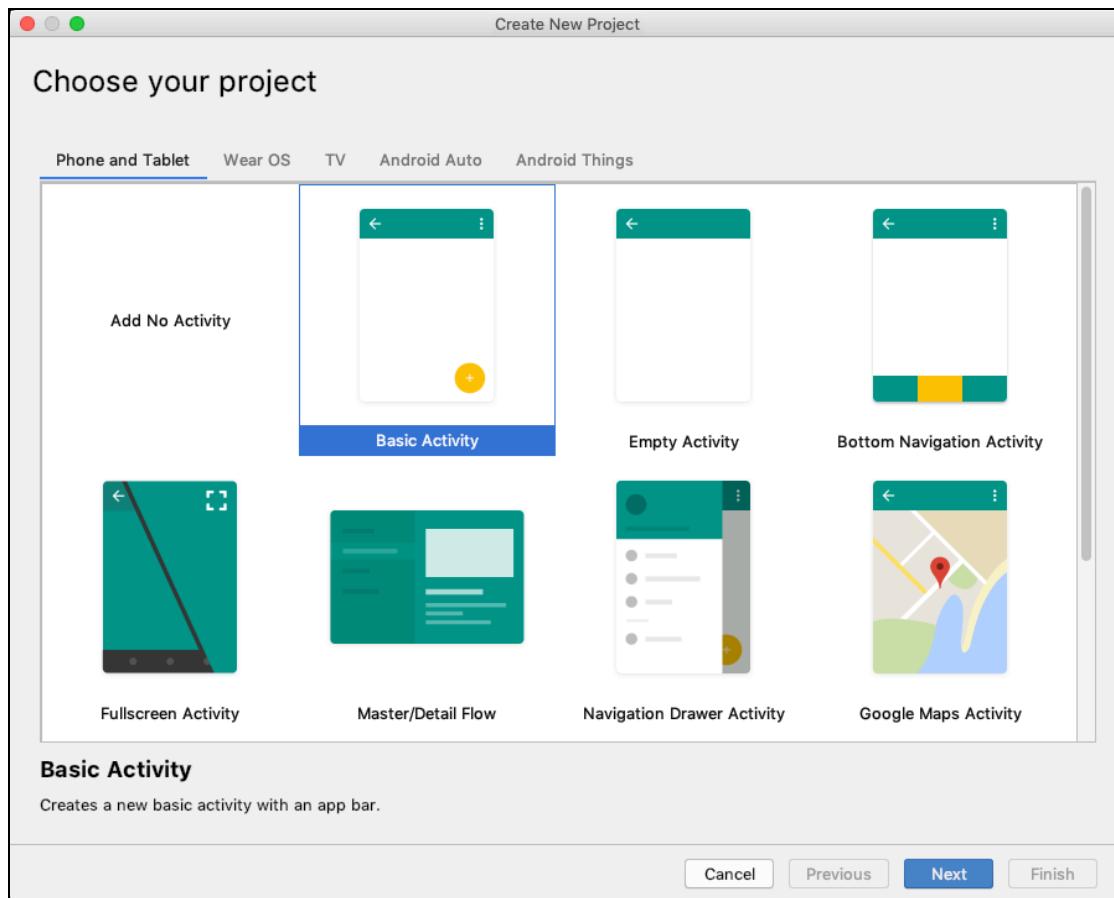
- **Import project (Gradle, Eclipse ADT, etc.):** Provides a way to import Android projects that have a complex build system setup or don't use Gradle for their build system. If you have a complex Android Studio project or an archive project to maintain, this is the place to go.
- **Import an Android code sample:** Lets you import a treasure trove of sample projects provided by Google that demonstrate Android features. You can find Android Studio projects covering topics from using emojis in your app, to more technical topics such as keeping your users' data secure.



Keeping it simple this time around. click **Start a new Android Studio project** to begin creating your app.

Creating a new Android project

After you click **Start a new Android Studio project**, a new window appears prompting for the project type.

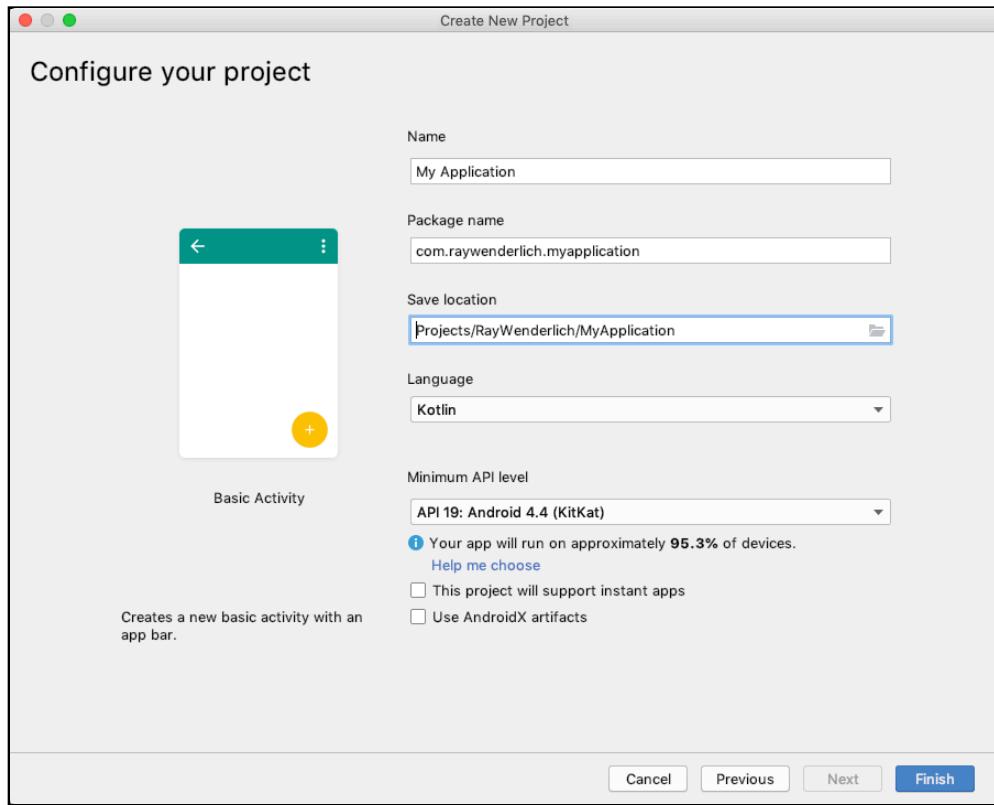


Along the top of the window there's a selection of tabs that gives you access to project setups for a specific version of Android.

Android is a popular operating system that runs on many different devices with different hardware. It extends beyond phones and tablets and runs on wearables such as watches, fitness trackers, television sets, and automotive systems within your car. It even runs on various electronics grouped under a wide umbrella known as the **internet of things**.

Trying to run an app on an Android watch when it's built for a phone will likely have issues. The **Choose your project** screen helps to avoid that and gets you set up with what you need quickly.

Take a moment to look at the available options. When you're done, select **Basic Activity** on the **Phone and Tablet** tab, and then click **Next**.



The next window requests information about your project. The field at the top, **Name**, is where you enter the name of your app. Type **ListMaker** into this field.

The second field, **Package Name**, is used to identify the packages within your app. Packages help organize how your code is structured, so it's best to name them in a way that describes what's inside of each package.

The package name also ensures your app is unique on a device. Android devices refuse to run apps if two apps contain the same package name, so it's an important field. It's also a security feature that helps keep your app safe from other apps who want to interfere with yours.

In this field, enter **com.raywenderlich.listmaker**.

Package name
com.raywenderlich.listmaker

The next field is **Save location**. This is where your project is created once you're done setting it up.

Clicking the folder to the right of the text field opens the file explorer where you can select a location to store the project. There's no wrong choice here, so choose a save location that's appropriate for you.

The next option is a drop-down menu where you can choose the language to build your app with. By default, this is set to Kotlin, so leave it as-is.

Targeting Android devices

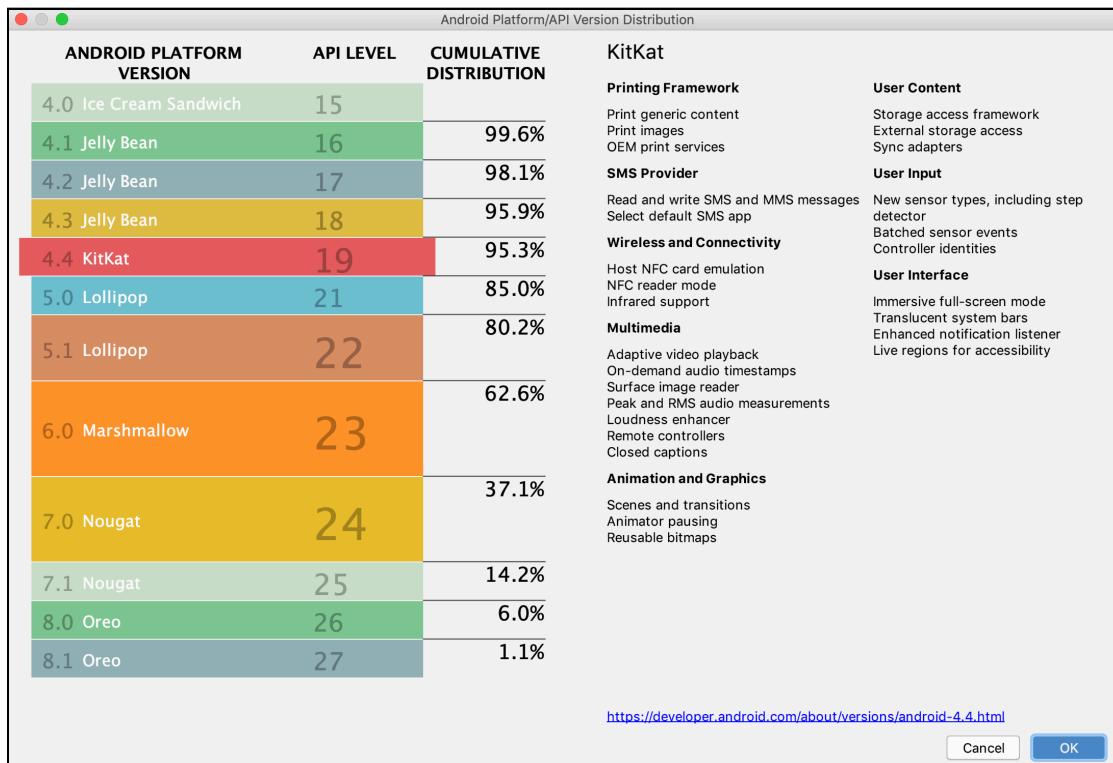
The next option is the **Minimum API level** screen. This drop-down menu specifies the earliest version of Android your program will support. This can be a tough decision.

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 difficult decisions on whether to use older features only while avoiding newer features found in more modern versions of Android.

There's good news, though: Android Studio can help you decide!

Below the **Minimum API level** drop-down, click **Help me choose**.



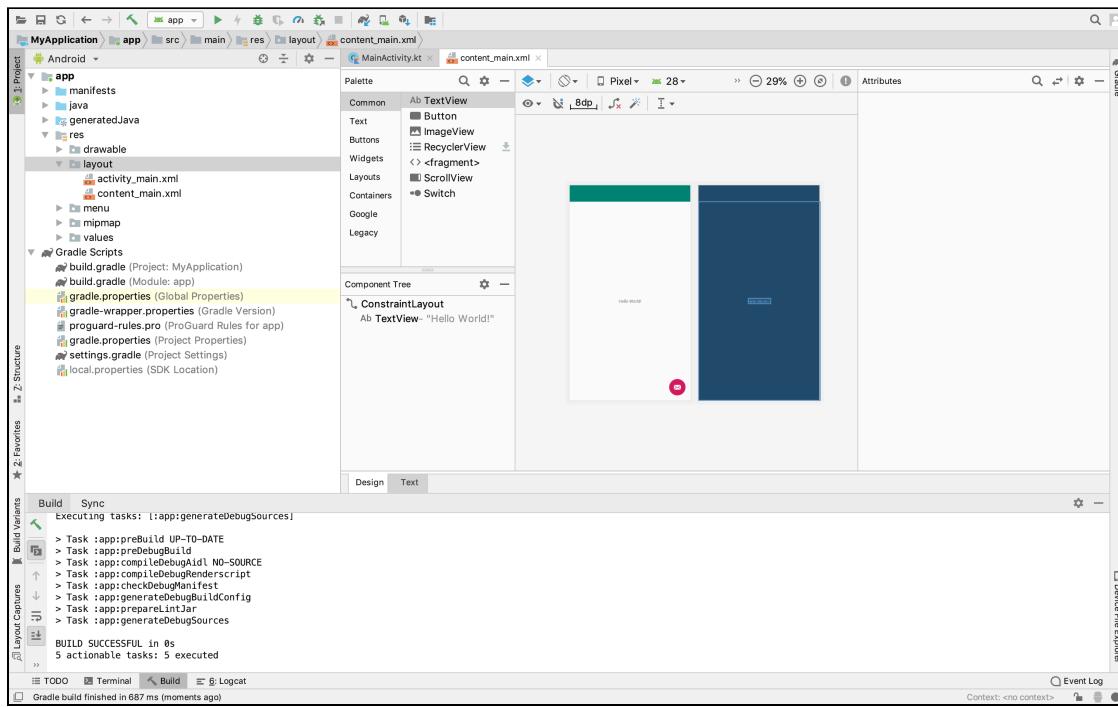
This is the **Android Platform Distribution** window. It shows the distribution Android versions running throughout the world. This gives you the opportunity to make an informed decision about which versions of Android your app will support.

The distribution works on a cumulative basis, shown by the percentages running alongside the colored boxes on the left. The earlier the Android version you choose, the more Android devices there are in the world that can run your app.

However, distribution isn't the only information this window provides: It also shows an overview of the features each version supports. **Android KitKat** is selected by default, but if you click each of the colored boxes, you'll see the features each one provides.

If you need additional information about a specific version, use the link in the bottom-right of the window, and you'll be sent to the About page on the Android developer site. This is a handy page. Use it whenever you're trying to decide on version support.

For now, you'll use the default selected version of Android: KitKat. Click **Cancel** to return to the **Target Android Devices** screen, and then click **Finish**. Android Studio takes your project settings and begins to create a new project for you. When it's done, Android Studio opens your project with your new Activity ready for editing.



Where to go from here?

Android Studio provides a way to set up a new project with some templates to get your app up and running as quickly as possible. Sample code is only a click away on the Welcome screen, and you can work with many exciting variants of Android with just a few clicks.

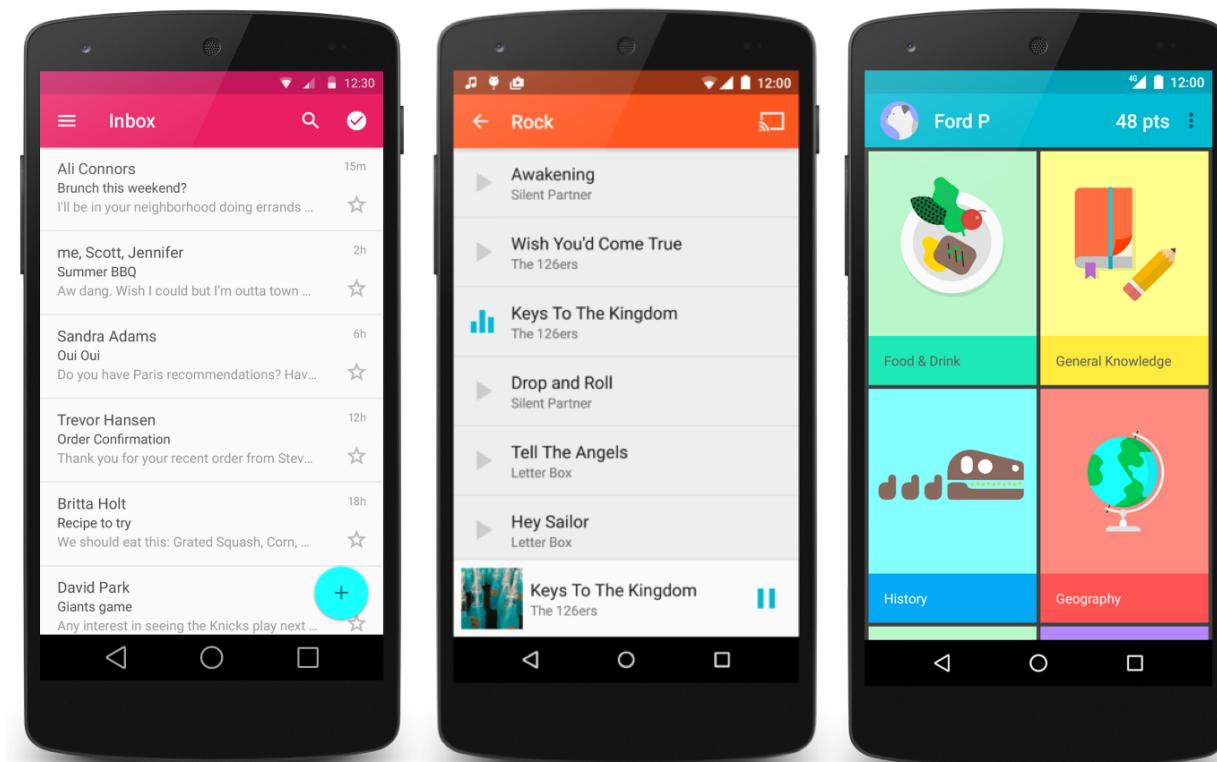
In the next chapter, you'll begin to build ListMaker.

Chapter 7: RecyclerViews

By Darryl Bayliss

In this chapter, you'll continue to build ListMaker, an app that helps organize all of your to-do lists in one handy place.

Lists are a common visual design pattern in apps because they allow developers to group collections of information together. They also allow users to scroll through and sometimes interact with each item in the list.



These apps are all using RecyclerView

An item in a list can range from a line of text to something more sophisticated such as a video with comments below it — a common style used in most social media apps.

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

1. Get started with RecyclerView.
2. Set up a RecyclerView Adapter to populate a list with data.
3. Set up a ViewHolder to handle the Layout of each item in the list.

Getting started

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

With the Android Studio project open, examine the project structure. In particular, look at the following files:

- **MainActivity.kt**: Located in the **java** folder.
- **activity_main.xml** and **content_main.xml**: Located in the **layout** folder.

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

In previous chapters, you used a single Layout file to build the user interface. In this project, there are two Layout files: **activity_main.xml** and **content_main.xml**.

Why are there two?

Open **activity_main.xml** and examine the **Component Tree**:

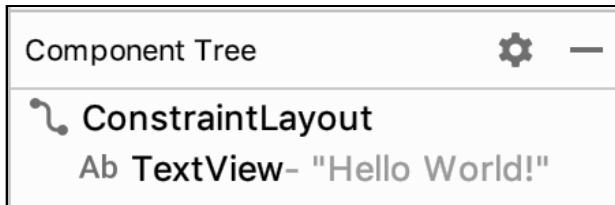


There's a **Toolbar** to display menu items you create, as well as a **FloatingActionButton**. You've used buttons before, so no surprises so far.

Keep scanning, and you'll see a component named **include**. This is where **content_main.xml** comes into play: The **activity_main.xml** Layout includes the Layout defined in **content_main.xml**, so you use both Layouts in the Activity.

While it looks strange to take this approach, it can be useful when using a Layout in multiple places within your app, or when the Layout is complex enough to benefit from being split into multiple files.

Open **content_main.xml** and review its contents:



It contains a single **TextView**.

Now that you know how everything is strung together, click **Run app** to see it in action.

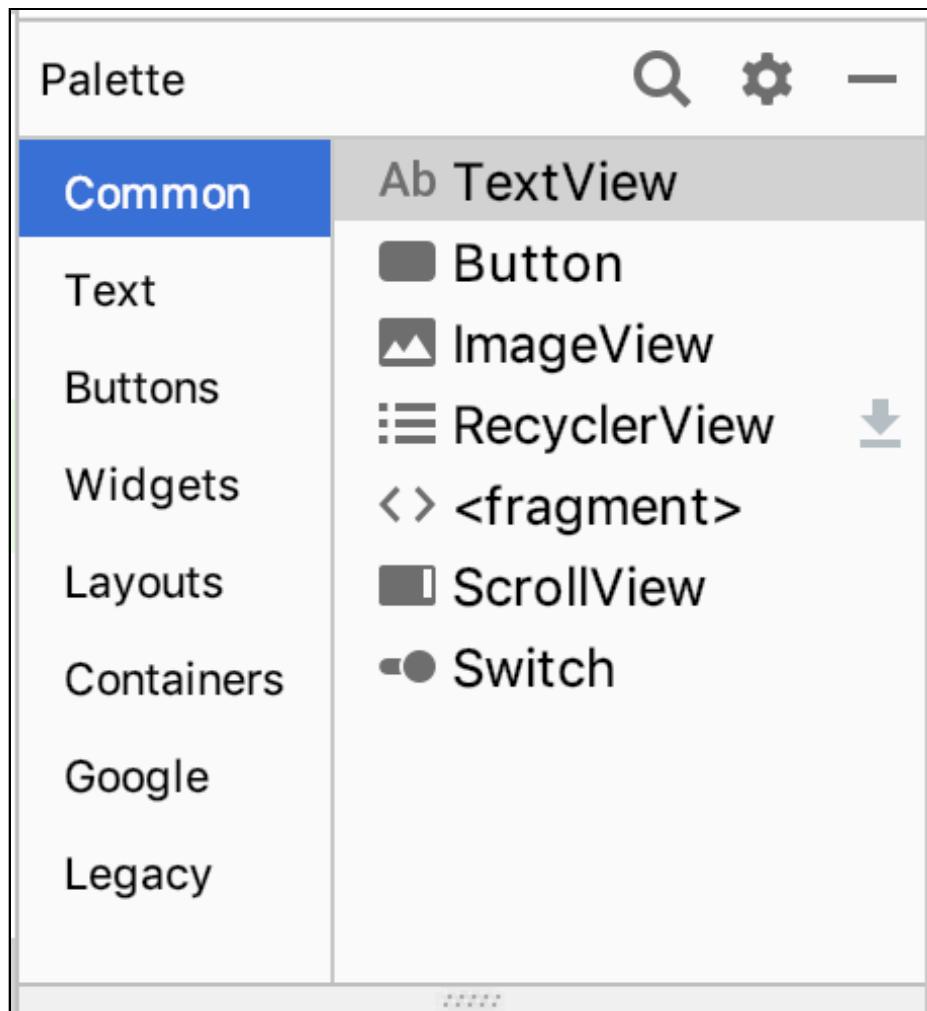


Adding a RecyclerView

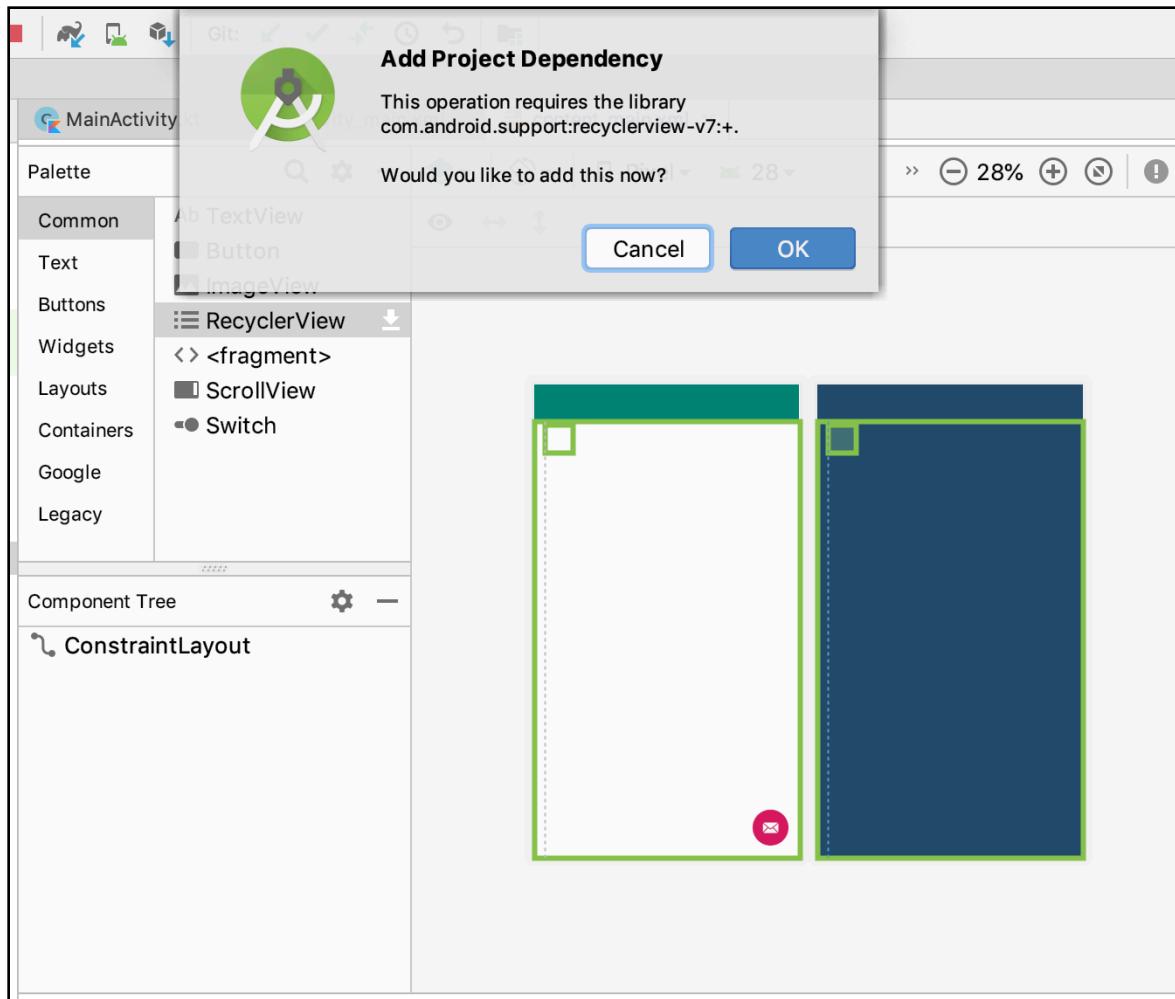
Did you notice that something important is missing from ListMaker? That's right! It's missing lists. At the moment, there isn't any way to show a list, let alone the master list of lists. It's like *Inception*, but...*Listception* instead.

Open `content_main.xml`. Then, select the `TextView` and delete it.

Next, go to the **Palette** and click **Common**.

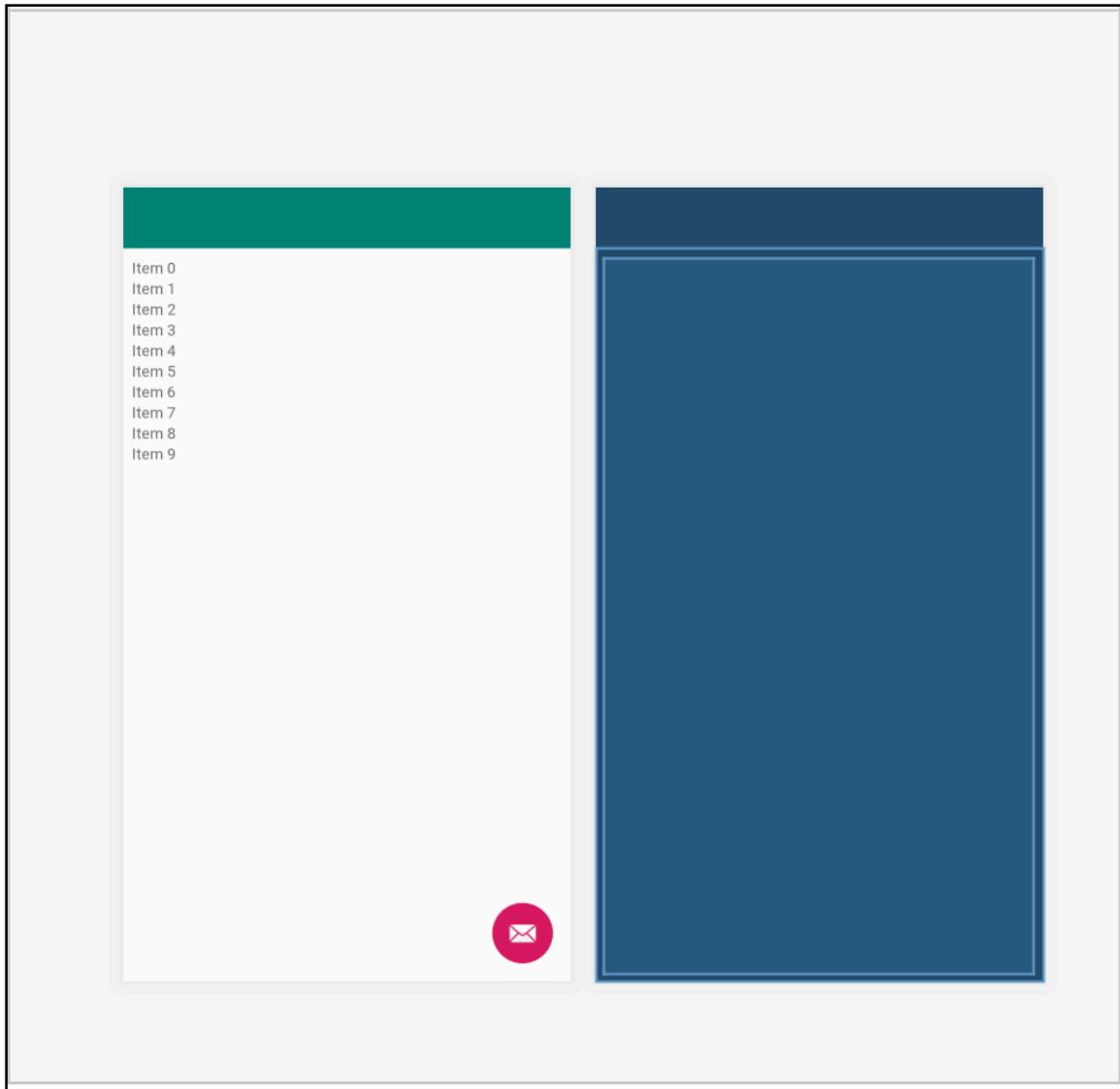


Click and drag a **RecyclerView** from the list of components into the middle of the Layout.



Hey, what's going on here? This prompt complains that the RecyclerView library isn't in the project because it exists as a separate library that Google provides — it even offers to add it for you.

Click **OK** to add the RecyclerView dependency, and let Android Studio set up the RecyclerView. After a short period of time, you'll see it in your Layout.

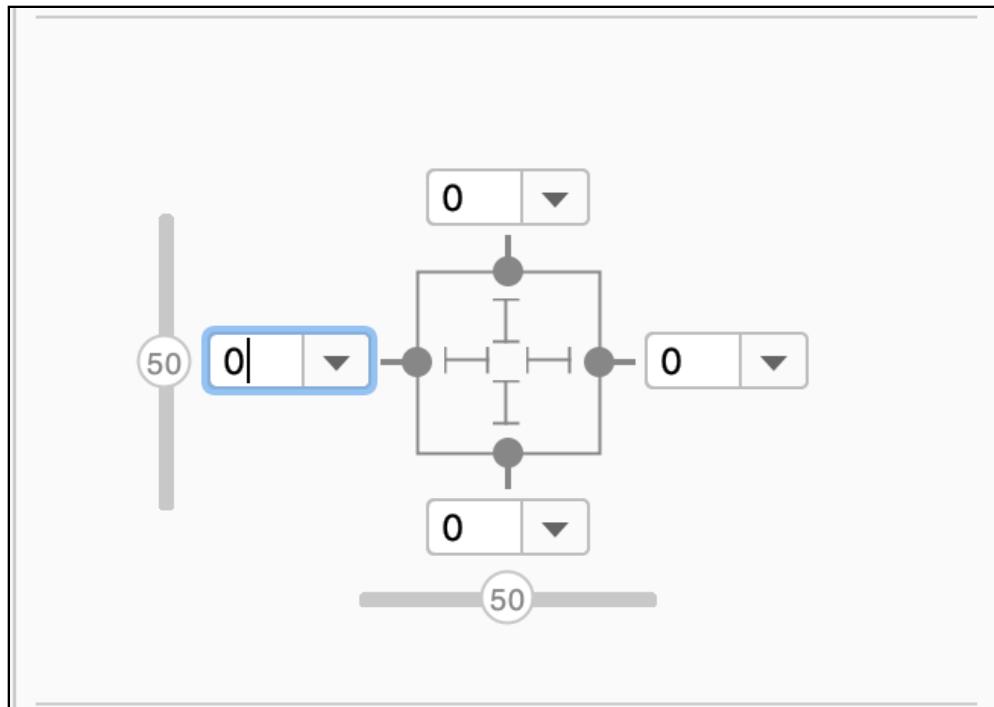


Behold, the RecyclerView!

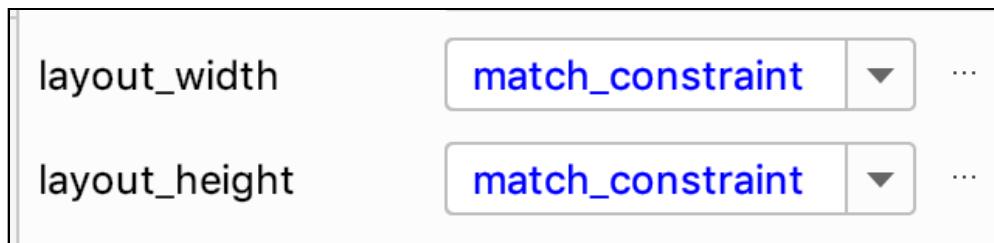
Once the RecyclerView is present in the Layout, select it. Then, move to the **Attributes** window and change the **ID** to **lists_recyclerview**. This lets you reference the RecyclerView in your Kotlin file.

Next, underneath in the Constraints pane, click all of the **plus symbols** to create constraint connections against the edges of the Layout for your RecyclerView.

Set the **margins** for each connection to **0**.



Finally, head back to the top of the Attributes and set **layout_width** and **layout_height** to **match_constraint**.

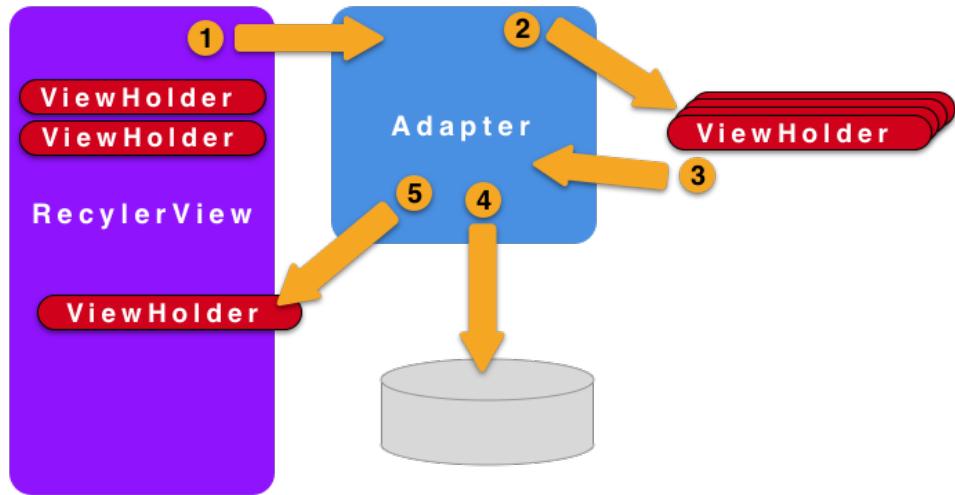


With the RecyclerView positioned correctly, it's time to use it.

The components of a RecyclerView

RecyclerView lets you display large amounts of data. Each piece of data is treated as an item within the RecyclerView. In turn, each of these items makes up the contents of the RecyclerView.

RecyclerView has two required components it uses to display a 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 how many items it has.
2. The RecyclerView asks the Adapter for an item or a ViewHolder at a given position.
3. The Adapter reaches into a pool of ViewHolders that have been created.
4. Either a ViewHolder is returned or a new one is created.
5. The Adapter then binds this ViewHolder to a data item at the given position.
6. The ViewHolder is returned to the RecyclerView for display.

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

ViewHolders are the visual containers for your item. Think of them as placeholders for each item in the table; this is where you tell the RecyclerView how each item should look.

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

Note: Janking is a common term 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 **MainActivity.kt** and add the following line to the top, just above `onCreate(savedInstanceState: Bundle?)`:

```
lateinit var listsRecyclerView: RecyclerView
```

You use the `lateinit` keyword to tell the compiler that a `RecyclerView` will be created sometime in the future.

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

```
// 1
listsRecyclerView = findViewById(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 to present your items in. This is similar to the Layouts you can use with your XML Layouts, and you need something to arrange your items in a linear format. The `LinearLayoutManager` works perfectly for this. You also pass in the Activity so that the Layout manager can access its Context.

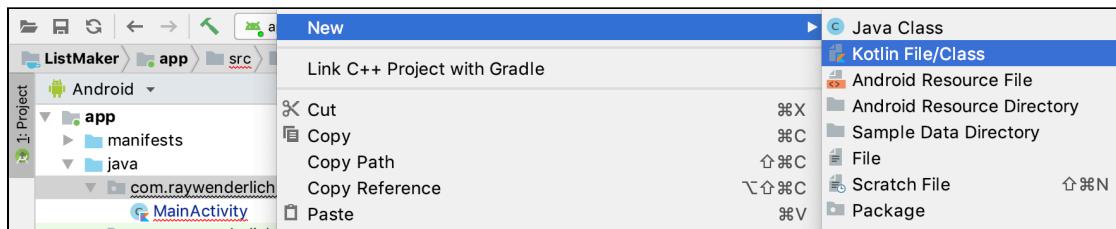
Note: `LinearLayoutManager` isn't the only layout provided by `RecyclerView`. Out of the box, `RecyclerView` also provides the `GridLayoutManager` and `StaggeredGridLayoutManager`.

3. The Adapter for the `RecyclerView` is set, letting it know to use this Adapter to acquire its data to show, and the `ViewHolders` to use to populate data with.

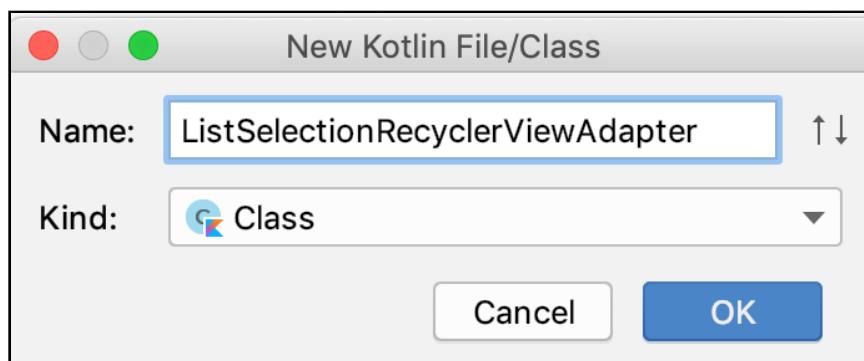
You'll notice that this shows an error in Android Studio. This is because `ListSelectionRecyclerViewAdapter` doesn't exist. You'll create this now.

Setting up a RecyclerView Adapter

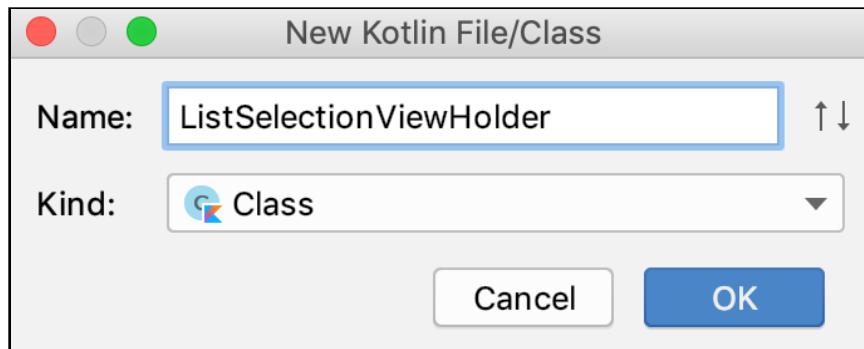
Right-click `com.raywenderlich.listmaker` in the Project navigator. In the floating options that appear, hover over **New**. In the next set of options that appear, click **Kotlin File/Class**.



In the popup that appears, enter `ListSelectionRecyclerViewAdapter` for the **Name** and change the **Kind** drop-down to **Class**. Then, click **OK**.



Android Studio creates the class for you. Now repeat the process and create a `ViewHolder` class too. Name this new class `ListSelectionViewHolder`.



You're ready to turn these classes into recycling machines. Open **ListSelectionViewHolder.kt** add a primary constructor to the class, so you can pass in the View for the ViewHolder and have it extend `RecyclerView.ViewHolder`:

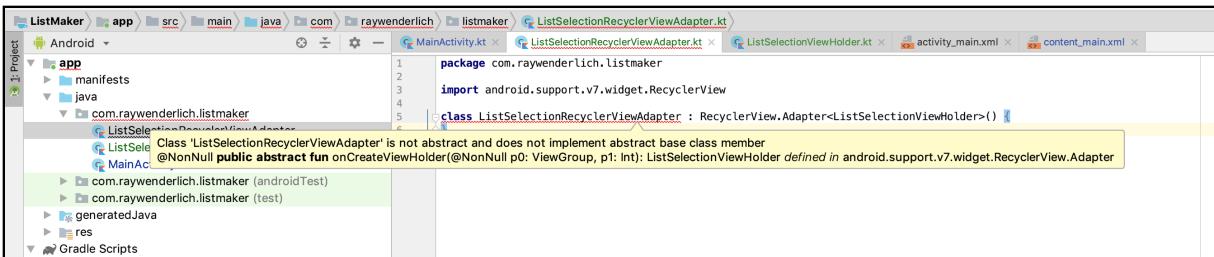
```
class ListSelectionViewHolder(itemView: View) :  
    RecyclerView.ViewHolder(itemView) {  
}
```

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

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

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

Notice that the name of the class is underlined with red. Move your mouse cursor over it, and Android Studio informs you why there's an error.

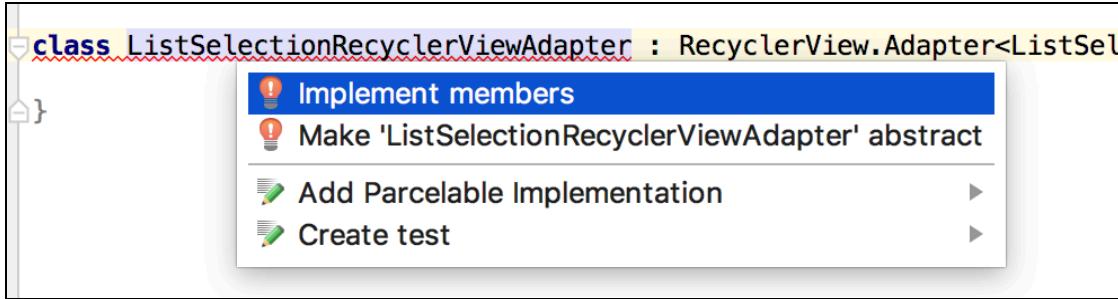


Because this class inherits from `RecyclerView.Adapter`, it needs to implement additional methods so it knows what to do when used in conjunction with a `RecyclerView`.

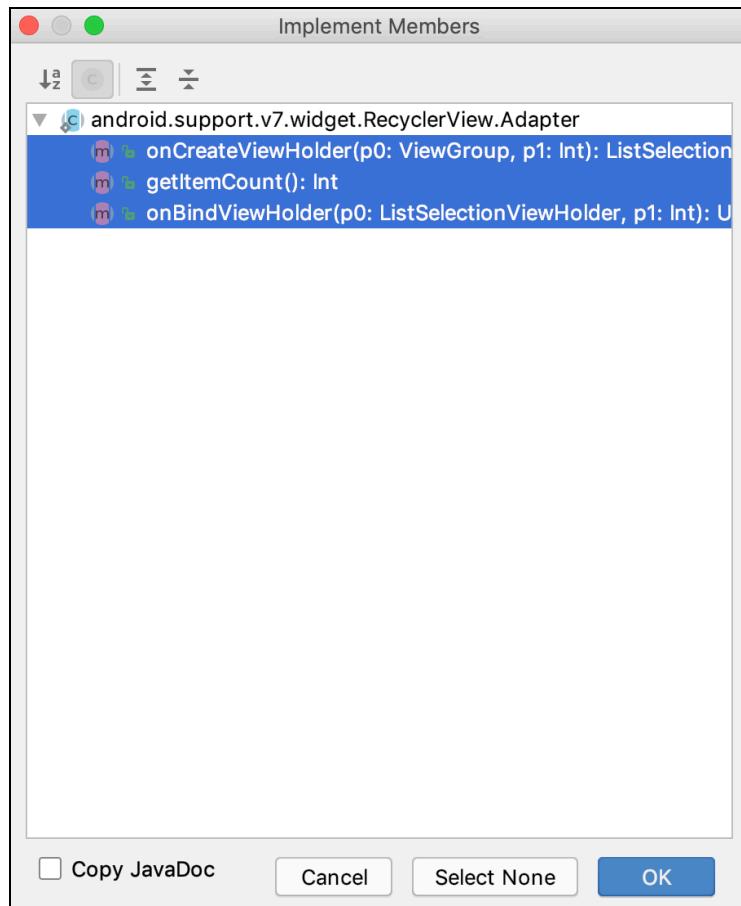
With your cursor over the class name, press **Option-Enter** to get a selection of options.

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**.

Having trouble getting this to work? Alternatively, hover the cursor over the class name and press **Control-I** or select **Code** along the top toolbar of Android Studio, and click **Implement Members**



Click **Implement Members**, and a new window appears with options for various methods to implement. Since the Recycler Adapter needs each one, you'll add them all. Ensure `onCreateViewHolder()` is highlighted, then **Shift-click** on the bottom-most available member.



Finally, click **OK** and Android Studio does the rest of the work for you. Make sure those methods are implemented in your `ListSelectionRecyclerViewAdapter`:

```
class ListSelectionRecyclerViewAdapter :  
    RecyclerView.Adapter<ListSelectionViewHolder>() {  
    override fun onCreateViewHolder(p0: ViewGroup, p1: Int):  
        ListSelectionViewHolder {
```

```
    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.
}

override fun onBindViewHolder(p0: ListSelectionViewHolder, p1: Int) {
    TODO("not implemented") //To change body of created functions use
    File | Settings | File Templates.
}
```

Filling in the blanks

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

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

You'll implement the mock list titles first. In **ListSelectionRecyclerViewAdapter.kt**, add the following new variable at the top of the class:

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

Here, you create an array of strings to use as the list titles. In future chapters, you'll change this to something more sophisticated — but for now, an array will do.

`getItemCount()` determines how many items the RecyclerView has. You want the size of the array to match the size of the RecyclerView, so you return that.

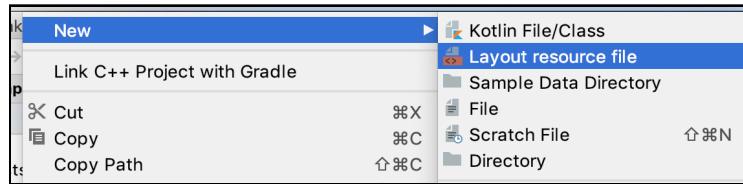
In `getItemCount()`, you return the size of the array, like so:

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

Your Adapter now knows how many items to display on the screen. Next, you need to create the Layout needed for the ViewHolder to display each item in the RecyclerView.

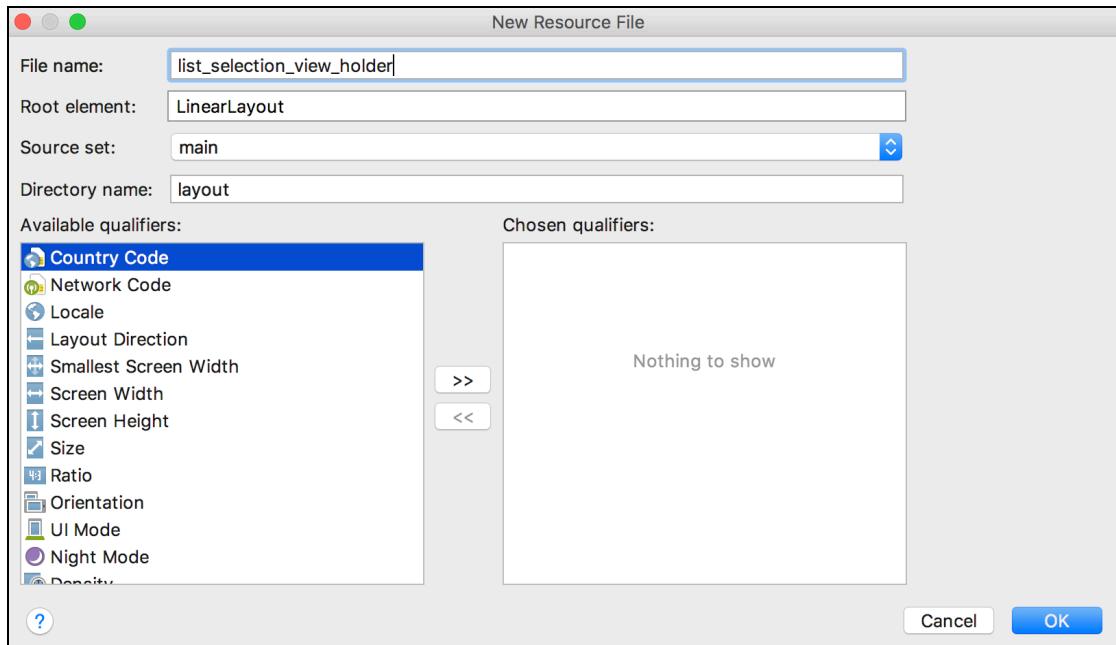
Creating the ViewHolder

In the Project navigator on the left, right-click on the **layout** folder and create a new Layout:



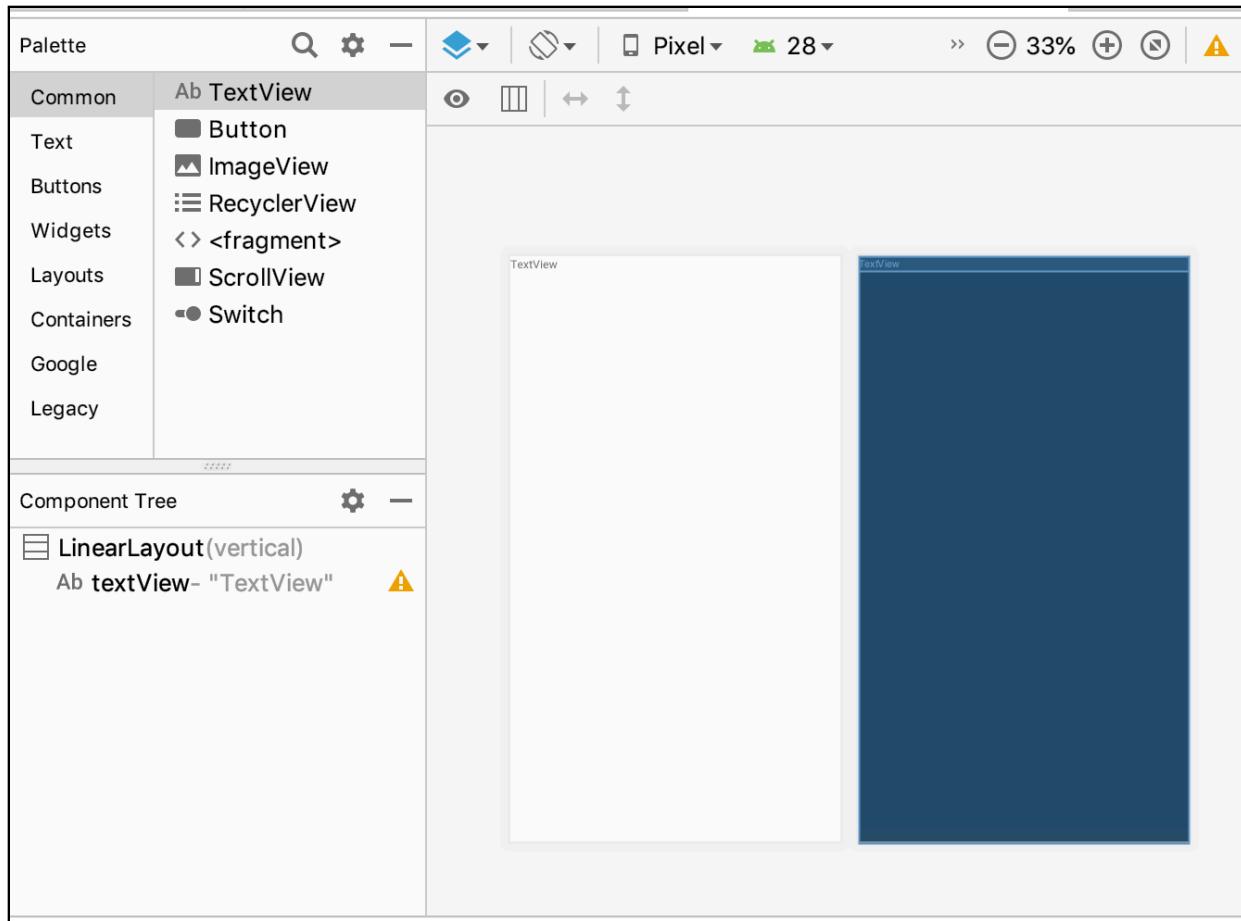
In the new window that appears, type **list_selection_view_holder** for the **File name**; this serves as the name of the Layout when created.

The **Root Element** defines the first tag in the Layout. For this Layout, you'll use a **LinearLayout**, so type **LinearLayout** into the text field and click **OK** at the bottom of the window.

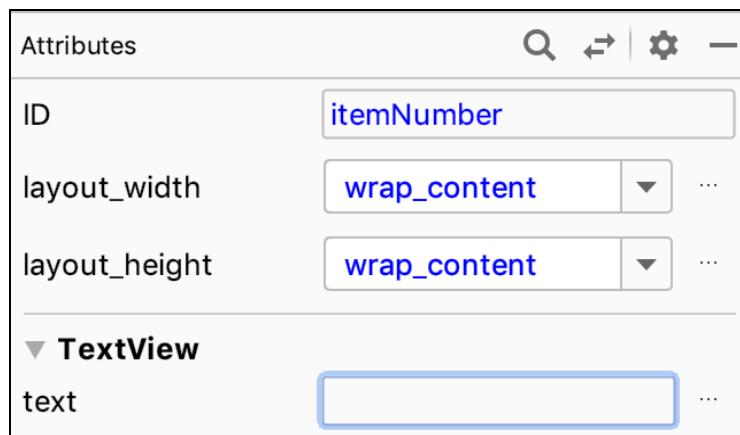


Android Studio opens your new Layout, ready for you to add the Views you want the ViewHolder to contain. You need two TextViews here: one to tell you the position of the list in the RecyclerView, and one to tell you the name of the list.

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:

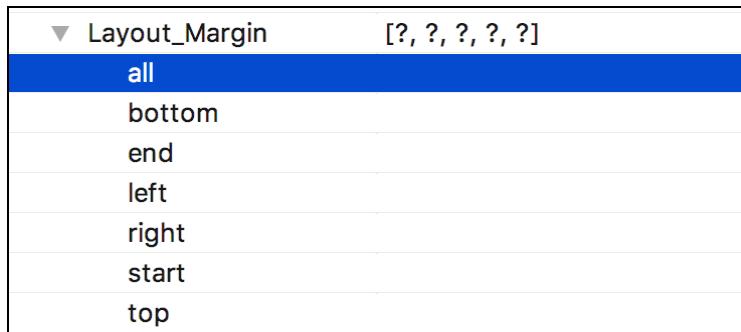


To ensure the text isn't sitting too closely to the edge of the screen, you need to give it space on its left edge. Click **View All Attributes** at the top of the Attributes window:



This brings up a list of attributes that you can change for the `TextView`. Feel free to look at what attributes you can change. There are a lot of them.

You need to set the **all** parameter of the `Layout_Margin` attribute to add padding to this `TextView`. First, find the `Layout_Margin` attribute and click the arrow next to it to reveal a drop-down for each of the parameters.

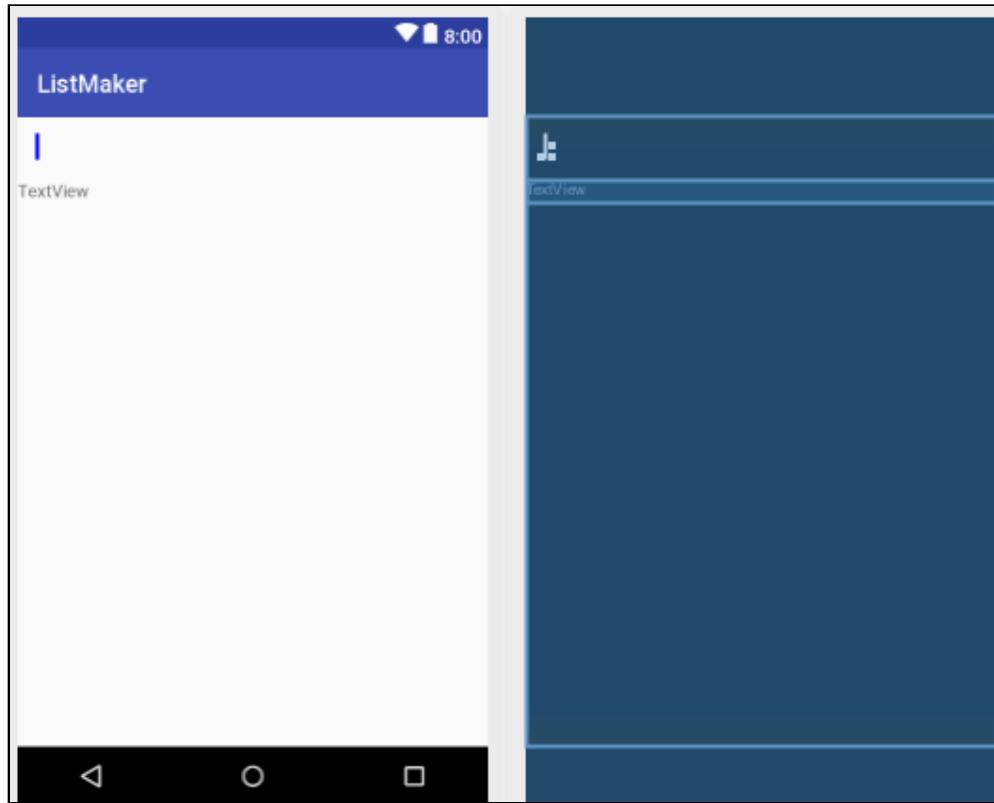


In the **all** text field, type **16dp**. This tells the `TextView` to pad itself by 16 **density pixels** (dp) on all sides.



Note: A density pixel is a unit of measurement Android uses to lay out your View relative to the size of the device screen. Because devices have many different screen sizes, using absolute pixels isn't feasible as screens will render differently from device to device. To learn more, review the Android Developer Documentation: https://developer.android.com/guide/practices/screens_support.html.

With that done, repeat the process for the first TextView, by dragging another TextView into the Layout. Place it underneath the first TextView.

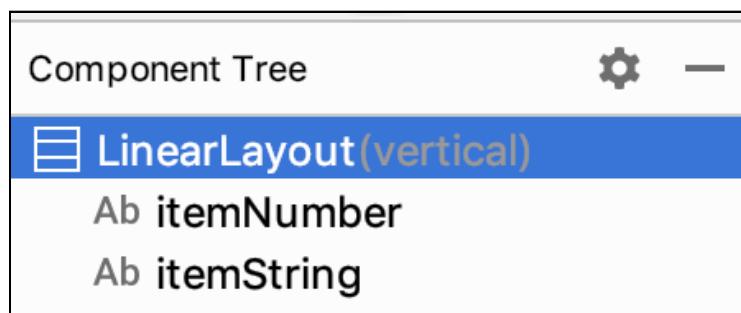


Change the ID of the second TextView to **itemString**. Change the **layout_width** and **layout_height** to **wrap_content**. Remove the placeholder text from the **text** attribute and change the **all** parameter in the **Layout_Margin** attribute to **16dp**.

You're nearly done with this Layout — there's only one more thing to do.

Currently, the TextViews are laid out in a vertical orientation. However, a horizontal orientation is better suited for this app, so you need to change some attributes on the **LinearLayout** the Layout uses.

Click the **LinearLayout** in the Component Tree window:



In the Attributes Window, click the drop-down button on the **orientation** attribute and select **horizontal**.



Change the **layout_width** and **layout_height** attributes to **wrap_content** to make the ViewHolder only as big as it needs to be:



You're ready to use the Layout. Open **ListSelectionRecyclerViewAdapter.kt** and change `onCreateViewHolder()` to the following:

```
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 parent context of the Adapter to create itself and 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. The Boolean value is used to specify whether the View should be attached to the parent. Always use false for RecyclerView layouts as the RecyclerView attaches and detaches the Views for you.

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

2. The ViewHolder is created from the Layout and returned from the method.

Binding data to your ViewHolder

Now that you've created a ViewHolder, you have to bind the list titles to it. To do this, you need to know what Views to bind your data to. You 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.

Open **ListSelectionViewHolder.kt** and add the following properties to the class, so the ViewHolder has references to the new TextViews:

```
val listPosition = itemView.findViewById(R.id.itemNumber) as TextView  
val listTitle = itemView.findViewById(R.id.itemString) as TextView
```

Open **ListSelectionRecyclerViewAdapter.kt** again and change `onBindViewHolder()` to the following:

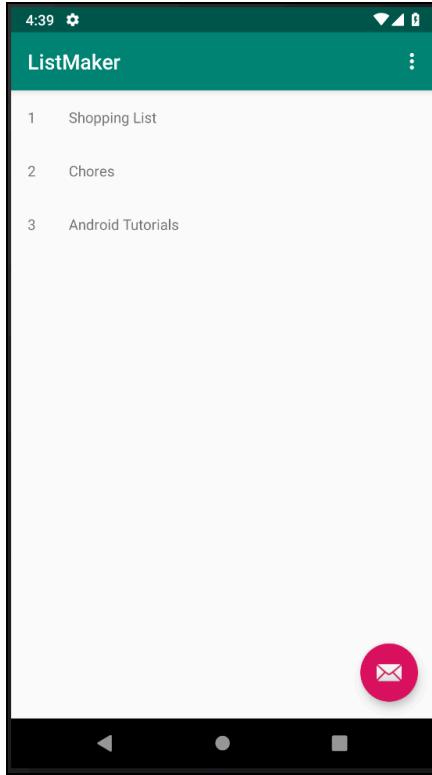
```
override fun onBindViewHolder(holder: ListSelectionViewHolder, position: Int) {  
    holder.listPosition.text = (position + 1).toString()  
    holder.listTitle.text = listTitles[position]  
}
```

For each call of `onBindViewHolder()`, you take the TextViews you created in the ViewHolder and populate them with their position in the list and the name of the list from the `listTitles` array.

This gets called repeatedly as you scroll through the RecyclerView.

The moment of truth

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

If you want to learn more about RecyclerView, review 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 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 how to use different LayoutManagers, and how to swipe to delete items in your list.

8

Chapter 8:

SharedPreferences

By Darryl Bayliss

In the previous chapter, you set up an Activity to use the powerful RecyclerView.

In this chapter, you'll update ListMaker so that it can create, save, and delete lists, as well as show these lists in the RecyclerView. You'll also learn about **SharedPreferences** and how you can use them to save and retrieve user data.

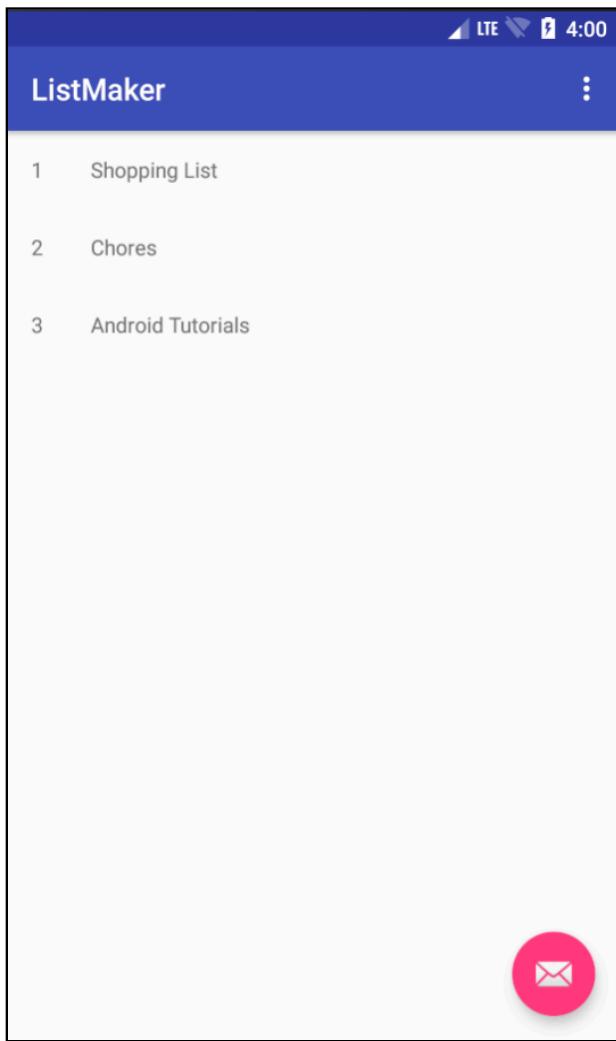


Getting started

If you're following along with your own project, open it and keep using it with this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **ListMaker** app inside the **starter** folder.

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

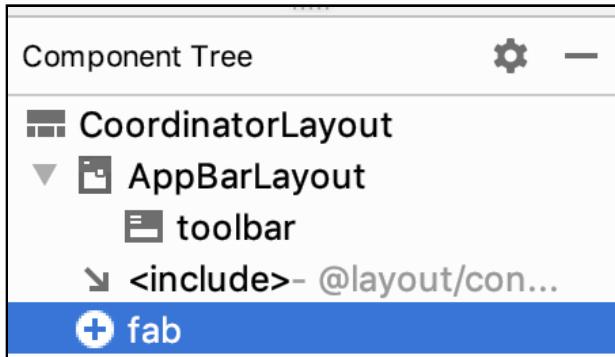
With the ListMaker project open in Android Studio, it's time to run the project in the emulator.



Notice the round pink button at the bottom right? That's called a **Floating Action Button**, better known as a **FAB**. You use a FAB to highlight an important action on the screen.

Creating lists is the most important action in ListMaker, so it makes sense to use a FAB to add new lists. Your first task is to select an appropriate icon for the new list button — an envelope image doesn't convey this action well.

Open `activity_main.xml`, and in the **Component Tree** window, select the **Floating Action Button**.



In the **Attributes** window on the right-hand side of Android Studio, locate the **srcCompat** text field. This is where you assign the image to the button. Currently, it has a value of `@android:drawable/ic_dialog_email`.

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



Thanks to this change, users will better understand the purpose of this button. Now it's time to add the code that allows users to create a new list.

Adding a Dialog

When users tap the FAB in ListMaker, you want this to action to open a Dialog where they can enter a name for their new list. This Dialog will contain labels that prompt users for this information.

Rather than hardcoding these prompt strings, open `strings.xml` and add the following:

```
<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 strings in **strings.xml**. By doing so, your app can be localized for different languages should you decide to do so in the future.

Open **MainActivity.kt**, and at the bottom of the file, add a new method:

```
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, _ ->  
        dialog.dismiss()  
    }  
  
    // 4  
    builder.create().show()  
}
```

With this method, you:

1. Retrieve the strings defined in **strings.xml** for use in the Dialog.
2. Create an `AlertDialog.Builder` to help construct the Dialog. An `EditText` View is created as well to serve as the input field for the user to enter the name of the list.

The `inputType` of the `EditText` is set to `TYPE_CLASS_TEXT`. Specifying the input type gives Android a hint as to what the most appropriate keyboard to show is. In this case, a text-based keyboard, since you want the list to have a name.

The title of the Dialog is set by calling `setTitle`. You also pass in the content View of the Dialog. In this case the `EditText` View, by calling `setView`.

3. Inform the Dialog Builder that you want to add a **positive button** to the Dialog; this tells the Dialog 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.

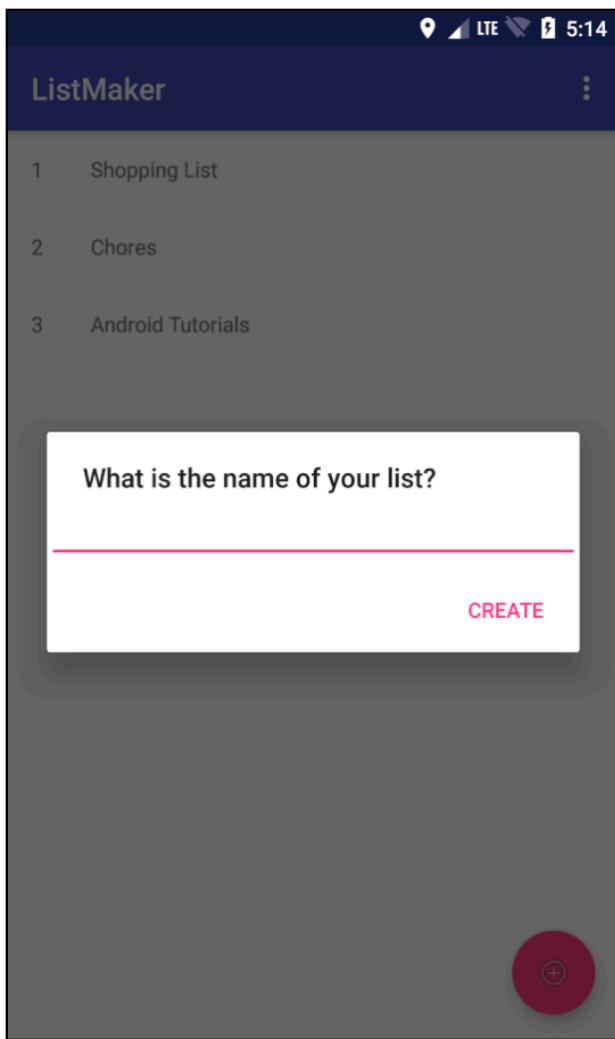
In this case, a positive button makes sense because the user is creating something. You pass in `positiveButtonTitle` as the label for the button and implement an `onClickListener`. For now, you dismiss the Dialog. You'll handle the resulting actions behind the button in the next section.

- Finally, you instruct the Dialog Builder to create the Dialog and display it on the screen.

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

```
fab.setOnClickListener {  
    showCreateListDialog()  
}
```

Run the app and tap on the pink FAB in the bottom-right of the screen. You'll see the Create List Dialog appear as expected.

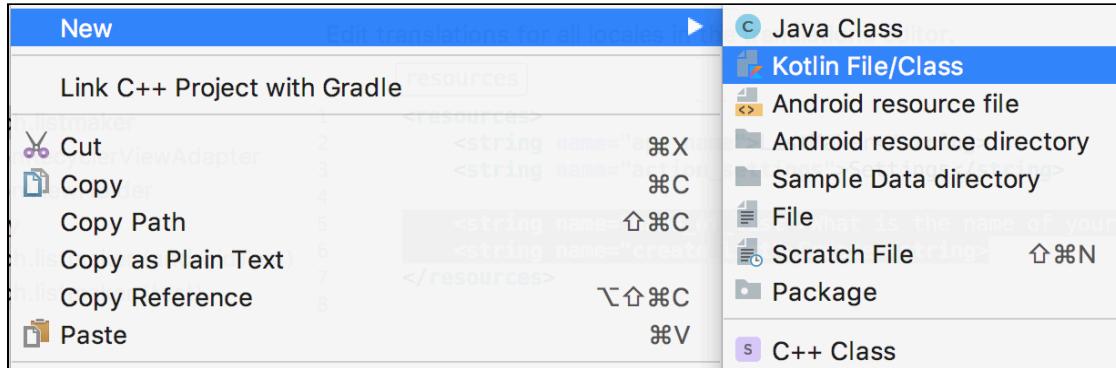


Type in a name for the list, click **Create** and... nothing happens. That's because you need to add some code to handle the creation of the list inside the `onClickListener` of the positive button for the Dialog.

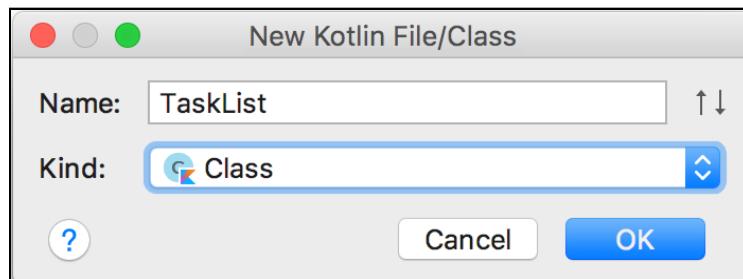
Creating a list

Before you can begin to work with a list, you need to define a list for ListMaker. You'll start by creating a model for a list to use throughout the app.

In the Project navigator, Right-click **com.raywenderlich.listmaker**. In the options that appear, select **New ▶ Kotlin File/Class**:



In the window that appears, name the new Kotlin file **TaskList**, change the kind to **Class** and click **OK**.



Android Studio creates and displays the 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()) {  
}
```

Next, you need a way to save the 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 save small bits of data in your app quickly, SharedPreferences is one of the first solutions you should consider.

Behind the scenes, SharedPreferences writes key-value pairs to a single file. You can configure it to write to multiple files for more complex apps. You can also allow other apps to access your apps' SharedPreferences store if you think other apps have a valid reason to access your data.

Note: SharedPreferences is a quick way to persist and retrieve data. However, there are better alternatives to SharedPreferences when you have complex data needs, which you'll learn about in later chapters.

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

```
class ListDataManager(private 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 pass a Context into ListDataManager and add a method named `saveList(list: TaskList)` to persist the list to sharedPreferences.

Here's what's going on:

1. Get a reference to the app's default SharedPreference store via `PreferenceManager.getDefaultSharedPreferences(context)`. With the `PreferenceManager` object it returns, append `.edit()` to it to get a `SharedPreferences.Editor` instance. This allows you to write key-value pairs to `SharedPreferences`.
2. Write `TaskList` to `SharedPreferences` as a set of Strings, passing in the name of the list as the key. Since the tasks in `TaskList` is an array of strings, you can't store it directly in a string, so you convert the tasks in `TaskList` to a **HashSet** which `SharedPreferences` can use as a value to save. Since `HashSet` is a Set, it ensures unique values in the list.
3. Instruct the `SharedPreferences` Editor instance to apply the changes. This writes the changes to ListMaker's `SharedPreferences` file.

That takes care of saving lists, but that's only half the solution! You also need a way to retrieve lists from SharedPreferences. Add the following method below `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 = ArrayList(taskList.value as HashSet<String>)
        val list = TaskList(taskList.key, itemsHashSet)
        // 5
        taskLists.add(list)
    }

    // 6
    return taskLists
}
```

Going through this code step-by-step:

1. Grab a reference to the default SharedPreferences file. This time, you don't request a `SharedPreferences.Editor` since you only need to read from SharedPreferences, not write to it.
2. Call `sharedPreferences.all` to get the contents of the 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 the items in the Map you received from SharedPreferences using a `for` loop. For each iteration, take the value of the object and attempt to cast it to a `HashSet<String>`. Recall from `SaveList()` that you couldn't store a `TaskList` directly as a string, so you converted the list of tasks into a `HashSet`. You perform the reverse of this to retrieve the tasks and convert them back to an `ArrayList` of strings. Then, recreate the `TaskList` by passing the key of the `MapEntry` as the name of the `TaskList` and the `ArrayList` of strings as the tasks.

5. Finally, add the newly reconstructed TaskList into the empty ArrayList you created earlier.
6. Once you iterate over the entire set of items you retrieved from SharedPreferences, return the contents of taskLists to the caller of the method.

Hooking up the Activity

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

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

This creates a new ListDataManager as soon as the Activity is created.

Next, update the positive button's onClickListener in showCreateListDialog():

```
builder.setPositiveButton(positiveButtonTitle) { dialog, _ ->
    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** created earlier.

Using the adapter, you pass the TaskList into the adapter using addList, so it knows it has something to show. Don't worry about the **Unresolved reference** error on addList; you'll create this method shortly.

That's the background work done for this feature. Now you need to let the RecyclerView and its Adapter know about the new datasource.

In the **onCreate**: method, replace the set up code for the 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 relatively straightforward code:

1. You get a list of `TaskLists` from `listDataManager`, ready for use.
2. Remember that static array of list titles you added earlier? You're beginning to replace that with the list of `TaskLists` your app stores. Ignore the **Too many arguments** error since you're going to update `ListSelectionRecyclerViewAdapter` to accept a parameter.

Now the RecyclerView Adapter has a source of information to display; there are a few changes you need to make to ensure everything works with the new lists.

Open `ListSelectionRecyclerViewAdapter.kt` and update the class definition to accept an `ArrayList` of `TaskList` in its primary constructor:

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

Find `onBindViewHolder()` and update it to use the list to populate the ViewHolder instead of the static array of strings:

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

Modify `getItemCount()` to get the size of the list:

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

Finally, create the `addList()` method you called from `MainActivity` to let the adapter know you have a new list to display. Add the following code to the bottom of the Adapter class:

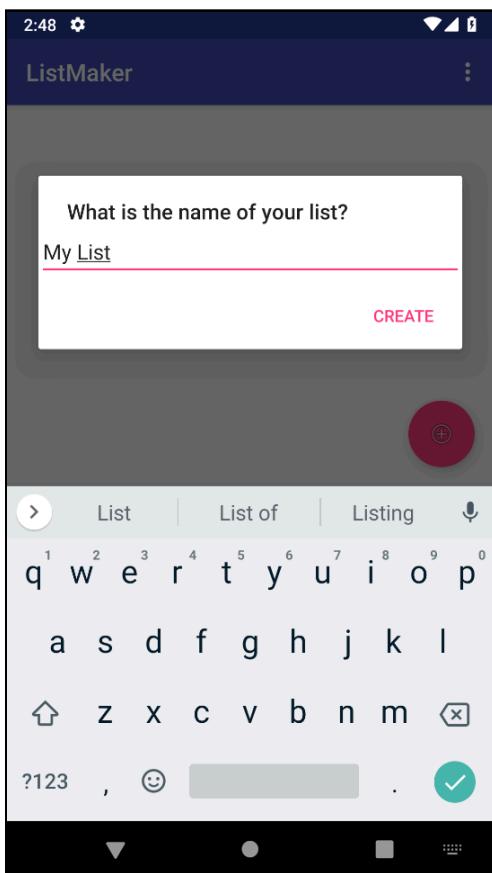
```
fun addList(list: TaskList) {  
    // 1  
    lists.add(list)  
  
    // 2  
    notifyDataSetChanged(lists.size-1)  
}
```

Here's what's happening:

1. You update the `ArrayList` with the new `TaskList`.
2. You call `notifyItemInserted()` to inform the Adapter that you updated the data source, and you update the `RecyclerView`. In this case, the data source is the `ArrayList` passed into the `ListSelectionRecyclerViewAdapter`, and any necessary `ViewHolders` are created to populate each View with the right data for each position.

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

Run the app, tap the FAB to display the Create List Dialog and give the list a name.



Tap **Create** and the new task list appears in the RecyclerView.



You're not quite done — there's one thing left to verify. Does the list stick around after you stop and restart the app?

Click the **Stop** button in Android Studio; it's the big red square in the toolbar at the top.



Your device stops running the app and goes back to the home screen. Once again, run the app from Android Studio and the list returns.



With this test done, you can be certain the app persists the list to SharedPreferences and loads it when relaunched. Great job!

Note: You may notice the order of the list titles changing as the app relaunches. This highlights one of the issues when using SharedPreferences.

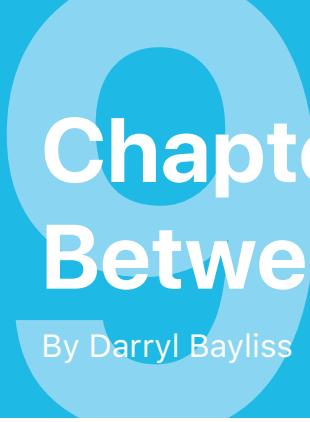
SharedPreferences is only a key-value store; it doesn't order your data.

For this example, SharedPreferences is a great option to store and read data quickly. However, as your needs become more complex, you should consider other methods of storage that adhere to ordering; these are explained later in the book.

Where to go from here?

SharedPreferences is the simplest way to persist values in an Android app, so it's worth keeping in your toolbox.

In this chapter, you learned how to write and read values from SharedPreferences and put that knowledge to good use in ListMaker — like letting your users save and load their lists! The next logical step is to let users add items to their lists, which is 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. However, as your app gets more complicated, trying to cram more visual elements into a single Activity becomes difficult, and it can make your app confusing for users. Keeping an Activity dedicated to a single task removes this problem.

At the moment, ListMaker has no way to add items to the lists you create. This is a perfect task to put in a separate Activity — which is what you'll do in this chapter — and when you're done, you'll have learned how to:

1. Create another Activity.
2. Communicate between Activities using an Intent.
3. Pass data between Activities.

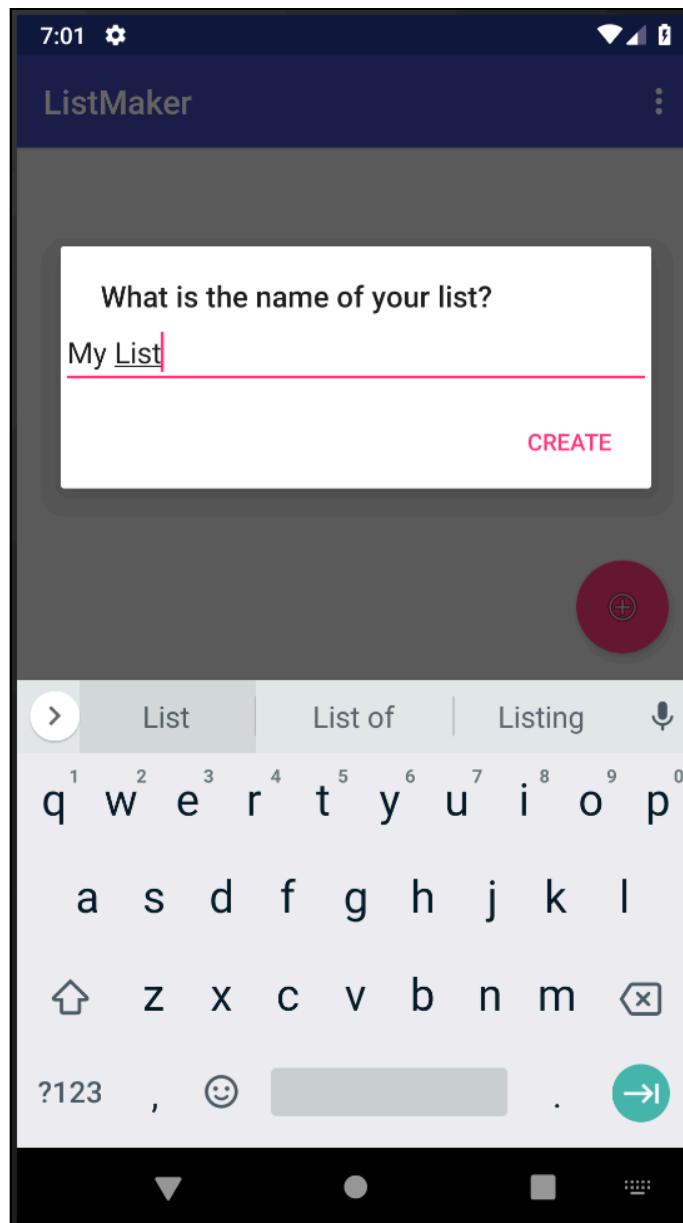
Getting started

If you're following along with your own project, open it and keep using it with this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **ListMaker** app inside the **starter** folder.

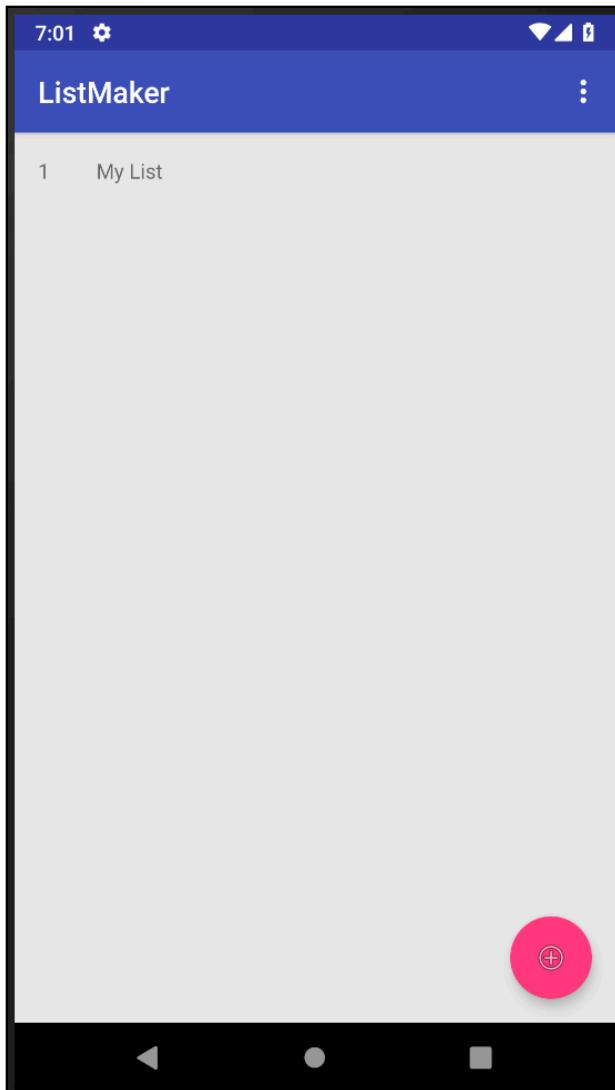
The first time you open the project, Android Studio takes a few minutes to set up your environment and update its dependencies.

Note: If you added lists in the previous chapter, you'll continue to see them inside your app. If you want to start fresh, delete the app from your device, then keep going with this chapter. All of the previous list data gets deleted when you delete the app.

With the ListMaker project open, run the app. When it appears, tap the Floating Action Button in the bottom-right and enter the list title as **My List**.



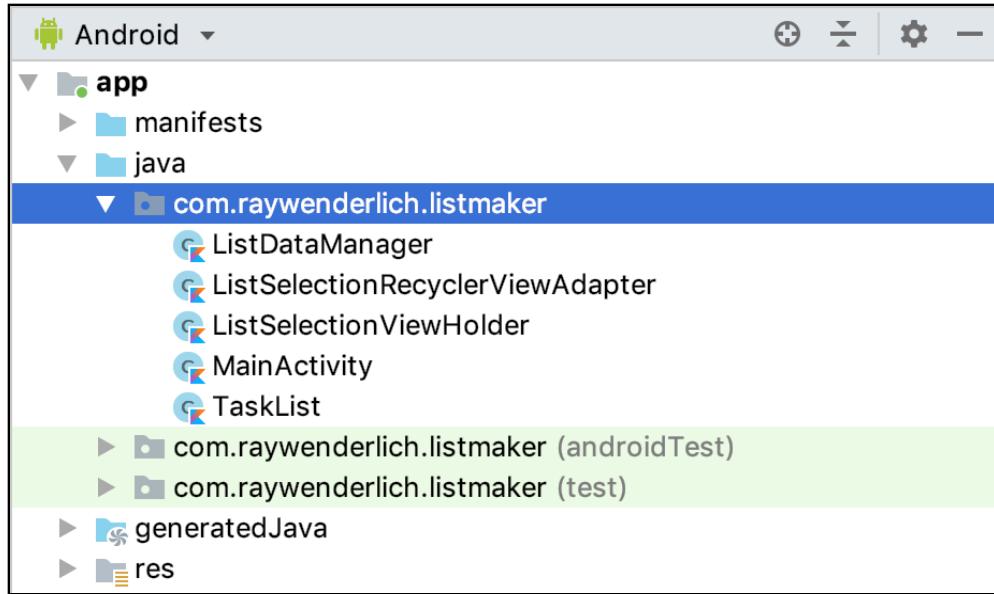
Tap **Create**, and the new list name shows up in the RecyclerView.



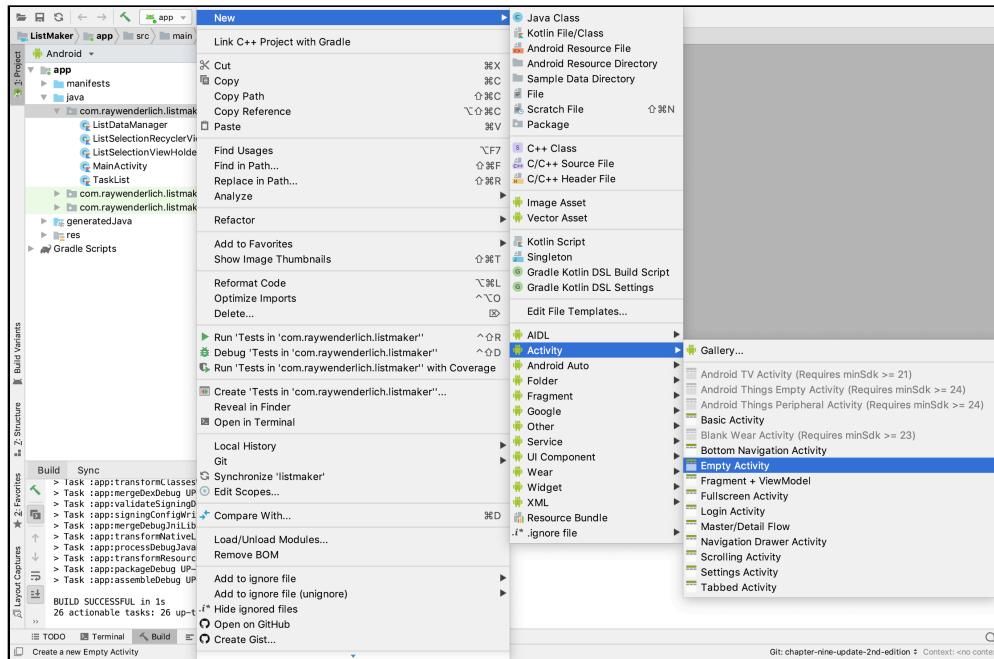
That works, but it isn't too useful. When you tap the title of the list in the RecyclerView, nothing happens. Wouldn't it be great if something were to happen? Absolutely!

You'll fix this by creating another Activity. As a general rule, Activities should focus on a single task, so the logic within an Activity stays clean and simple as you build it. Single task Activities also benefit your users because navigation between screens becomes more intuitive.

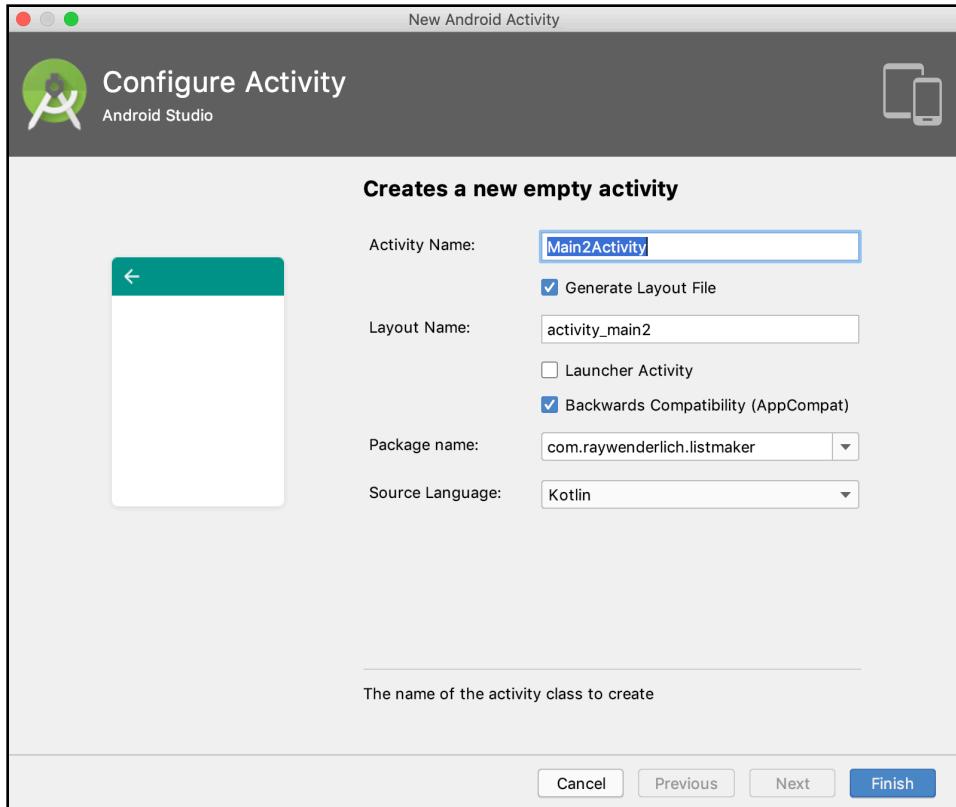
In the Project navigator, right-click **com.raywenderlich.listmaker**.



In the floating selection that appears, choose **New > Activity > Empty Activity**.



Android Studio presents a new window that gives you the opportunity to customize the new Activity before creating it.



Creating a new Activity

The **Configure Activity** wizard provides some fields to customize the Activity:

- **Activity Name:** Sets the name for the Activity. This is used to name the Kotlin class associated with the Activity.
- **Generate Layout File:** A checkbox to generate a Layout XML for use in conjunction with the Activity. This is checked by default since it's rare that you won't want to create a Layout.
- **Layout Name:** A field to set the name of the XML file used to hold the Activity's Layout.
- **Launcher Activity:** A checkbox that gives you the option to set the new Activity as the first one shown when the app starts. This is unchecked by default. You'll see how to change the starting Activity later in the chapter.

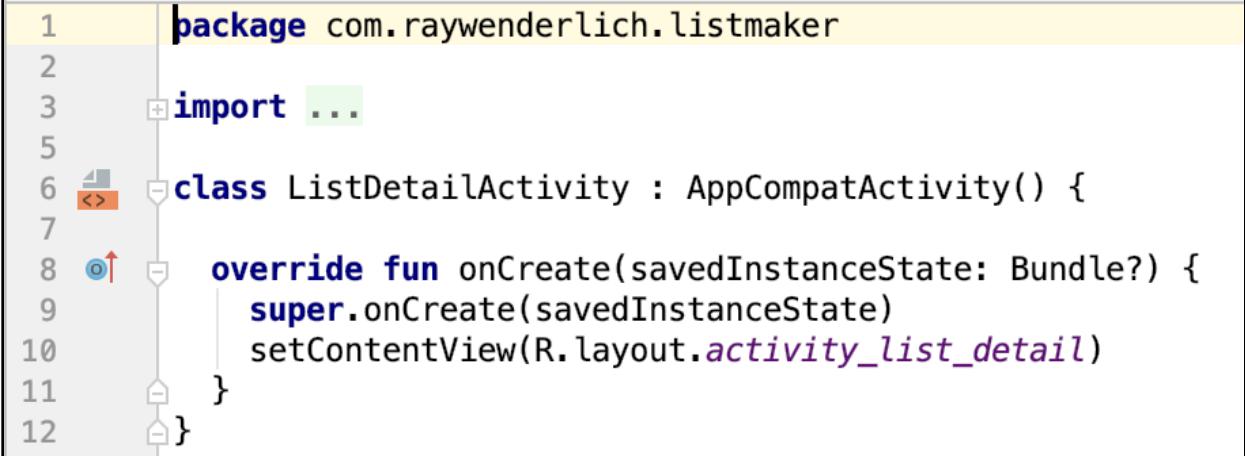
- **Backwards Compatibility (AppCompat)**: A checkbox that sets the Activity to inherit from AppCompatActivity. This ensures backward compatibility across the many versions of Android. You'll learn more about this topic in Chapter 28 "Android Fragmentation & Support Libraries", but for now, it's best to leave this checked.
- **Package Name**: A field that sets the package the Activity class will be created in. Since you only have one package in the project, this defaults to **com.raywenderlich.listmaker**.
- **Source Language**: A drop-down that lets you choose the programming language the Activity will use. The choices are Java and Kotlin. In this project, the default is Kotlin.

Note: If you don't see **Source Language** as an option, try scrolling inside the area where all of the options are located. Depending on your screen size, the value for language might not be visible unless you scroll.

Most of the options 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 keeps your filenames related to follow Android platform conventions and to make it easier to find your files later.

Click **Finish** in the bottom-right of the window, and Android Studio creates the new Activity.



```
1 package com.raywenderlich.listmaker
2
3 import ...
4
5
6 class ListDetailActivity : AppCompatActivity() {
7
8     override fun onCreate(savedInstanceState: Bundle?) {
9         super.onCreate(savedInstanceState)
10        setContentView(R.layout.activity_list_detail)
11    }
12 }
```

Android Studio even hooks up your Layout, so you don't have to do this yourself. However, 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**. It's important because it tells an Android device everything it needs to know about your app.

Android is strict about its requirements for a manifest. The file name must be **AndroidManifest.xml** and has to be located in the correct spot in the project file hierarchy. Without this file, Android refuses to run your app.

On the left side of Android Studio in the Project navigator, navigate to **app > manifests > AndroidManifest.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=".ListDetailActivity"></activity>
        <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>
    </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 inside **app/src**

Also, don't worry about the warning that appears in the manifest that says App is not indexable by Google Search; consider adding at least one Activity with an ACTION_VIEW intent filter. See issue explanation for more details.

This is a tag you can add to make the details of your app manifest searchable by Google's search engine.

This is an XML-based file containing various tags. The main tags in this file are `manifest`, `application` and `activity`; there are plenty more you'll use in chapters to come.

The `manifest` tag is the root element of the app manifest. You must declare all of the other tags within this tag. You also need to declare the package where your code sits within this tag as well. This is a security measure to ensure 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 the Settings.

Perhaps the most interesting tags 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 come 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` is used to set your main Activity as the startup Activity.

You'll 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 difficult work of updating the manifest, so you don't have to do this yourself.

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

Intents

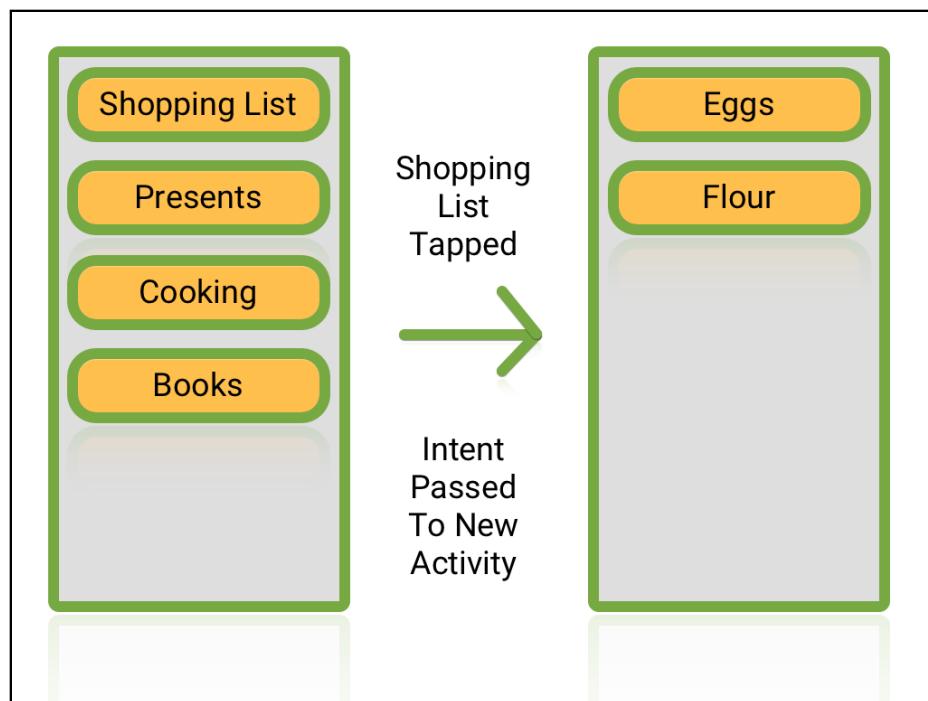
Now that you have the two Activities, it's time to give your app the ability to navigate between them. In the MainActivity, you have two main points of entry for the new activity:

- When a user taps the name of the list in the RecyclerView.
- When a user enters the name of a new list and taps Create.

You'll navigate between the two Activities 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, so it's best that you use them for your app communication as well. Intents are incredibly flexible and can perform a wide range of tasks such as communicating with other apps, providing data to processes or starting up another screen.

In fact, your app is 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

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 essential method. It creates an Intent. Here's the breakdown:

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're 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 this 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()` call.

You also pass in a constant named `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 a method call to inform the current Activity to start another Activity, making use of the information provided within the Intent.

Intents and Parcels

You've already set up the presentation of your new screen. However, there's a small problem: `TaskLists` can't be passed through Intents since Android doesn't understand how to handle that type of object. Fortunately, there's 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 down your object into types the Intent system is already familiar with: strings, ints, floats, Booleans and other objects which conform to `Parcelable`. You can then put all of 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 the list as an Extra in the Intent you set up earlier.

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

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

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. 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()`.

Note that `readString()` and `createStringArrayList()` return optionals. You know that the objects a `TaskList` expect are a string and an `ArrayList` of strings, so you use the non-null assertion operator (`!!`) to get the non-optinal values.

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 function, 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 some methods in it using 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. **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 of the data from the `Parcel`.

Note: For more information about the `Parcelable` interface, review the Android Documentation: <https://developer.android.com/reference/android/os/Parcelable.html>.

Bringing everything together

Now that the `TaskList` can 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 at the bottom of `MainActivity.kt`:

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

This key is used by the Intent to refer to a list whenever it needs to pass one to the new Activity.

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

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

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

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

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

When you create a new list now, the app passes that list to the new Activity. Perfect!

You also want the same thing to happen if a user taps on an existing list in the RecyclerView. Open **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`:

```
override fun onBindViewHolder(holder: ListSelectionViewHolder, position: Int) {
    holder.listPosition.text = (position + 1).toString()
    holder.listTitle.text = lists[position].name
    holder.itemView.setOnClickListener {
        clickListener.listItemClicked(lists[position])
    }
}
```

Here, you create an object to listen for any taps that occur on the rows of the RecyclerView. When a click event happens, this tells the listener which list was tapped.

Open **MainActivity.kt** and update the class declaration to state that it conforms to the `ListSelectionRecyclerViewClickListener` interface you just created:

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

Then, at the bottom of the class above the companion object, add the following:

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

This calls `showListDetail()` and passes in the list the user taps on.

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

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

`MainActivity` is now ready to send your list. Before you can see it in action, you need to do one final thing: Handle the Intent in `ListDetailActivity`.

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

```
lateinit var list: TaskList
```

Recall that the Intent is passing a list that the detail Activity is operating on, so you create a local variable to hold the list.

Next, in `onCreate()`, you need to retrieve the list you passed in as an Extra. Change `onCreate()` so it matches this:

```
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, you:

1. Use the key assigned to the list in `MainActivity.kt` to reference the list in the Intent and assign it to the `list` variable.
2. Assign the title of the Activity to the name of the 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 toolbar of Android Studio. Once the app is running, create a new list and name it **Your New List**. Tap **Create**, and behold: The 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 that it can 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.

If you find the `Parcelable` interface cumbersome to implement. The **Kotlin Android Extensions** (KAE) library contains an annotation called **Parcelize** that automatically generates the parcelable code for your annotated class. You can learn how to use the `Parcelize` annotation at: <https://www.raywenderlich.com/84-kotlin-android-extensions>.

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 is empty.

In this chapter, you'll add to that Activity using familiar components such as a RecyclerView to display the list, and a FloatingActionButton to add tasks to the list. You'll also learn how to communicate back to the previous Activity using an Intent.

Getting started

If you're following along with your own project, open it and keep using it with this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **ListMaker** app inside the **starter** folder.

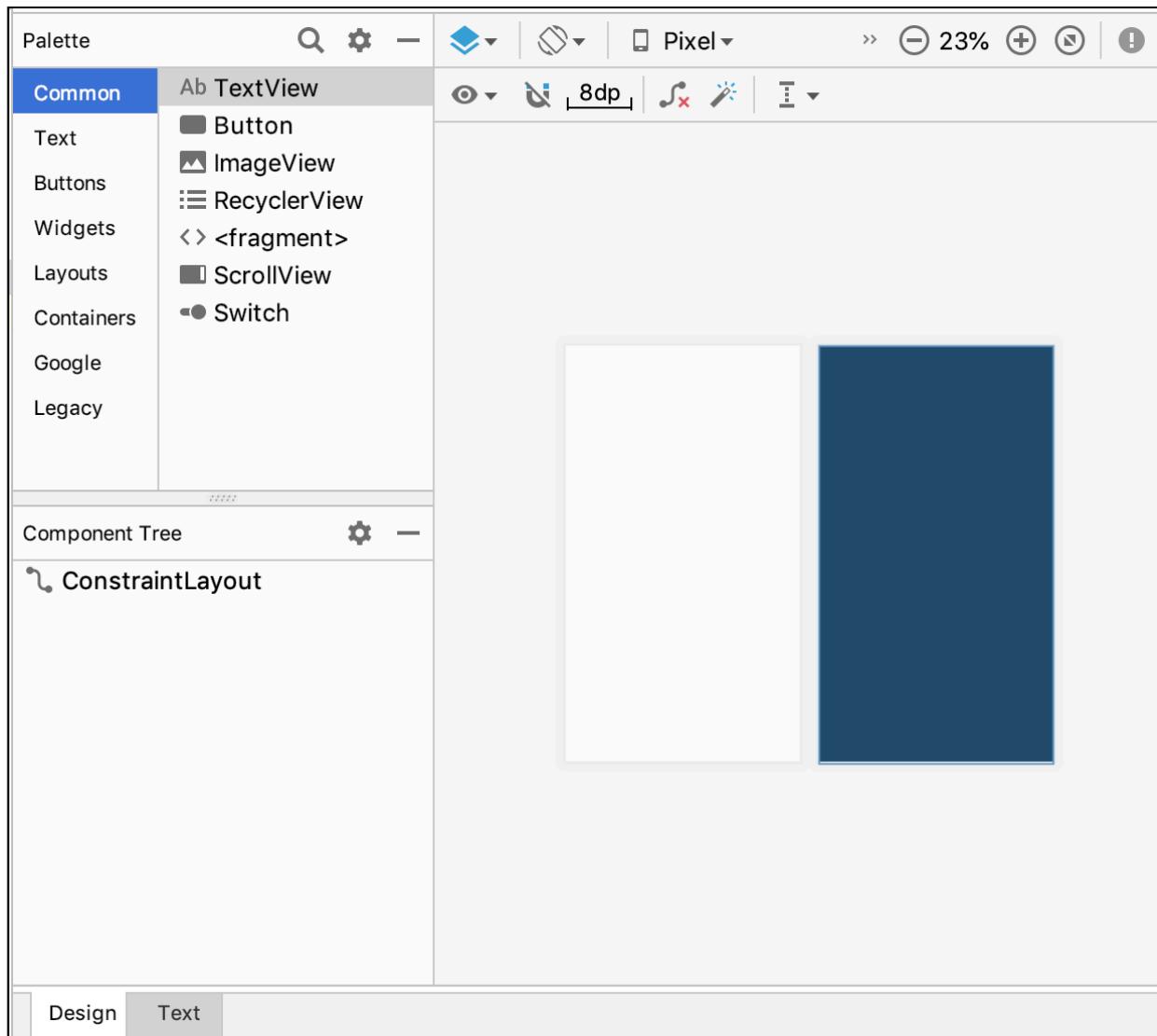
The first time you open the project, Android Studio takes a few minutes to set up your environment and update its dependencies.

Open **ListDetailActivity.kt** and review its contents.

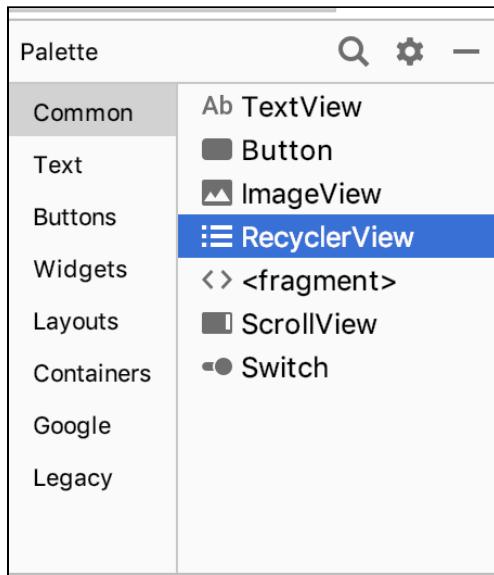
Currently, you pass in a list from `MainActivity.kt` via an Intent and set the title of the Activity to the name of the list. That's cool, but this Activity needs to do more. For starters, it needs to let a user view all of the items in the list, as well as add new items.

You can accomplish the first task — viewing all of the items — using a RecyclerView.

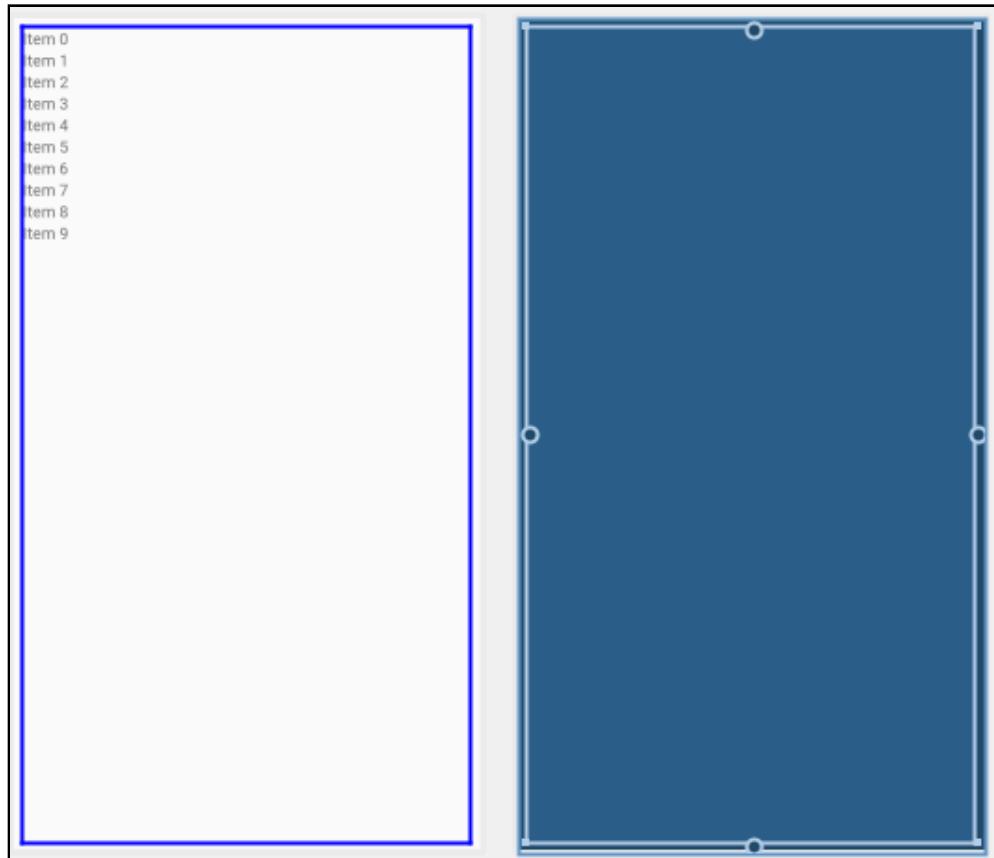
Open `res\layout\activity_list_detail.xml` from the `layout` folder, and select the **Design** tab in the Layout window if it's not already selected.



In the Palette window, select the **Common** option from the left-hand list. You'll see the RecyclerView available for selection in the right-hand list.



Click and drag the **RecyclerView** to the whitespace in the Layout shown on the right of the Layout Window.



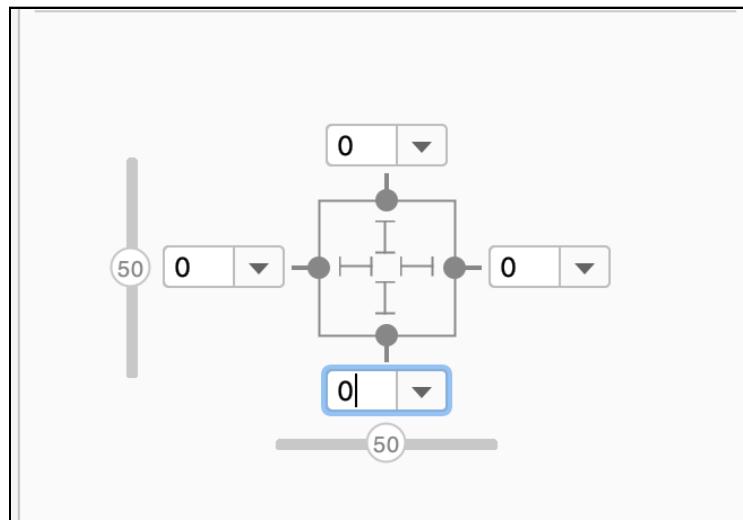
With the RecyclerView added, you need to give it an ID and some dimensions. In the **Attributes** window, change the **ID** of the RecyclerView to **list_items_recyclerview**.



Next, update the **layout_width** and **layout_height** to **match_constraint**. This ensures the RecyclerView adheres to the constraints you're about to set, and that it takes up the entire screen.



Click the four + buttons around the square to add constraints to the RecyclerView. You'll see this below the ID text field. Change the **margins** for each constraint to **0**.



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

Coding the RecyclerView

Open **ListDetailActivity.kt**, and at the top of the class, add a property to hold a reference to the RecyclerView:

```
lateinit var listItemsRecyclerView : RecyclerView
```

If offered a choice, select the `android.support.v7.widget.RecyclerView` version of the RecyclerView component.

Update `onCreate(savedInstanceState: Bundle?)` to connect the RecyclerView from the Layout to the Activity, and then create an Adapter and Layout Manager for the RecyclerView:

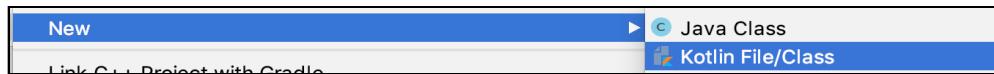
```
// 1
listItemsRecyclerView =
    findViewById(R.id.list_items_recyclerview)
// 2
listItemsRecyclerView.adapter = ListItemsRecyclerViewAdapter(list)
// 3
listItemsRecyclerView.layoutManager = LinearLayoutManager(this)
```

As a recap, here's how it works:

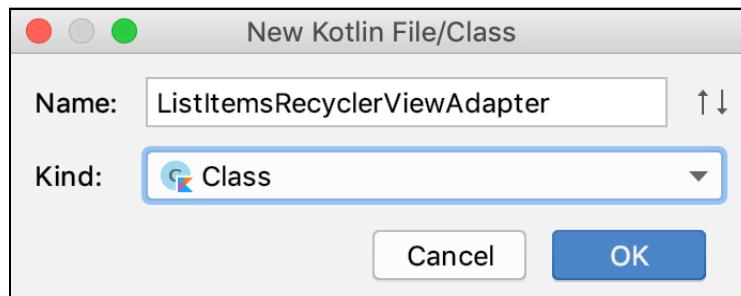
1. Find the RecyclerView in the Activity Layout and assign it to the local variable.
2. Assign the RecyclerView an Adapter, and pass in the list. It needs to know about the list so it can tell the RecyclerView what tasks to show. You'll create the Adapter shortly, so ignore the **Unresolved reference** for now.
3. Assign the RecyclerView a Layout Manager that uses a LinearLayoutManager to handle the presentation.

You're ready to create a new Adapter for the RecyclerView. In the Project navigator, right-click **com.raywenderlich.listmaker**.

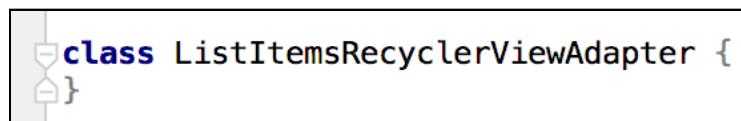
In the popup that appears, navigate to **New ▶ Kotlin File ▶ Class**.



Name the class **ListItemsRecyclerViewAdapter**, and ensure **Kind** is set to **Class**. When you're ready, click **OK**.



Android Studio creates the new Kotlin class and opens it.



Before using this new class, you need to make a few adjustments.

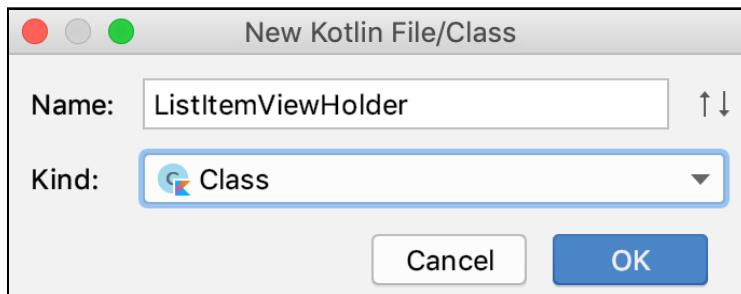
In **ListDetailActivity.kt**, you need to pass a list to the RecyclerView Adapter, and to use this list, the Adapter needs a constructor that accepts a TaskList. You also need to make the class implement the `RecyclerView.Adapter<ViewHolder>` Interface so the Adapter can create ViewHolders for the RecyclerView and reuse them as necessary. Finally, you need to create a custom ViewHolder that you can use to show the tasks in the list.

First, update the class definition so it has a primary constructor that accepts a TaskList and have it conform to `RecyclerView.Adapter<ListViewHolder>`:

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

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

Create another Kotlin class. Set the name of the file to **ListViewHolder**, and set **Kind** to **Class**.



After Android Studio creates the class, update its definition so it has a primary constructor to pass in a View and make it implement the `RecyclerView.ViewHolder(itemView)` interface:

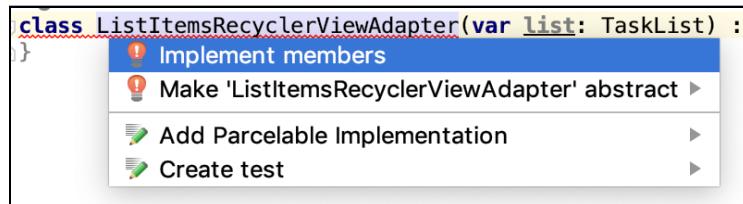
```
class ListViewHolder(itemView: View) :  
    RecyclerView.ViewHolder(itemView)
```

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

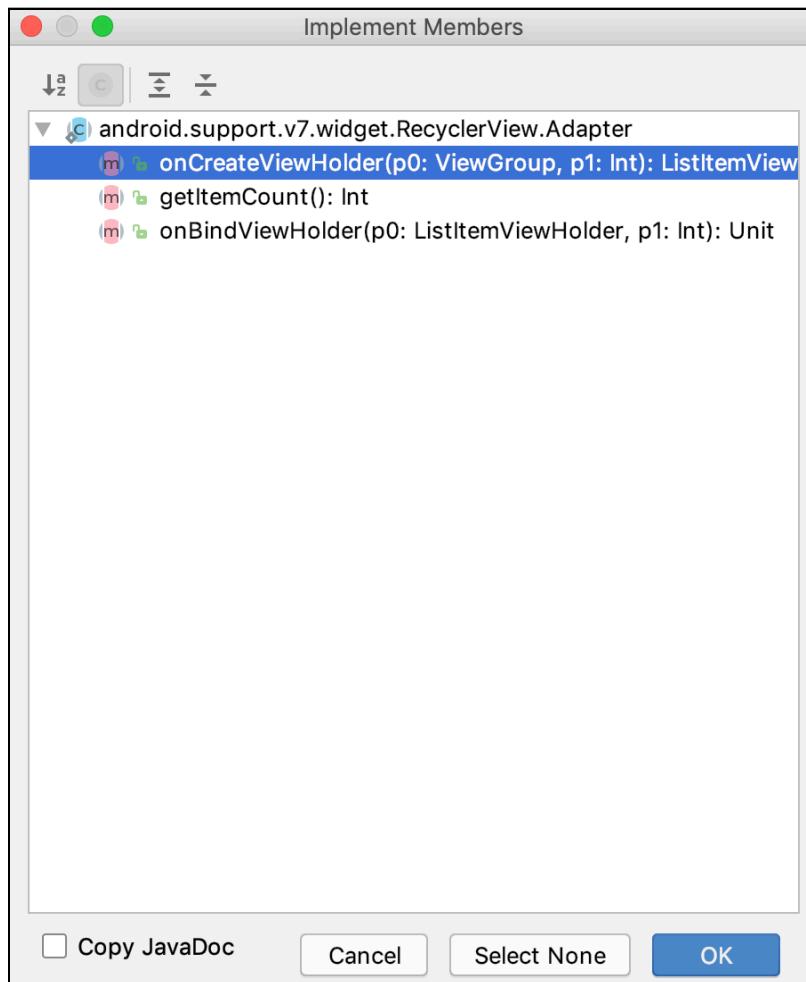
Adapting the Adapter

Open `ListItemsRecyclerViewAdapter.kt`.

This Adapter must implement methods required by `RecyclerView.Adapter` to ensure the RecyclerView knows how to present each task in the list. To get started quickly, there's a way to let Android Studio do most of the work for you. Click the class name (the part where the red squiggly line is) and press **Alt-Return**.

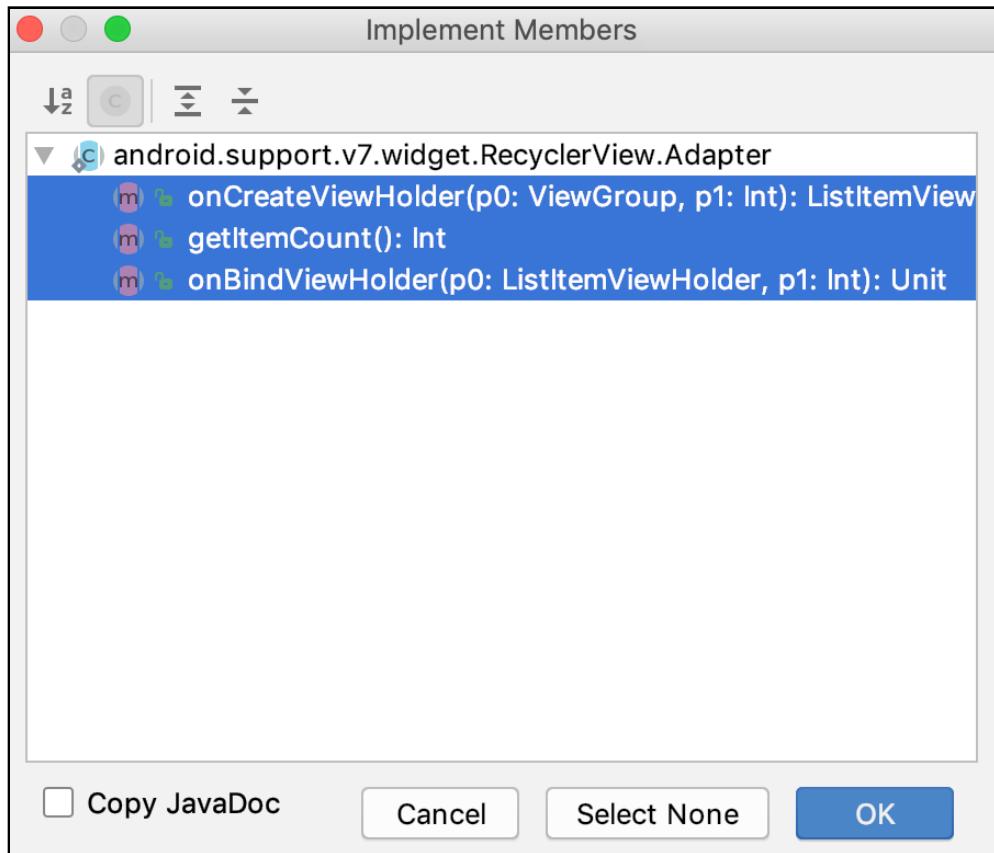


In the popup that appears, you'll see the first option highlighted is **Implement Members**. Press **Return** again, and Android Studio presents another window.



This window shows the methods you need to implement to conform to `RecyclerView.Adapter`, the Interface your class has implemented. You need to implement all of these methods, so hold down **Shift** and click the bottom-most method.

This highlights all of the methods in blue, which means you've selected all of the ones you want Android Studio to implement. To finish this set up, click **OK**.



With that, Android Studio automatically generates the chosen methods for you:

```
override fun onCreateViewHolder(p0: ViewGroup, p1: Int): ListItemViewHolder {
    TODO(reason: "not implemented") //To change body of created functions use File | Settings | File Templates.
}

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

override fun onBindViewHolder(p0: ListItemViewHolder, p1: Int) {
    TODO(reason: "not implemented") //To change body of created functions use File | Settings | File Templates.
}
```

All you need to do is write the logic behind each method.

Begin with `getItemCount()`. This method tells the RecyclerView how many items to display. You want it to show all of the tasks in your list, so update the method so it returns the number of tasks it contains:

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

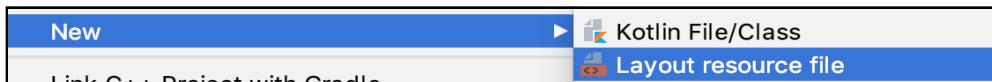
OK! Time to move onto creating the ViewHolder in `onCreateViewHolder()`.

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

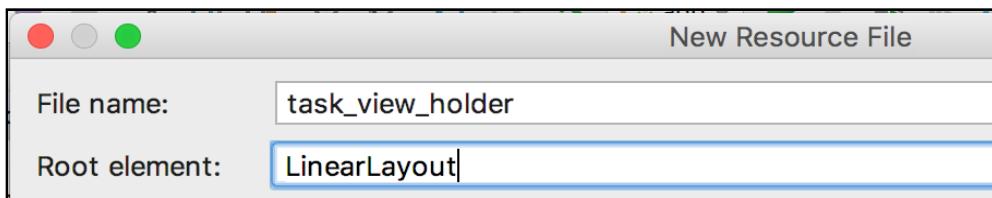
```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):  
    ListItemViewHolder {  
  
    val view = LayoutInflater.from(parent.context)  
        .inflate(R.layout.task_view_holder, parent, false)  
    return ListItemViewHolder(view)  
}
```

Don't worry about the **Unresolved reference** for `task_view_holder`; you'll create that next.

With the ViewHolder ready for use, you now need a Layout for the `LayoutInflater` to inflate. In the Project navigator, to the left of Android Studio, right-click `layout` inside `res`. Move the cursor to `new`, and click **Layout Resource File**.

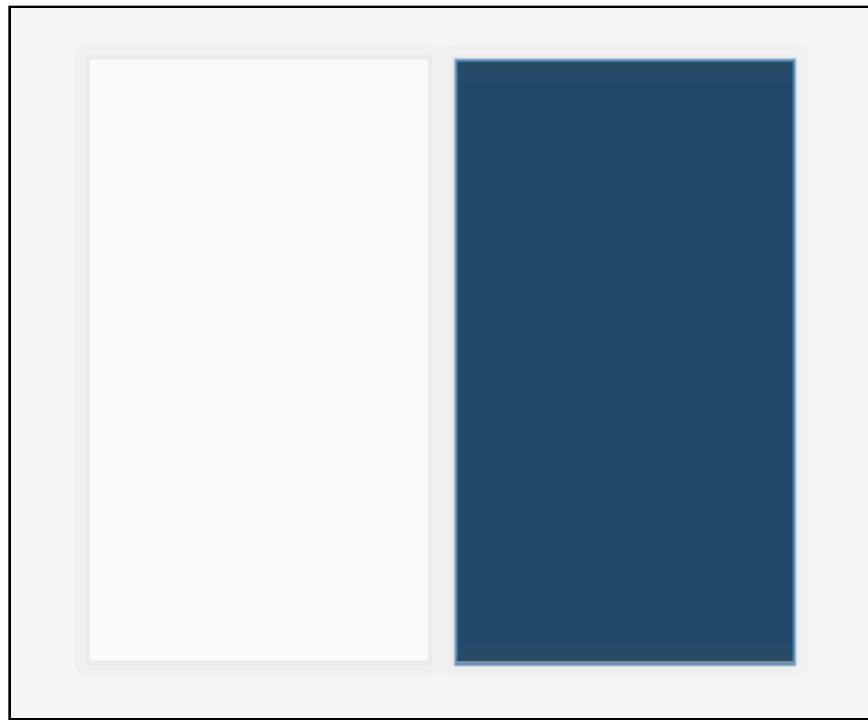


The New Resource File window appears. Enter the file name as `task_view_holder`. Change the **Root Element** below the name from `android.support.constraint.ConstraintLayout` to `LinearLayout`.



This sets the Layout to use a **LinearLayout**. A LinearLayout allows you to stack Views in a vertical or horizontal direction. For simple Views like `task_view_holder`, a LinearLayout is easier to use than a ConstraintLayout.

Click **OK**, and let Android Studio create the new Layout.



Before you continue, you need to set the **LinearLayout** so it's only as tall as the content within it; Otherwise, every row will get set to the size of its entire parent - in this case, the RecyclerView, which takes up the whole screen!

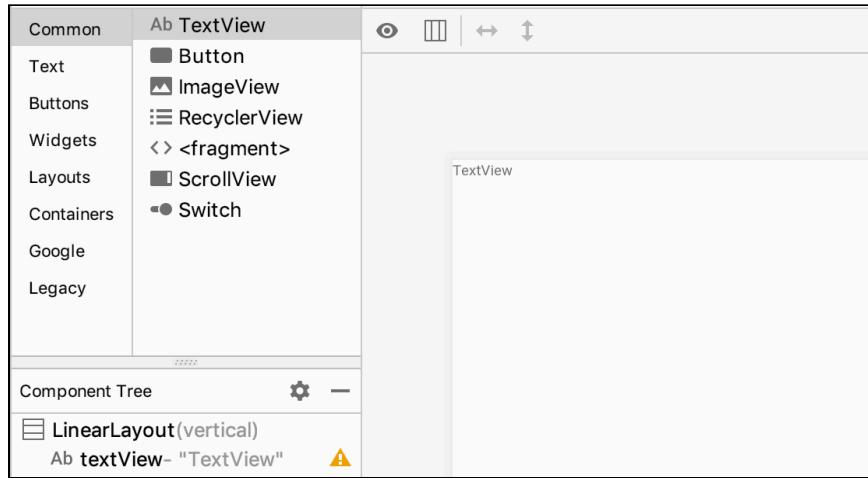
Select **LinearLayout** in the **Component Tree** window. In the Attributes window, set the **layout_height** to **wrap_content**:



Your Layout will now only be as large as whatever is inside of it. At this point, you might see its height shrink down to nothing in the Design tab as there's nothing in it yet.

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

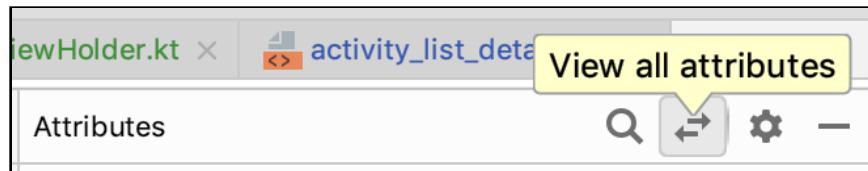
In the Palette window, click **Common**, and then drag a **TextView** into the **LinearLayout** via the Component Tree.



In the Attributes window, with the TextView selected, change the **ID** to **textview_task**, and set the **layout_width** and **layout_height** to **wrap_content**.



There's one final tweak needed here: margin spacings. To add margin spacings, you need to get to the larger list of attributes for the TextView. Click **View all attributes** at the top of the Attributes window.



The Attributes window slides across to reveal the entire list of attributes available for the TextView. Click the arrow next to **Layout_Margin** to expand the list. In the **left** and **top** text fields, enter **16dp**.



With the Layout ready, it's time to get back to coding!

Visualizing the ViewHolder

You now have a Layout for the ViewHolder, but you need to reference the TextView in the Layout in your code. Open **ListItemViewHolder.kt** and add the following line between the class brackets:

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

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

Open **ListItemsRecyclerAdapter.kt**, and in `onBindViewHolder()`, update it to bind to a specific task from the list depending on the position of the ViewHolder:

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

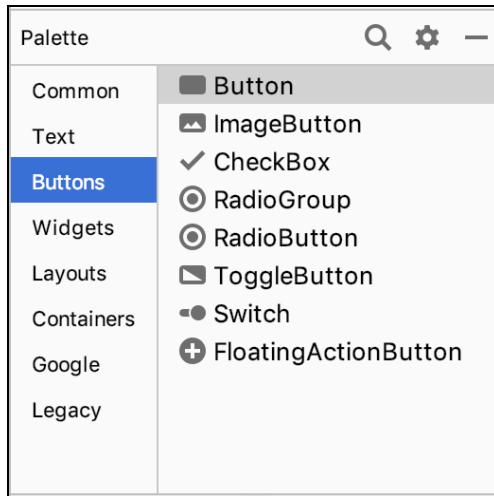
Excellent! You just finished hooking up the Adapter, and the RecyclerView will now run as expected.

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



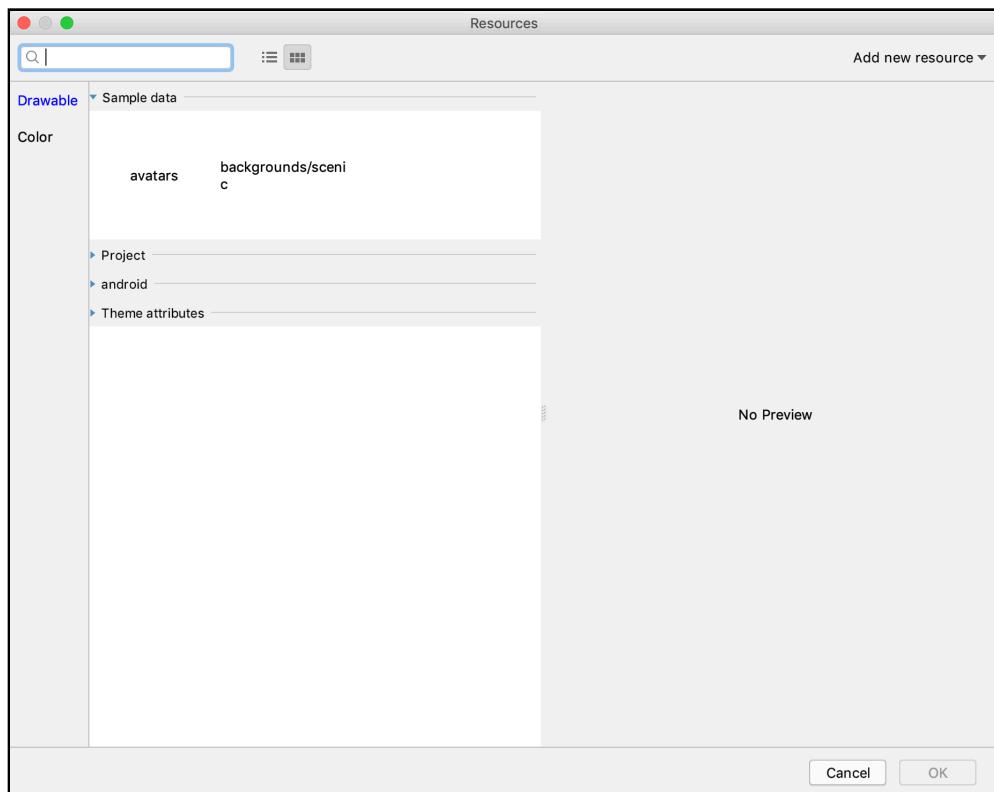
Currently, there's no way to add tasks to the lists. Time to fix that!

Open `res\layout\activity_list_detail.xml`, and ensure the **Design** tab is selected. In the palette, select **Buttons** and grab a **FloatingActionButton**.

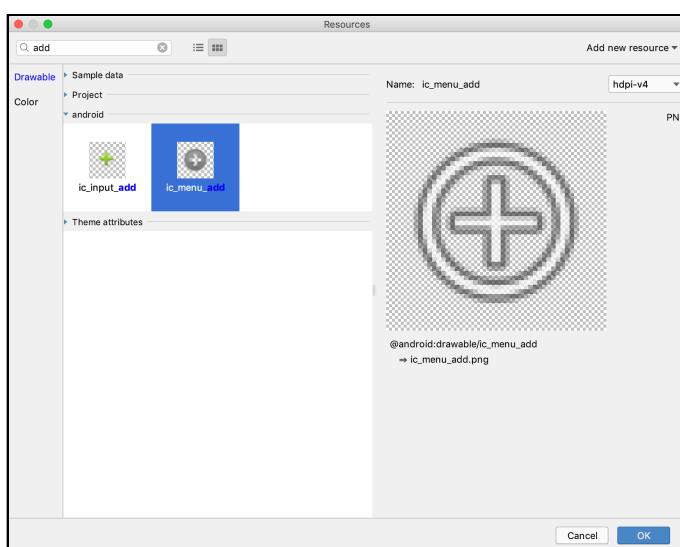


Drag the FAB into the Layout. Be careful to drag it into the **ConstraintLayout** using the Component Tree window and not the RecyclerView. Otherwise, it won't show up.

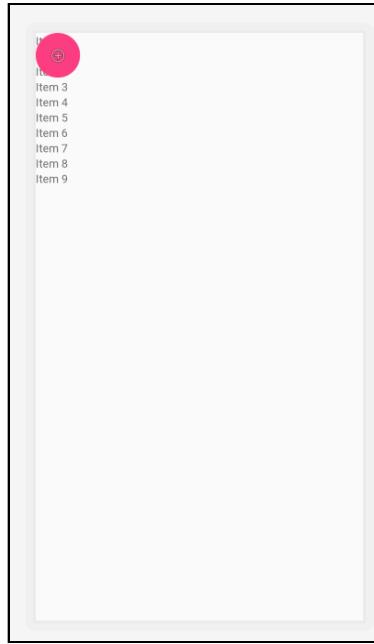
Once you drop it in, a window appears asking you to select a resource for the action button.



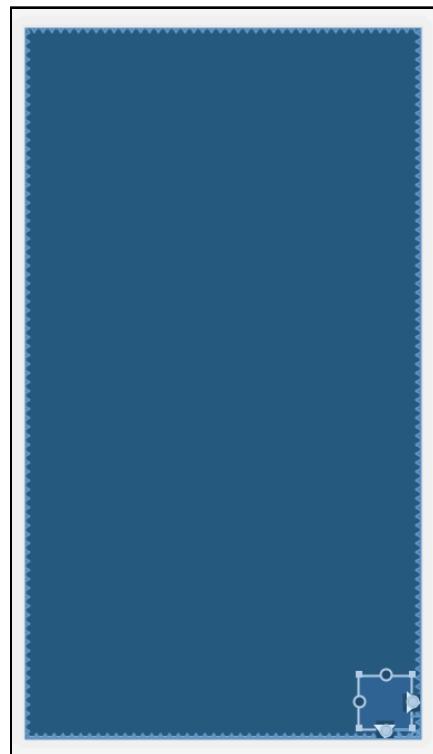
In the Search bar, type **add** to filter the list of resources available. Click the familiar-looking **ic_menu_add** resource and click **OK**.



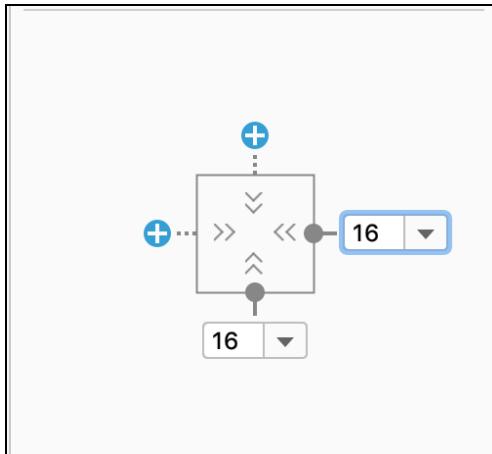
The button now appears in the Layout, ready for you to position.



Select the button. Now, in the Blueprint view (the blue screen next to the design preview), 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 of the FAB to **add_task_button**. In the Constraints view, make sure the **bottom** and **right constraints** are added. If they're not, add them by using **+**. Set the **margin** of the **bottom** and **right constraints** to **16**.



You're ready to use the new action button to add tasks to your list. Open **ListDetailActivity.kt** and add a new property to the top of the class to hold the reference for the new button:

```
lateinit var addTaskButton: FloatingActionButton
```

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

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

In the click listener, you call a method that prompts the user for the task to be added to the list. You'll create that method next. Below `onCreate()`, add the following:

```
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
            ListItemsRecyclerAdapter
            recyclerAdapter.notifyItemInserted(list.tasks.size-1)
    }
}
```

```
    //5
    dialog.dismiss()
}
//6
.create()
.show()
}
```

The code should look familiar to you, as it's similar to `showCreateListDialog()`, which you created in **MainActivity.kt**. It's a bit shorter, but here's what's happening:

1. Create an `EditText` so you can receive text input from the user.
2. Create an `AlertDialogBuilder` and use 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 for the next method. Here, when any method is called on the Builder, it returns the builder instance, modified with whatever you just added.
3. In the Positive Button's click listener, you access the `EditText` to grab the text input and create a task from the input.
4. Still in the click listener, you notify the `ListItemsRecyclerViewAdapter` that a new item was added. This gives the Adapter a chance to check its datasource (the list) so it can inform the `RecyclerView` to create any new rows with the new information.
5. Once the `RecyclerAdapter` updates, you close the dialog by dismissing it.
6. Back outside the click listener, you continue to use method chaining to create and then show the `AlertDialog` without ever needing to have the `AlertDialogBuilder` as a separate variable.

Android Studio will let you know (in its angry red text) that you're missing some strings used for the title and positive button of the dialog.

Open `res\values\strings.xml` and add the following new string elements 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 are shown in the app when the user adds a new task.

Finally, you need to save any new tasks that were added to the list.

Remember that `ListDataManager` you created when you first began to save lists? You'll use that again to update the saved list with any new tasks it might have.

Getting the list back

ListDataManager is declared and used in **MainActivity.kt**. To save the list with the newly added task, you *could* also declare it in **ListDetailActivity.kt** and save the list anytime the user adds a new task.

That would work, but it means you have two separate places in your app where data is saved, which gives you double the places where bugs could occur. To avoid that, you'll pass the list back to the **MainActivity** and use the original list data manager.

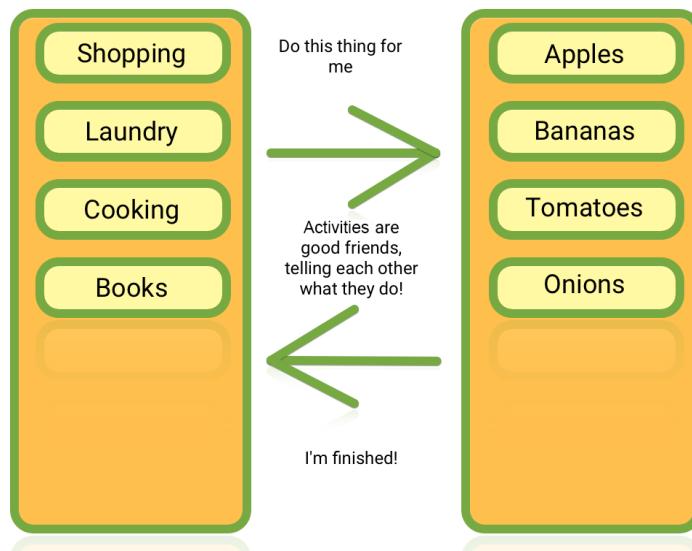
Open **MainActivity.kt** and edit `showListDetail()` so it looks like this:

```
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. Notice that `startActivity()` was changed to `startActivityForResult()`. While this change may seem small, the difference is very important. This line starts the detail Activity and adds the expectation that **MainActivity.kt** will hear back from **ListDetailActivity.kt** once it finishes and removes itself from the screen.

Think of it as asking someone to do something for you, and then reporting back with the results when they're finished. That's what's going on here: You want to hear back about that list you're passing to **ListDetailActivity.kt**.



You may notice an additional parameter gets passed into `startActivityForResult()`. The second parameter is a request code that lets you know which result you're dealing with. Because you can be dealing with multiple Activities that pass back multiple results, having a unique way of identifying results is handy.

Add the request code in the companion object at the bottom of `MainActivity.kt`:

```
companion object {
    const val INTENT_LIST_KEY = "list"
    const val LIST_DETAIL_REQUEST_CODE = 123
}
```

Next, you need to deal with the returned result. For that, you need to override a new method in `MainActivity` named `onActivityResult`. This method allows the Activity to receive the result of any Activities it starts. In this case, it looks for the result that `ListDetailActivity.kt` provides once it finishes adding tasks to a list:

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data:
    Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    // 1
    if (requestCode == LIST_DETAIL_REQUEST_CODE && resultCode ==
        Activity.RESULT_OK) {
        // 2
        data?.let {
            // 3
            listDataManager.saveList(data.getParcelableExtra(INTENT_LIST_KEY))
            updateLists()
        }
    }
}
```

Going through this method step-by-step:

1. You first check 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. You also check that the `resultCode` is `RESULT_OK` because there are times where a user might cancel an action.
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 and it contains null, so it's good to first make sure you have something to deal with.
3. Once you confirm there's data here, you save the list to the list data manager and then call `updateLists()`, which you'll create shortly.

Note: You may have noticed the `data?.let` block in the code snippet. The `.let` function is a shorthand method available in Kotlin. It allows you to only execute a block of code if the variable `.let` is used on is not 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 falls under the **Null Safety** paradigm of Kotlin, which you can read more about here: <https://kotlinlang.org/docs/reference/null-safety.html>.

Below `onActivityResult()`, add the `updateLists()` method:

```
private fun updateLists() {
    val lists = listDataManager.readLists()
    listsRecyclerView.adapter =
        ListSelectionRecyclerViewAdapter(lists, this)
}
```

This code reads the saved lists again so the RecyclerView is aware of any new tasks added to a List.

Finally, open **ListDetailActivity.kt**, and at the bottom of the class, 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()
}
```

`onBackPressed()` gives you a chance to run 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 you put that into an Intent that is passed back to **MainActivity.kt**. Finally, you set the result to `RESULT_OK`, informing the Activity that everything happened according to plan.

It's time to test the app again. Click **Run App** at the top of Android Studio and select your device. Create a list if necessary, or select an existing list. Once inside the list, add a new task.



Tap the back button, then click into the list where you added a task, and you'll see the newly added task. Well done!

Where to go from here?

You reused a lot of your knowledge in this chapter and picked up new tricks to reuse code in your apps in a clean way while also learning how to pass data between Activities. You now have a fully functioning list app.

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 includes across devices, adding new features to your app is easy. However, when it comes to coding an appealing user interface that adapts across all of these devices with varying screen sizes, things can get tricky!

In this chapter, you'll adapt ListMaker to make full use of the additional screen space a tablet provides. Along the way, you'll also learn:

- What Fragments are and how they work with Activities.
- How to split Activities into Fragments.
- How to provide different Layout files for your app depending on the device's screen size.

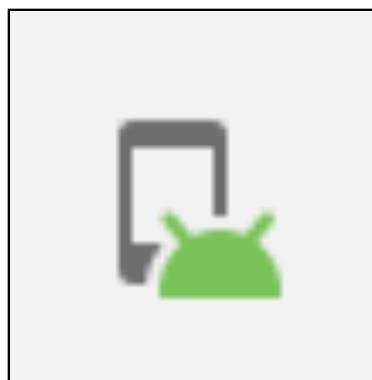
Getting started

If you're following along with your own app, open it and keep using it with this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **ListMaker** app inside the **starter** folder.

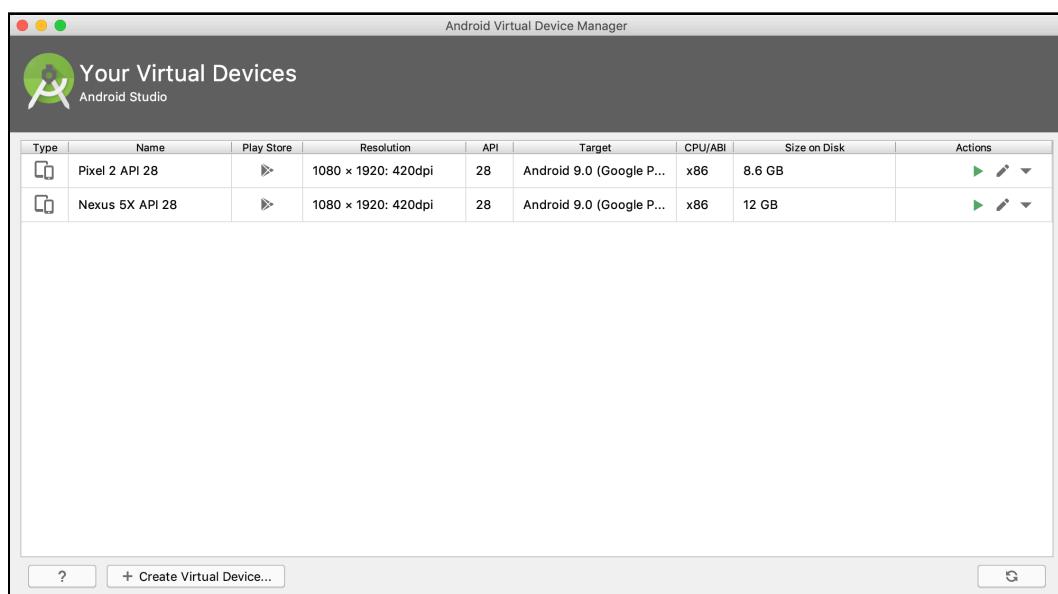
The first time you open the project, Android Studio takes a few minutes to set up your environment and update its dependencies.

You'll start things off by creating a device that emulates a tablet. If you have a physical tablet available, you can use that if you prefer.

With the ListMaker project open, click **Android Virtual Device Manager** along the top of Android Studio.



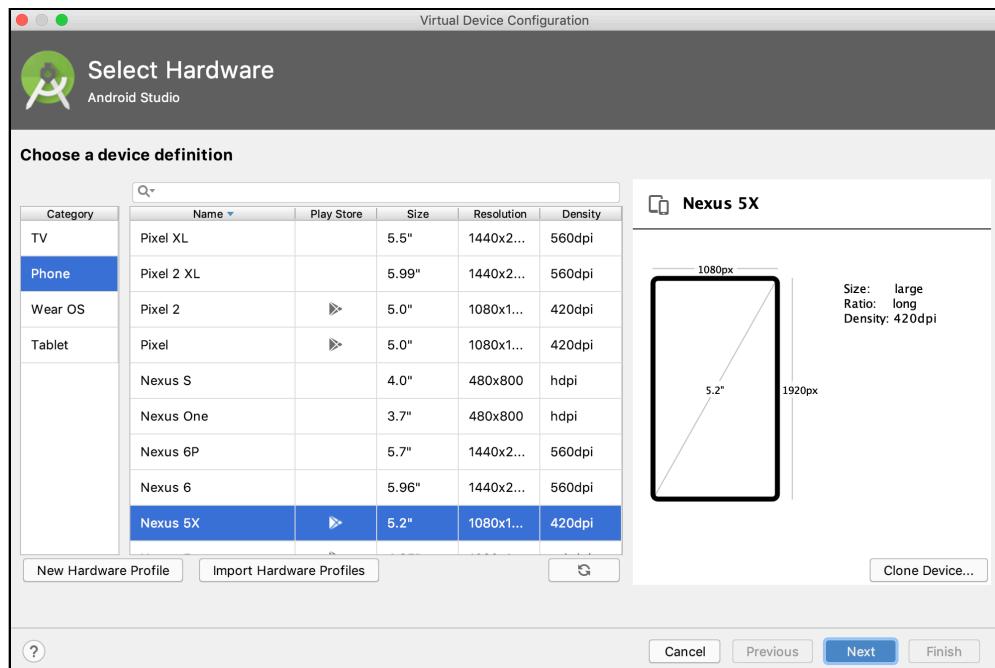
The AVD window pops up, showing you the emulators already available on your machine.



Click **Create Virtual Device** at the bottom of the window.

+ Create Virtual Device...

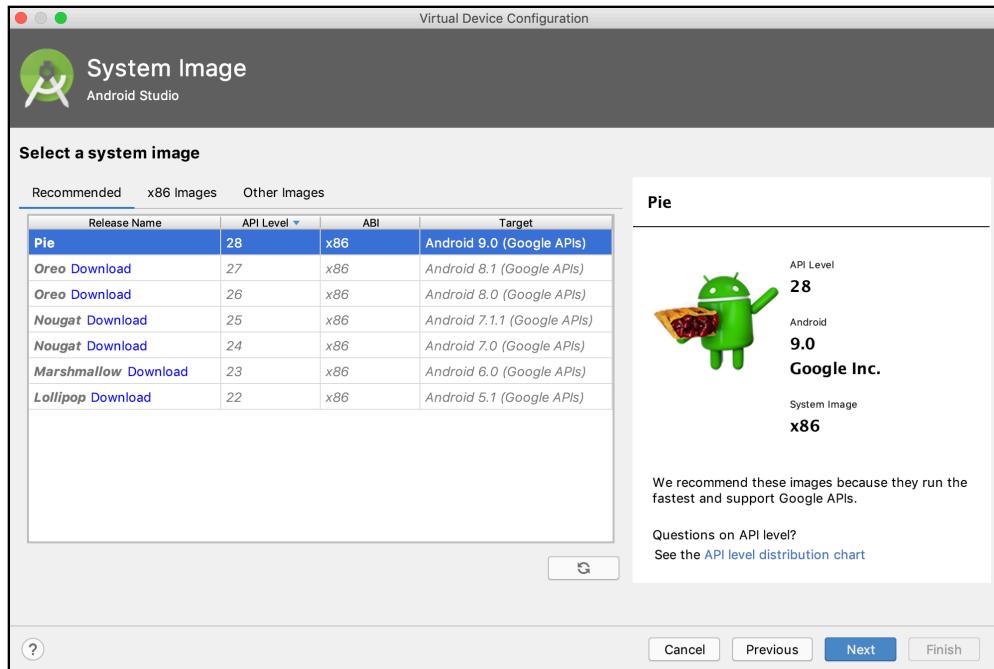
A new window pops up asking what hardware you want the virtual device to emulate.



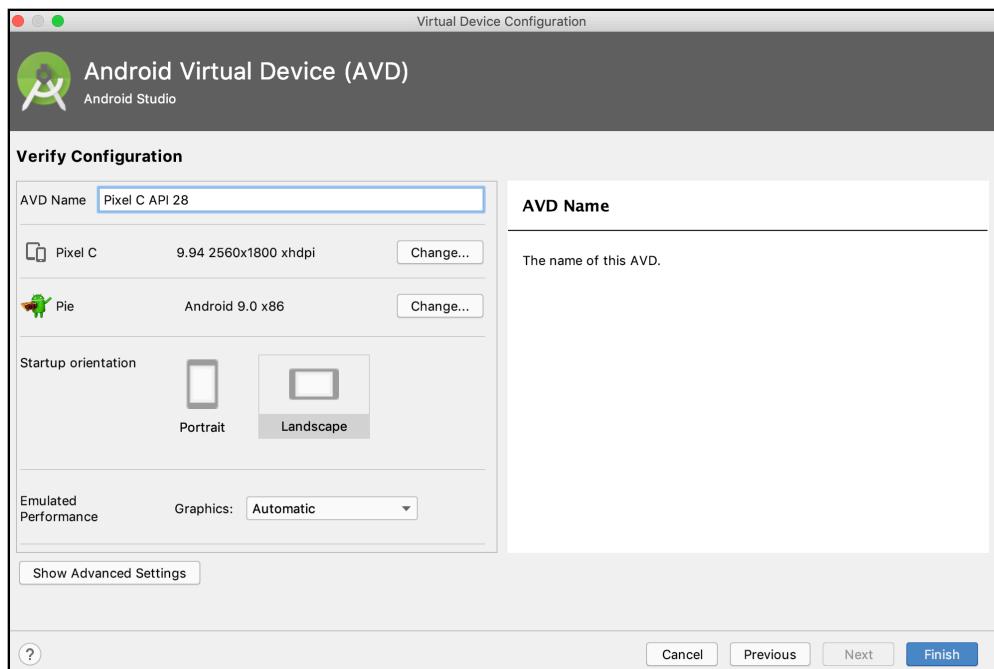
Select the **Tablet** category on the left. Notice the table in the middle of the window changes to offer a selection of tablets.

Name ▾	Play Store	Size	Resolution	Density
Pixel C		9.94"	2560x1...	xhdpi
Nexus 9		8.86"	2048x1...	xhdpi
Nexus 7 (2012)		7.0"	800x1280	tvdpi
Nexus 7		7.02"	1200x1...	xhdpi
Nexus 10		10.05"	2560x1...	xhdpi
7" WSVGA (Tablet)		7.0"	600x1024	mdpi
10.1" WXGA (Tablet)		10.1"	800x1280	mdpi

Select **Pixel C**. Then, in the bottom-right, click **Next**.



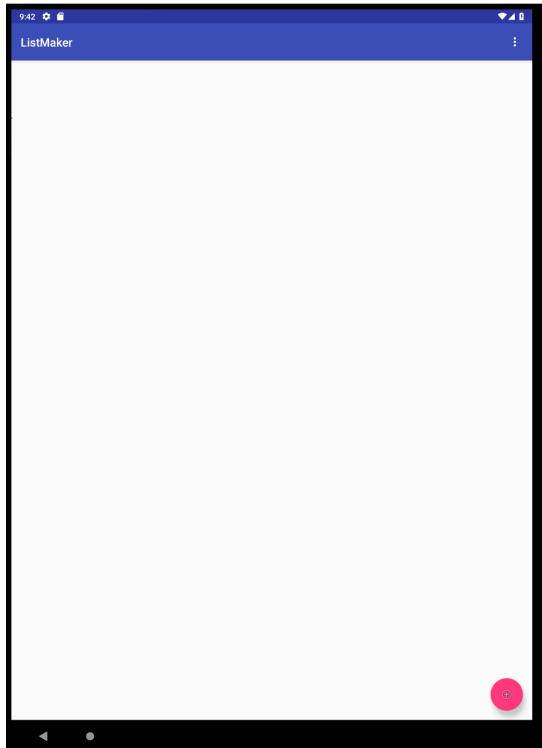
The next screen asks what version of Android you want the device to run. Select the highest API level and click **Next**:



The final screen displays the configuration for the device while allowing you to dive into advanced settings. Don't worry about changing anything here. Click **Finish** to complete setting up the 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 the app loads, you'll see that it looks exactly as it did on the phone.



Note: Just like we saw before in Chapter 4 "Debugging", for Android Pie devices, you may need to enable auto-rotate on your device or emulator if the screen doesn't rotate automatically.

To do this, swipe the notification drawer down to reveal the quick settings and ensure the auto-rotate button is colored green to signify it's enabled.

Create some lists and add tasks to each one, taking note of the extra real estate in the app.

Although the app works perfectly on a tablet, its design isn't optimized for the extra space available on the screen, so you need to consider how to make your app adapt to the size of a device's screen.

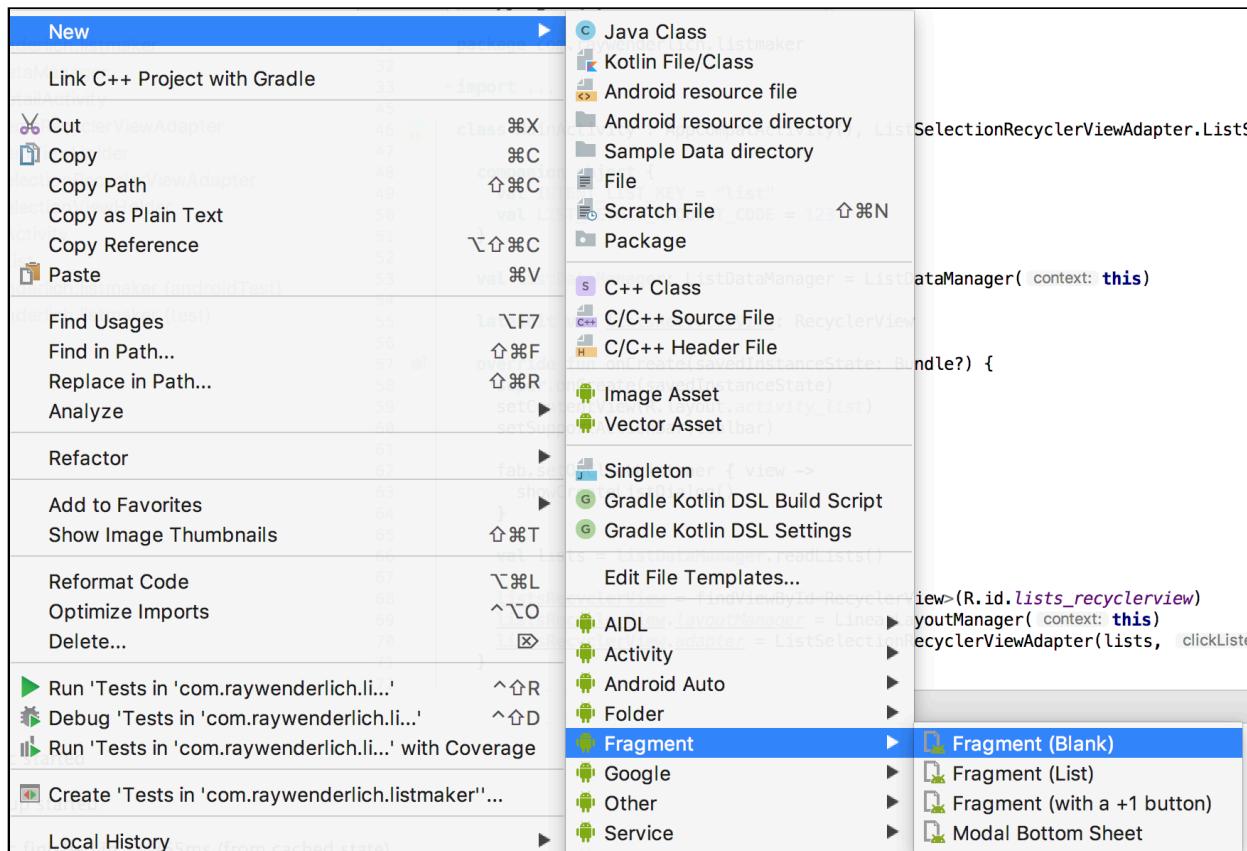
One solution is to split the screen in half so that one side can show the lists, while the other side can show the tasks that belong to each list.

While this solution does use the extra screen space available on the tablet, it doesn't work on small phone screens and will make the app unusable.

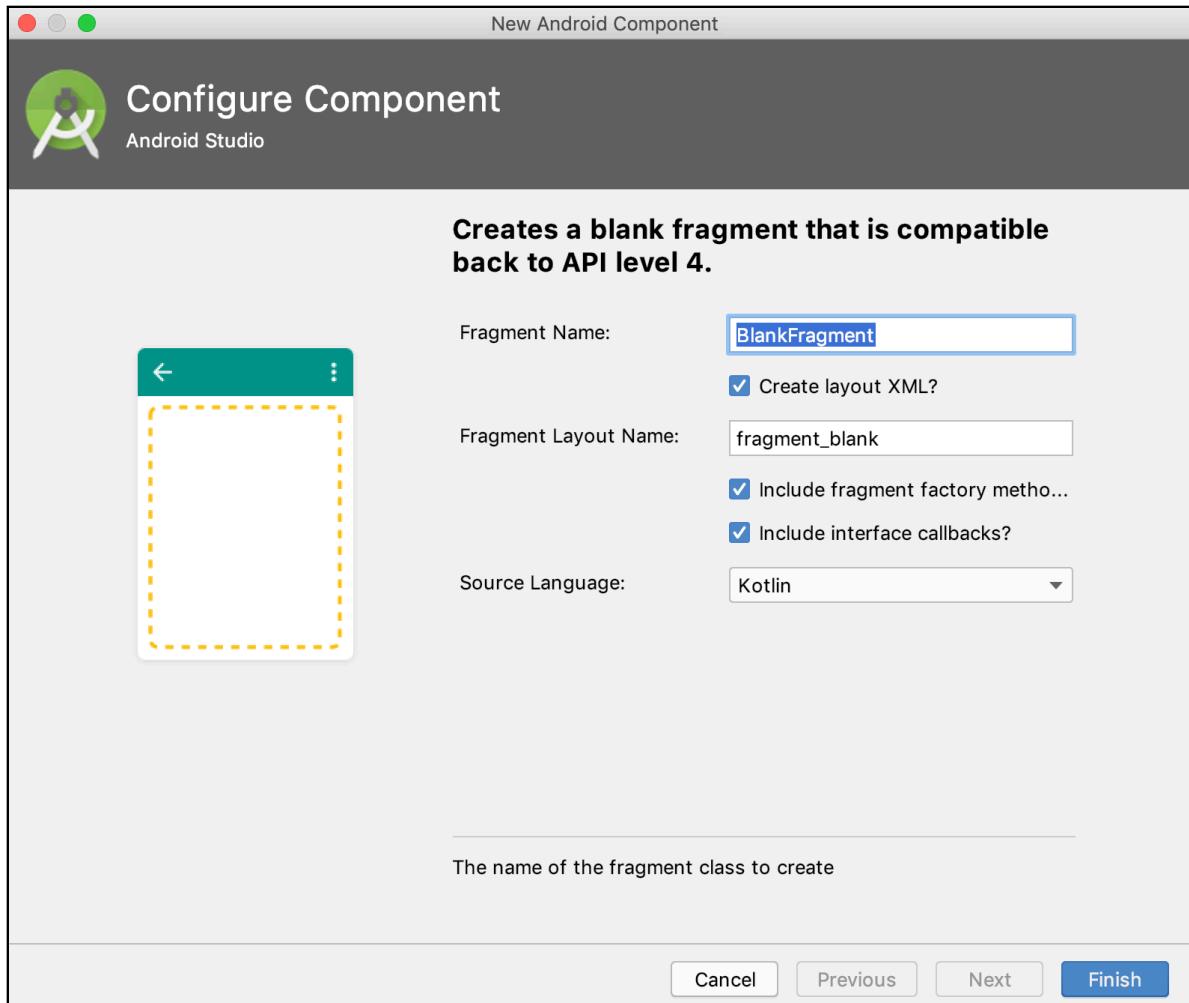
A better solution is to restructure ListMaker in a way that supports the best layout for both phones and tablets. This is where the concept of **Fragments** comes in.

Creating a Fragment

In the Project navigator, right-click `com.raywenderlich.listmaker`. In the selection dialog that appears, select **New > Fragment > Fragment (Blank)**.



Click **Fragment (Blank)** and Android Studio displays a new window.



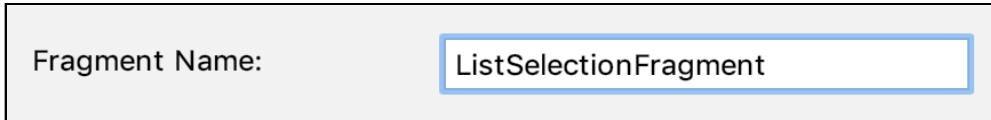
This window is dedicated to creating a new Fragment for your app. For now, don't worry too much about what a Fragment is; think of it as something similar to an Activity. What you're doing in this window is creating another screen for your app.

Let's go through the options available:

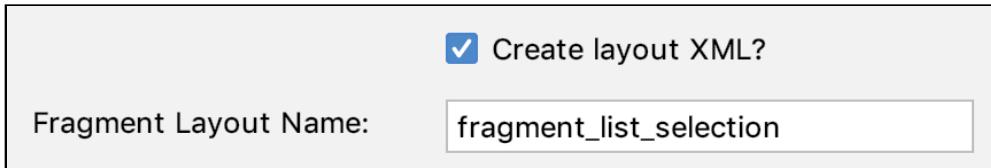


The **Fragment Name** allows you to name the Fragment, similar to the way you name an Activity.

Change the Fragment name to **ListSelectionFragment**.



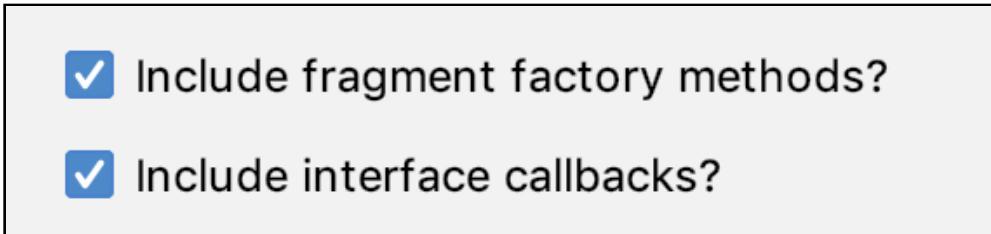
The next two options are **Create Layout XML** and **Fragment Layout Name**:



Again, similar to creating an Activity, Android Studio can create a Layout file for you. The **Create Layout XML** checkbox is checked by default, meaning Android Studio will create the Layout for the Fragment.

The **Fragment Layout Name** is used to name the Layout file for the Fragment. Android Studio has pre-populated this field with **fragment_list_selection**, which is based on the **Fragment Name** you entered. Leave this as it is.

The next three options are more complicated:



The first option, **Include fragment factory methods**, tells Android Studio to generate code to help create the Fragment in your app.

The second option, **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 Fragments a little later. The important thing to know now is that they are both required in this example.

The final option is **Source Language**:



This drop-down tells Android Studio what language to use to generate the code for your Fragment. Make sure **Kotlin** is selected and click the **Finish** button in the bottom-right of the window.

It's time to explain the mystery behind Fragments.

What is a Fragment?

A Fragment is 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. For instance, you can use this to your advantage to decide how many Fragments an Activity should show at runtime depending on the size of a screen.

If ListMaker is running on a tablet, you can 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 can show only one of the Fragments in an Activity and show the next Activity when a list selection is made.

Fragments give you a lot of power to help you use as much of the available screen space on a device 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, read 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 **ListSelectionFragment.kt**, so you can clean up the generated file to make it easier to understand. Change the file so it matches this:

```
class ListSelectionFragment : Fragment() {

    // 1
    private var listener: OnListItemFragmentInteractionListener? = null

    // 2
    override fun onAttach(context: Context) {
        super.onAttach(context)
        if (context is OnListItemFragmentInteractionListener) {
            listener = context
        } else {
            throw RuntimeException("$context must implement
OnListItemFragmentInteractionListener")
        }
    }

    // 3
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
    }

    // 4
    override fun onCreateView(inflater: LayoutInflater, container:
ViewGroup?,
                           savedInstanceState: Bundle?): View? {
        return inflater.inflate(R.layout.fragment_list_selection, container,
false)
    }

    // 5
    override fun onDetach() {
        super.onDetach()
        listener = null
    }

    interface OnListItemFragmentInteractionListener {
        fun onListItemClicked(list: TaskList)
    }

    // 6
    companion object {

        fun newInstance(): ListSelectionFragment {
            return ListSelectionFragment()
        }
    }
}
```

There's a lot of code here, all responsible for different things. Let's go through the file:

1. You define a private `OnListItemFragmentInteractionListener` variable to hold a reference to an object that implements the Fragment interface. The interface is also defined at the bottom of the class, requiring a single method to be implemented to inform objects that a list has been tapped. `MainActivity` will implement this interface.
2. `onAttach` is a lifecycle method run by a Fragment. Fragments have lifecycle methods available to override, similar to Activities. `onAttach` 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 `listener` if it implements the interface. This context is the `MainActivity` because it implements that interface.
3. 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.
4. Another lifecycle method, this one named `onCreateView()`. This is where the Fragment acquires the layout it wants to present within the Activity. Here, a Layout inflator is used to inflate the Layout and pass it back to the Fragment.
5. This is the final lifecycle method in the class that's called by a Fragment. `onDetach()` is called when a Fragment is no longer attached to an Activity, which happens when the Activity containing the Fragment is destroyed or the Fragment is removed. At this point within the method, `listener` is set to `null` as the Activity is no longer available.
6. You define a companion object here with a `newInstance()` method inside. This is used by any object that wants to create a new instance of the Fragment.

From Activity to Fragments

With the code cleaned up, the next job is to move parts of `MainActivity.kt` and its associated Layout to the new Fragment.

Remember that splitting your code into individual, isolated Fragments makes them reusable. It's essential that the Fragment needs nothing inside the Activity.

Open **MainActivity.kt** and remove the following properties:

```
val listDataManager: ListDataManager = ListDataManager(this)
lateinit var listsRecyclerView: RecyclerView
```

Move the properties to the top of **ListSelectionFragment.kt** with a slight modification:

```
lateinit var listDataManager: ListDataManager
lateinit var listsRecyclerView: RecyclerView
```

Notice that you're no longer initializing `listDataManager` inline since Fragment does not extend from Context. This means you'll have to initialize `listDataManager` at the earliest moment you get a Context, which is 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 must implement
OnListItemFragmentInteractionListener")
    }
}
```

Wonderful, your `ListDataManager` works exactly the same, except it now gets the Context via the Fragment. You'll notice errors in **MainActivity.kt** after you remove the last two variables. Time to fix that!

In `onCreate()` from **MainActivity.kt**, cut the following lines (you'll paste them shortly inside the Fragment):

```
val lists = listDataManager.readLists()

listsRecyclerView = findViewById(R.id.lists_recyclerview)
listsRecyclerView.layoutManager = LinearLayoutManager(this)
listsRecyclerView.adapter = ListSelectionRecyclerViewAdapter(lists, this)
```

You need to move these lines into a new lifecycle method in the Fragment named `onActivityCreated()`. This method runs when the Activity to which the Fragment is attached has finished running its lifecycle method `onCreate()`. This ensures that 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 to move from your Activity is the `ListSelectionRecyclerViewClickListener` interface implementation. This is the interface `ListSelectionRecyclerViewAdapter` provides to inform any interested objects that a list was selected.

Since `ListSelectionRecyclerViewAdapter` no longer exists in `MainActivity.kt`, you can move that to the new Fragment. Your Activity, however, still needs to be aware of the list click event. This is because only Activities should start other Activities. Fragments, being isolated views, should only inform Activities of any events it should handle.

You may recall that `ListSelectionFragment` provides an interface you can use to talk back to its Activity. You can use that. At the top of `MainActivity.kt`, change the class declaration as follows:

```
class MainActivity : AppCompatActivity(),
ListSelectionFragment.OnListItemFragmentInteractionListener {
```

The only change here is the Activity now implements the Fragments interface.

In `MainActivity.kt`, 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 has to be moved into the Fragment. The Fragment also needs to implement the interface so it can receive the list item click to pass up to the Activity.

In **ListSelectionFragment.kt**, update the class declaration so that it implements the `ListSelectionRecyclerViewClickListener` interface:

```
class ListSelectionFragment : Fragment(),
ListSelectionRecyclerViewAdapter.ListSelectionRecyclerViewClickListener {
```

Next, implement the interface method by adding `listItemClick()` to the `ListSelectionFragment` class:

```
override fun listItemClicked(list: TaskList) {
    listener?.onListItemClicked(list)
}
```

When the method receives an item click from the RecyclerView Adapter, it uses `listener` to inform the Activity that it's received an item click. This, in turn, allows the Activity to receive the list and to start a new Activity to show the list while keeping the 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 the Fragment is adding a list to the `ListDataManager`.

The data manager is now handled by your Fragment, but you still need to be able to use the data manager. Since it now resides in `ListSelectionFragment`, you need a reference to the Fragment in your Activity.

At the top of **MainActivity.kt**, add the following line:

```
private var listSelectionFragment: ListSelectionFragment =
ListSelectionFragment.newInstance()
```

This line creates a new instance of your Fragment when the Activity is created.

Next, In `showCreateListDialog()` of the Activity, update the positive button click listener to the following:

```
builder.setPositiveButton(positiveButtonTitle) { dialog, _ ->
    val list = TaskList(listTitleEditText.text.toString())
    listSelectionFragment.addList(list)

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

The change is subtle but important. What you've done is removed the lines that added the list to the Data Manager residing in the Activity, and replaced them with a method call to the Fragment. By doing so, the Fragment now adds the list to the Data Manager.

You'll see `listSelectionFragment.addList(list)` lighting up in red; that's because the Fragment doesn't yet know how to add a list.

In **ListSelectionFragment.kt**, add the missing method so it saves the list to the data manager and updates its RecyclerView:

```
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, the Fragment needs to handle this.

In **MainActivity.kt**, change `onActivityResult()` so that the parcelable extra is passed into a method provided by the 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 a new method so that the Fragment can save the updated state of the list received from `MainActivity` and update the RecyclerView:

```
fun saveList(list: TaskList) {  
    listDataManager.saveList(list)  
    updateLists()  
}
```

Finally, move `updateLists()` from `MainActivity` into the 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 the Activity to the Fragment. If you recall, the RecyclerView also resides in the Layout of this Activity, so you also need to adjust the Layouts so the RecyclerView is provided by the Fragment.

You also need to ensure the Activity Layout knows it needs to show the Fragment. This means you're going to have to dive into the not-quite-so pretty side of Layouts.

Open **content_main.xml** available in **res > layout**. If not already selected, select the **Text** tab in the bottom left-corner of the Layout editor.



The editor updates to look like something that resembles XML, rather than a user interface:

```
<?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$ScrollingViewBehavior"
    tools:context="com.raywenderlich.listmaker.MainActivity"
    tools:showIn="@layout/activity_main">

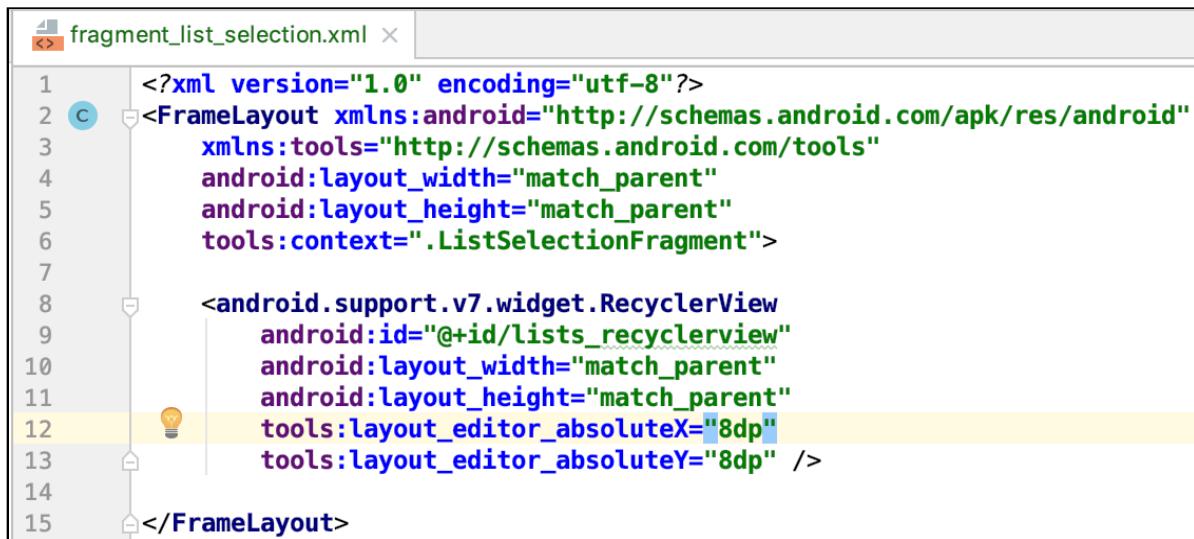
    <android.support.v7.widget.RecyclerView
        android:id="@+id/lists_recyclerview"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:layout_editor_absoluteX="8dp"
        tools:layout_editor_absoluteY="8dp" />
</android.support.constraint.ConstraintLayout>
```

Up until now, you've used the Design tab to create your Layouts. For this part, it's easier to work with the XML representation of the widget because you need to copy the Views across different files.

In **content_main.xml**, cut the entirety of the `android.support.v7.widget.RecyclerView` tag:

```
<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" />
```

Open the **fragment_list_selection.xml** Layout, select the **Text** tab if needed, and paste the `RecyclerView` over the generated `TextView`:



With the `RecyclerView` in its new layout, it's time to update the Activity Layout so it shows the Fragment. 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 the Activity and set the `layout_width` and `layout_height` to match the size of your Activity.

Open **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 the Activity so it grabs the reference to the FrameLayout via the ID you assigned it in the 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 the UI as flexible as possible for various screens.

The `SupportFragmentManager` makes use of a **FragmentTransaction**. Transactions are how you describe to the `SupportFragmentManager` what do with the Fragments.

To begin manipulation of any Fragments that use `SupportFragmentManager`, you first call `beginTransaction()` to begin a transaction.

Once the transaction starts, you then call `add()`, which tells `SupportFragmentManager` to add a Fragment into a container view that will hold the Fragment. To do this, `add()` takes two parameters: 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 creates. Once the transaction is defined, `commit()` informs the `SupportFragmentManager` that it should add the Fragment so it's visible in the Activity.

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.

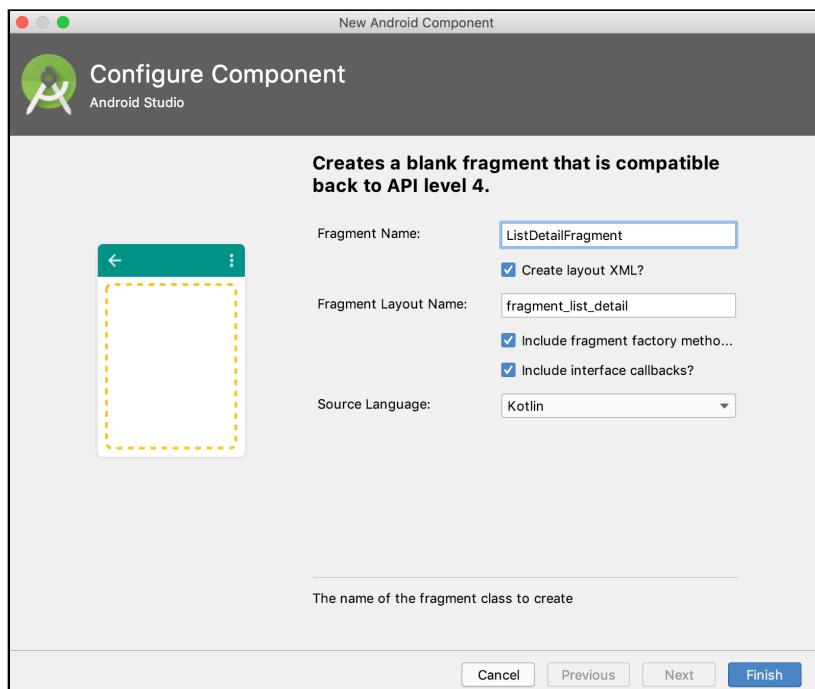
The app doesn't look any different at this point, but under the hood, you're now using an Activity that contains a Fragment. This is a good foundation to start making use of all that space on the tablet screen.

The next step is to replicate the `ListDetailActivity` screen into its own Fragment.

Creating your next Fragment

Right-click **com.raywenderlich.listmaker** in the Project navigator and select **New ▶ Fragment ▶ Fragment (Blank)**.

The Create Fragment window you used earlier pops up. Change the Fragment name to **ListDetailFragment** and click **Finish** in the bottom-right.



Android Studio creates a **ListDetailFragment.kt** and a **fragment_list_detail.xml** Layout file for the Fragment. Clean up the Fragment of any generated code that isn't needed. The final **ListDetailFragment.kt** should look like this:

```
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 {
        private const val ARG_LIST = "list"
    }
}
```

```
    fun newInstance(list: TaskList): ListDetailFragment {
        val fragment = ListDetailFragment()
        val args = Bundle()
        args.putParcelable(ARG_LIST, list)
        fragment.arguments = args
        return fragment
    }
}
```

The main change here to the original code is the bundle argument passed in via `newInstance()`. It now expects a `TaskList` to be passed in since this Fragment is responsible for showing your list.

You also define an `ARG_LIST` constant as a top-level object to use as the key to put into the `Bundle` object for the Fragment and also retrieve the `TaskList` from the `Bundle` object when the Fragment is created.

Next, you need to copy some of the properties in **ListDetailActivity.kt** to the new Fragment. From the top of the Activity, copy the following lines to the top of **ListDetailFragment.kt** — be careful not to delete these as you need them later:

```
lateinit var list: TaskList
lateinit var listItemsRecyclerView: RecyclerView
```

Update `onCreate()` in **ListDetailFragment.kt** so it grabs the list from the bundle passed in, if it exists:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    arguments?.let {
        list = it.getParcelable(MainActivity.INTENT_LIST_KEY)!!
    }
}
```

Change `onCreateView()` to set up the `RecyclerView` via the ID in the Layout and initialize the `RecyclerView` Adapter and LayoutManager:

```
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 = ListItemsRecyclerAdapter(list)
    listItemsRecyclerView.layoutManager = LinearLayoutManager(context)
}

return view
}
```

Finally, add a method named `addTask` to the Fragment. You'll use this method later to instruct the Fragment to add tasks to the list:

```
fun addTask(item: String) {
    list.tasks.add(item)

    val listRecyclerAdapter = listItemsRecyclerView.adapter as
        ListItemsRecyclerAdapter
    listRecyclerAdapter.list = list
    listRecyclerAdapter.notifyDataSetChanged()
}
```

Now that the Fragment is using the `RecyclerView`, you also need to make sure that it exists in the Fragment Layout. Repeating your approach from the previous Activity, you need to copy the `RecyclerView` from the `ListDetailActivity` layout to the `ListDetailFragment` layout.

In `layout > activity_list_detail.xml`, with the `Text` editor open, copy the following lines from the Layout:

```
<android.support.v7.widget.RecyclerView
    android:id="@+id/list_items_reyclerview"
    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_toTopOf="parent" />
```

Open `layout > fragment_list_selection.xml`, still with the `Text` editor open, and paste the `RecyclerView` in between the `FrameLayout` tags, replacing the `TextView` that was auto-generated when you created the Fragment.

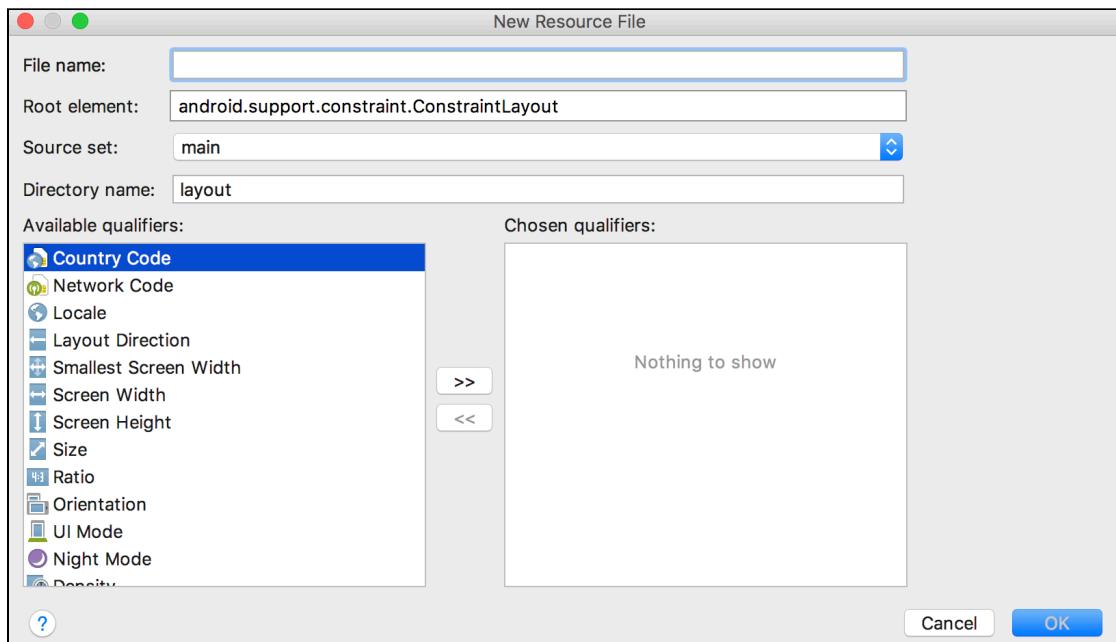
Remove the lines that begin with `app:layout_constraint` and update the `RecyclerViews` `layout_width` and `layout_height` to `match_parent`. You no longer need the constraint attributes now that the `RecyclerView` is sitting within a `FrameLayout`.

Bringing the Activity into action

So far, you've focused on transferring code over from the old Activities to the new Fragments. Remember though, that Fragments need to exist within an Activity to be of any use. The Activity also 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 1) intelligently show the new Fragments at the right time, 2) that it provides information to each Fragment, and 3) lets your app shift its appearance depending on the device.

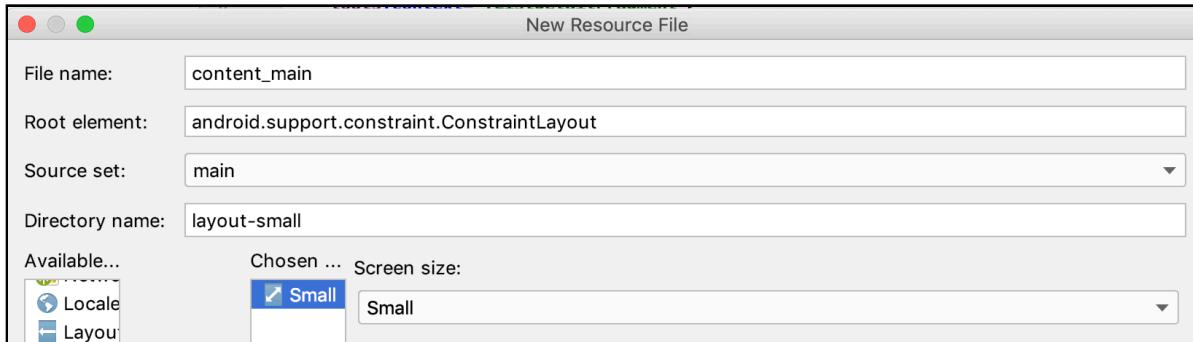
First, you need to create a Layout that works for a large screen. In the Project navigator, right-click **layout**, then select **New > Layout resource file**:



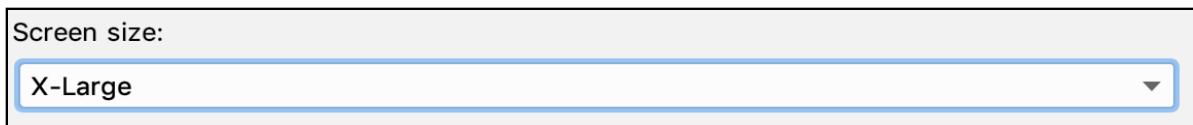
You're going to create a new Layout file as you've done before. There's a tiny difference here though: You're creating a version of the **content_main.xml** Layout that will only display on large screens.

This gives you the option to customize the UI for various sizes of screens. Android is even intelligent enough to automatically choose which Layout it should use as well. Very helpful!

For the **File name**, name your Layout **content_main**. Then, in the **Available qualifiers** list in the bottom-left of the window, select the **size** option and click **>>**.



From here, you can select various screen sizes to determine which sizes your Layout will use. You want the layout to be used by big screens (for example, tablets), so in the screen size dropdown, choose **X-Large**.



Click **OK** in the bottom-right and Android Studio creates the new Layout. Take a moment to look at the Project navigator to the left:



Android Studio now shows both the Layout files together in a drop-down, 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. In **xlarge/content_main.xml**, ensure the Text editor tab is selected at the bottom of the Layout editor window.

Replace the existing XML with the following code for the entire layout:

```
<?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_main">
<!-- 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_marginStart="0dp"
    android:layout_marginTop="8dp"
    android:layout_marginBottom="0dp"
    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_marginStart="0dp"
    android:layout_marginTop="0dp"
    android:layout_marginEnd="0dp"
    android:layout_marginBottom="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's going on with the Fragment and FrameLayout? When the larger layout is shown on a large screen, you need both the **ListSelectionFragment** and **ListDetailFragment** to appear to make use of the extra space.

The list selection Fragment is 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 newly selected list.

Open the original **content_main.xml** file and change the tag in between the ConstraintLayout tags so it only shows 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_marginStart="0dp"
    android:layout_marginTop="8dp"
    android:layout_marginBottom="0dp"
    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 to work out whether or not your app is running on a device with a large screen. You'll investigate this in detail later.

You now need to change **MainActivity.kt** so it can handle both Layouts, depending on the size of the screen for the device ListMaker is running on. The first thing you need is a way to know if you're using the larger layout.

At the top of **MainActivity.kt**, add a Boolean to track whether the larger Layout is in use. You also need a **ListDetailFragment** instance for use later, so create a property for it while you're here:

```
private var largeScreen = false
private var listFragment : ListDetailFragment? = null
```

In **onCreate()**, update the method to 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 the larger Layout is in use.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    setSupportActionBar(toolbar)

    listSelectionFragment =
        supportFragmentManager.findFragmentById(R.id.list_selection_fragment) as
            ListSelectionFragment

    fragmentContainer = findViewById<FrameLayout>(R.id.fragment_container)
    largeScreen = fragmentContainer != null
```

```
    fab.setOnClickListener {
        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 a `ListDetailFragment` is already showing, then it will automatically show the new Fragment 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)
        listFragment?.let {
            supportFragmentManager.beginTransaction()
                .replace(R.id.fragment_container, it,
                    getString(R.string.list_fragment_tag))
                .addToBackStack(null)
                .commit()
        }

        fab.setOnClickListener {
            showCreateTaskDialog()
        }
    }
}
```

Note that you have to unwrap `listFragment` using a `?.` let because the compiler has no way of knowing if it was reset between the assignment and trying to pass it into the fragment manager.

You're also using the `list_fragment_tag` string above to use with the `replace` transaction. This string is known as a **tag** and is used by the `supportFragmentManager` in case you want to reference it in the future. That string doesn't exist yet in `ListMaker`, so you need to add the string to `strings.xml`.

Open `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. Click the build button in the top tool bar, or from the menu *Build*, select *Make Project*.

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'll see a compilation error at this point.

Modify the positive button callback to pass the newly created task into the Fragment by adding the following method to **MainActivity.kt**:

```
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()
}
```

Next, override `onBackPressed()` so the Activity knows how to deal with the back button being pressed when using Fragments.

```
override fun onBackPressed() {
    super.onBackPressed()

    // 1
    title = resources.getString(R.string.app_name)

    // 2
    listFragment?.list?.let {
        listSelectionFragment.listDataManager.saveList(it)
    }

    // 3
    listFragment?.let {
        supportFragmentManager
            .beginTransaction()
            .remove(it)
            .commit()
        listFragment = null
    }

    // 4
    fab.setOnClickListener {
```

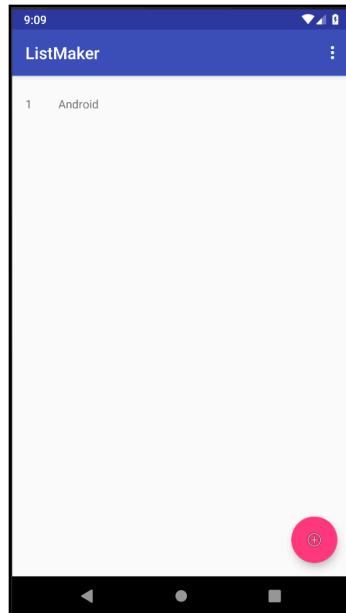
```
        showCreateListDialog()  
    }  
}
```

Going through the method:

1. When the back button is pressed and isn't focused on a List, you want the Activity to show the name of the app. You retrieve the name of the app from **Strings.xml** by using the `resources` property available through `MainActivity`.
2. 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.
3. 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. Again, you need to use a `? . let` since if you directly check the `listFragment` variable, it can get reset between the check and the Fragment manager transaction.
4. Update the FAB to create lists again.

It's worth mentioning that because `super.onBackPressed()` is being called, you're also deferring to any behavior from `MainActivity`. In this case, if a detail Fragment isn't visible to the user, the app will close and bring you to the Android home screen.

With that done, you're ready to see your hard work in action! Run your app on a **phone-sized** device and start playing around with it.



Run your app on a **tablet-sized** device, tab the FAB to create a list and then when the title changes to the name of the list. Tap the FAB again to add a task. You'll immediately see the difference.



Your app now displays two different screens, at the same time, making better use of the available space.

Where to go from here?

Fragments are a 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 need to use Fragments to ensure it provides the best experience for its users.

Chapter 12: Material Design

By Darryl Bayliss

When 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. You'll often hear this referred to by designers as **design language**.

This is such an important topic that Google created its own design language called **Material Design**. In this final chapter for Section II, you'll learn:

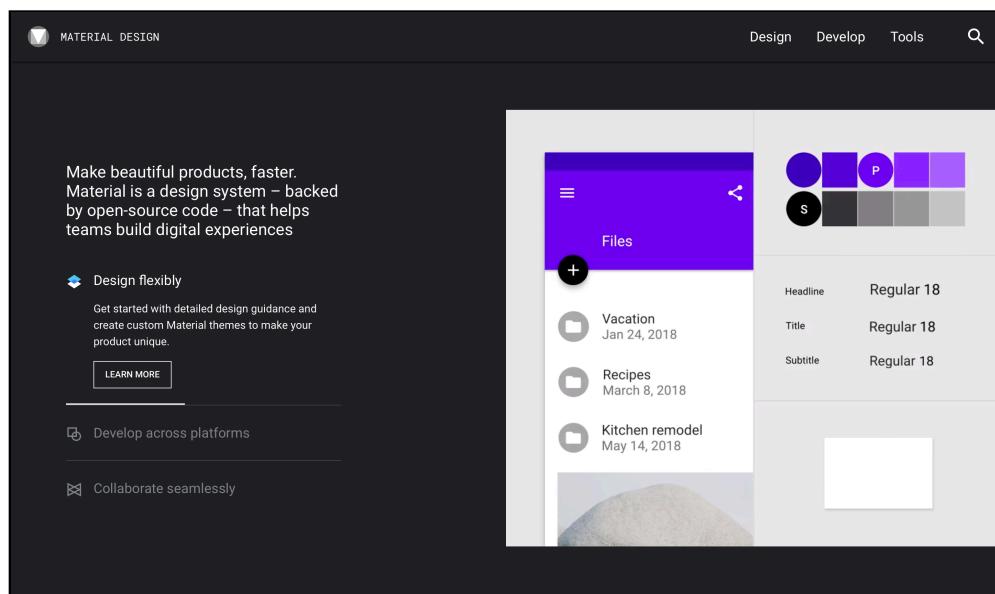
- What Material Design is.
- What resources are available to learn about Material Design.
- How to update ListMaker so that it adopts some Material Design principles.

What is Material Design?

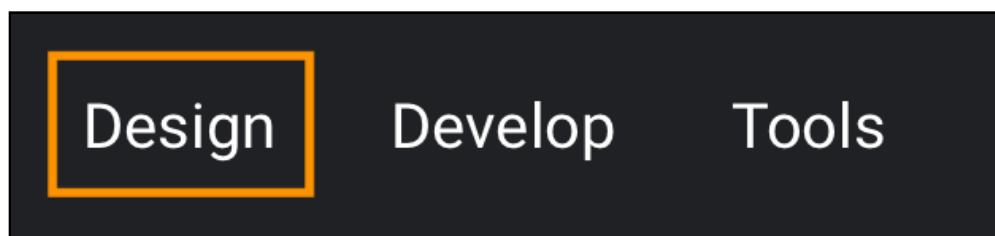
Material Design is a design language that aims to standardize how a user interacts with an app. This ranges from everything to button clicks, to widget presentation and positioning, even to animation within the app.

Before Material Design existed, there was no specific user interface to which an app was expected to adhere. This was a problem for users because different apps worked in different ways, which meant users had to figure out how each app was intended to work.

All of that changed with the introduction of Material Design. Android developers finally had a set of concise User Interface (UI) and User Experience (UX) guidelines for their apps to follow. In fact, Google is so invested in Material Design that it dedicated an entire site to it: <https://material.io>.



Open the site in your browser and look around. There's a lot to see, so once you're ready to proceed, click the **Design** button in the toolbar along the top of the page.



The Material Design guidelines are kept here.

The screenshot shows the Material Design website's homepage. The left sidebar contains a navigation menu with sections like 'Material System' (Introduction, Material studies), 'Material Foundation' (Foundation overview, Environment, Layout, Navigation, Color, Typography, Iconography, Shape, Motion, Interaction, Communication), and 'GUIDELINES' (Material Theming). The main content area features a large 'Design' heading and a sub-headline 'Create intuitive and beautiful products with Material Design.' Below this is a large, colorful graphic illustrating various design concepts like typography, layout, and color. To the right, there's a 'POPULAR' section with links to 'Tools for picking colors', 'Iconography', and 'Text fields'. A 'DESIGN TUTORIAL' section titled 'Make your own Material theme' provides instructions on how to build a custom theme. The top navigation bar includes tabs for 'Design' (which is selected), 'Develop', 'Tools', and a search icon.

Along the left of the page is a list of guidelines for each component that makes up Material Design. On the right, you'll see news, tutorials and other information about Material Design. As Material Design is constantly evolving, it's a good idea to keep an eye on this page to see what kind of apps Google highlights as good examples of Material Design usage.

Take a moment to familiarize yourself with the page. Once you're ready to move on, click **Color > The color system** on the left of the page.

This takes you to the section of Material Design specific to color.

The screenshot shows the Material Design website's navigation bar at the top, with tabs for 'Design', 'Develop', and 'Tools'. A search icon is also present. The main content area has a dark background. On the left, there's a sidebar with sections like 'Material System' (Introduction, Material studies), 'Material Foundation' (Foundation overview, Environment, Layout, Navigation), 'Color' (The color system, Color usage and palettes, Color theme creation, Tools for picking colors, Applying color to UI, Color usage, Text legibility), and 'Typeography'. The main content area features a large heading 'The color system' and a sub-section 'Color usage and palettes'. Below these are descriptive paragraphs and links.

The color system

The Material Design color system can be used to create a color theme that reflects your brand or style.

CONTENTS

- Color usage and palettes
- Color theme creation
- Tools for picking colors

Color usage and palettes

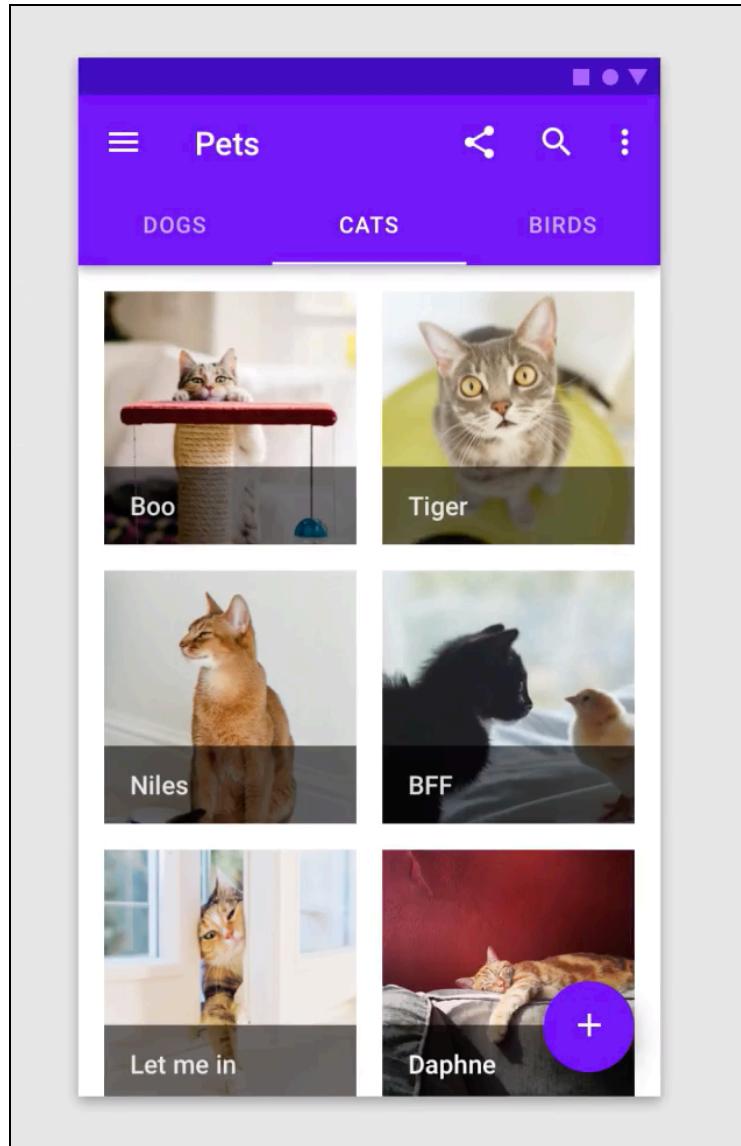
The Material Design color system uses an organized approach to applying color to your UI. In this system, a primary and a secondary color are typically selected to represent your brand. Dark and light variants of each color can then be applied to your UI in different ways.

Primary and secondary colors

Reading through this page, you'll understand the amount of emphasis color has in Material Design. To paraphrase, Color is important because it helps draw the user's attention to important areas of your app that you want them to interact with. However, Material Design also stresses that you shouldn't overdo it with color, because having too many colors in an app is distracting to a user and can confuse the purpose of Views on the screen.

Material Design focuses on two color choices: **Primary** and **Secondary**.

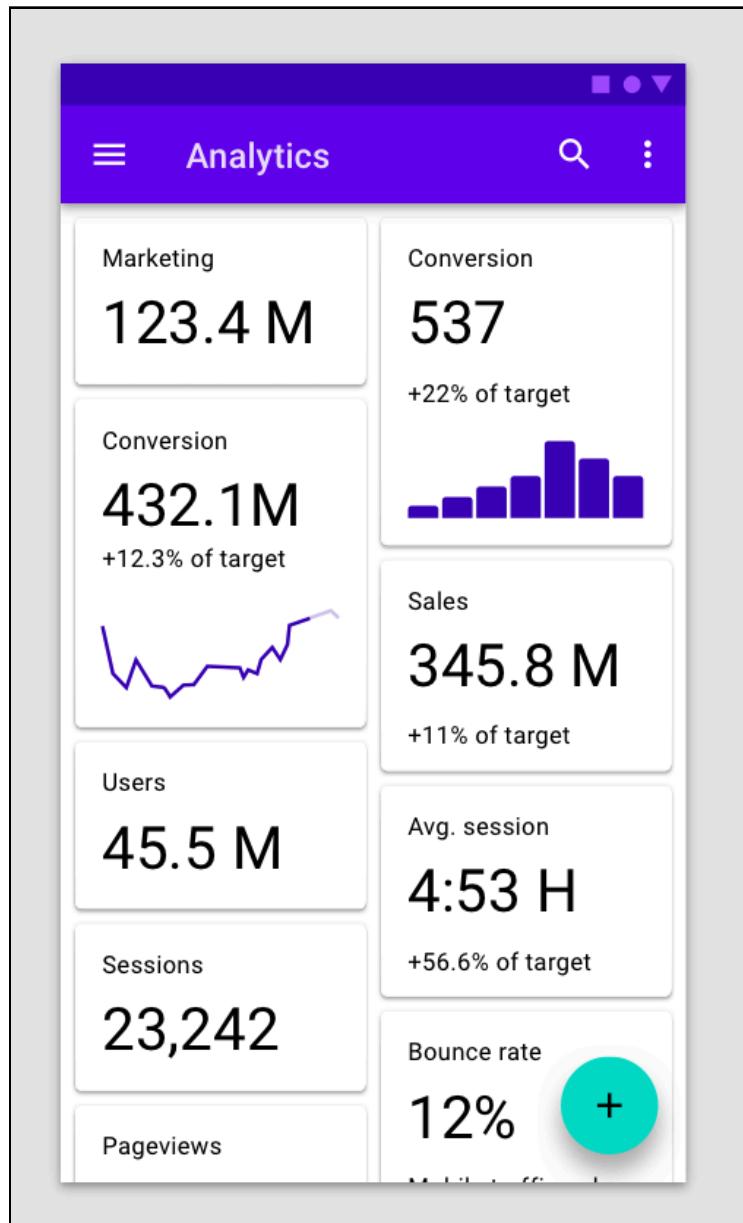
The Primary Color is what you'll use most often within your app. Generally speaking, it should be the color of your app's brand, and it works well for things like action bars and backgrounds.



The primary color here is purple

As for the Secondary Color, you can use it to accent certain areas of your app. A good rule of thumb is that it should contrast with the Primary Color.

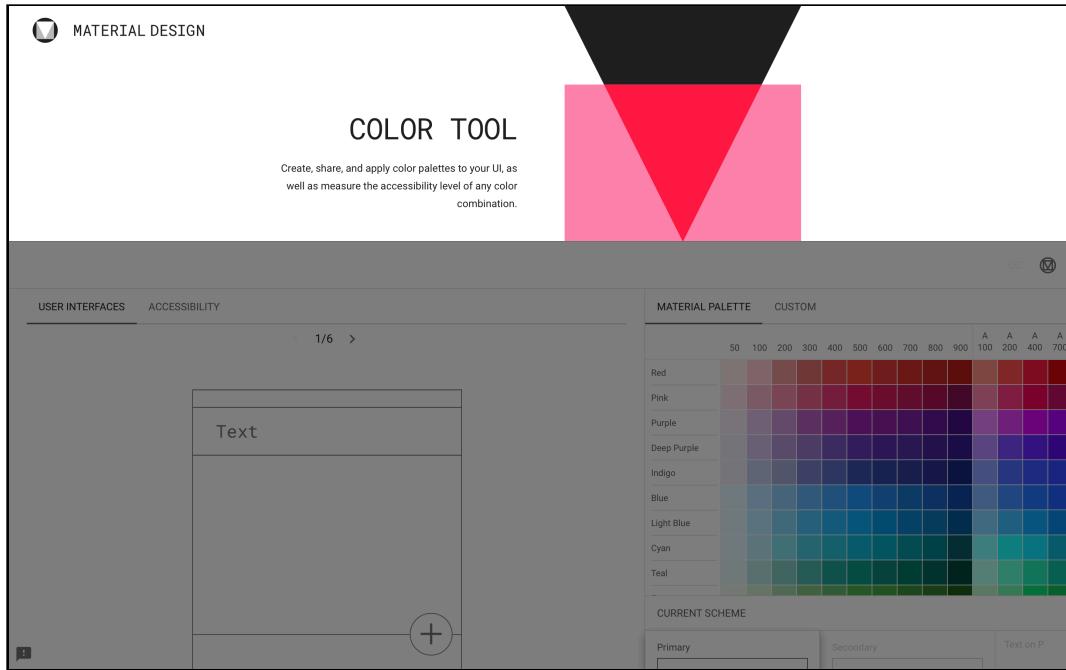
The Secondary Color works well for things like buttons, floating action buttons and progress bars — things you want your user to notice.



The secondary color here is turquoise

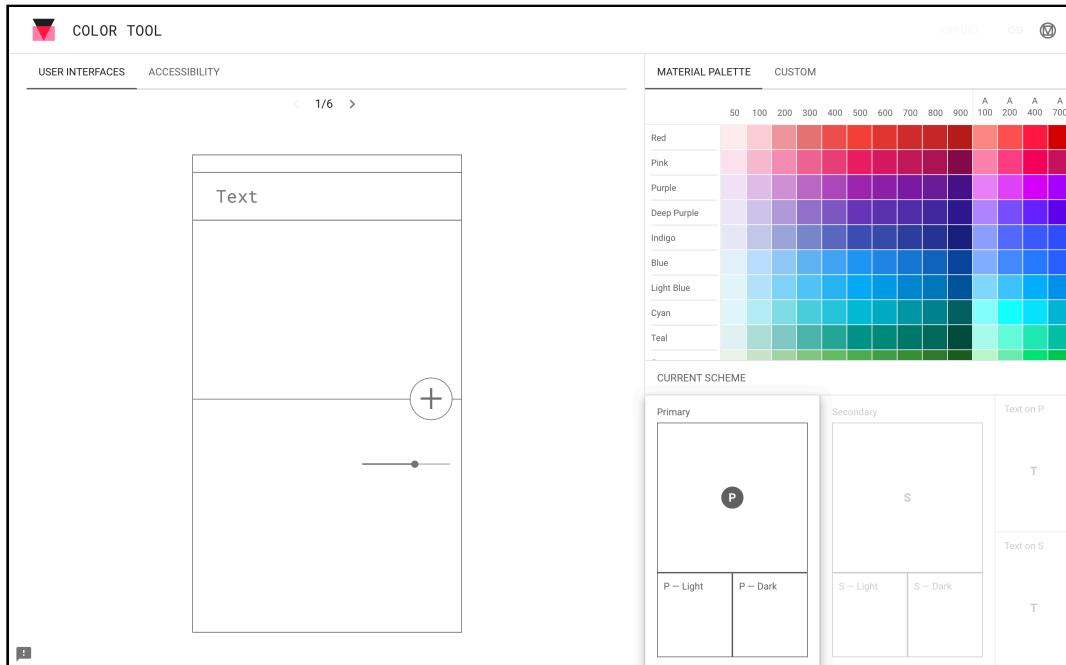
Using this knowledge about color in Material Design, it's time to bring some color to ListMaker.

In your browser, navigate to <https://material.io/tools/color/>; this takes you to the **Color Tool**.



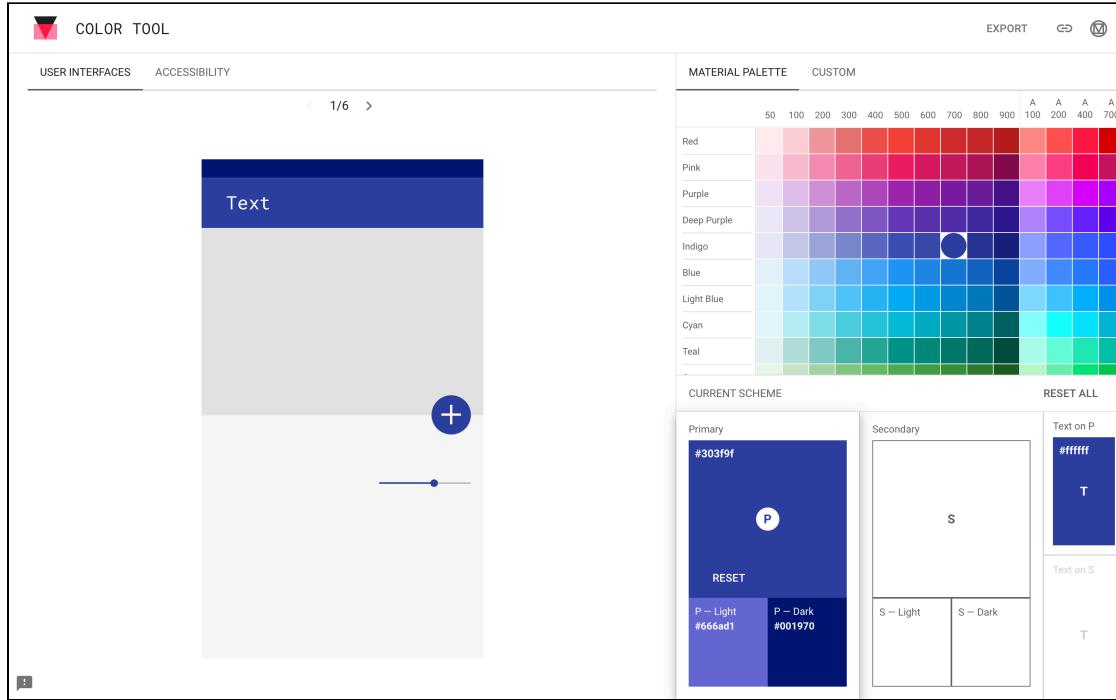
The Color Tool allows you to pick out colors to preview in a mockup app to see how they look.

At the moment, the Color Tool has no colors chosen, so the mockup app on the left of the screen doesn't contain any color.

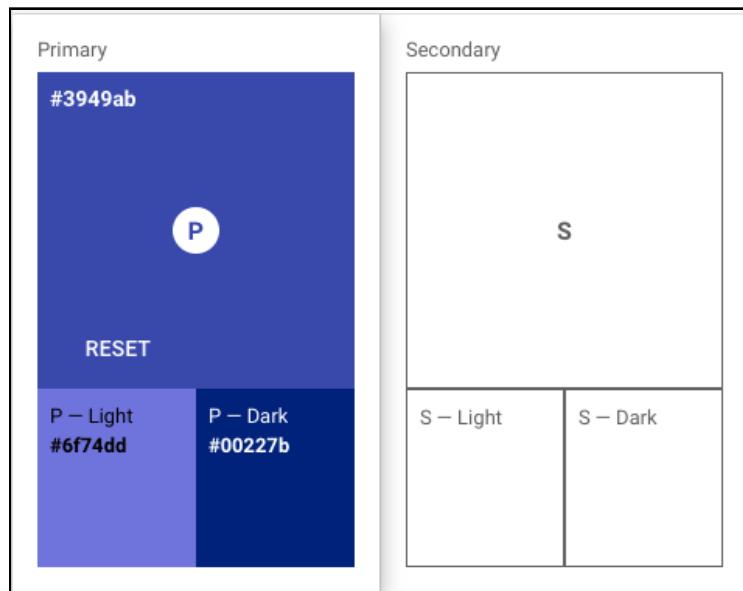


It's time to change that!

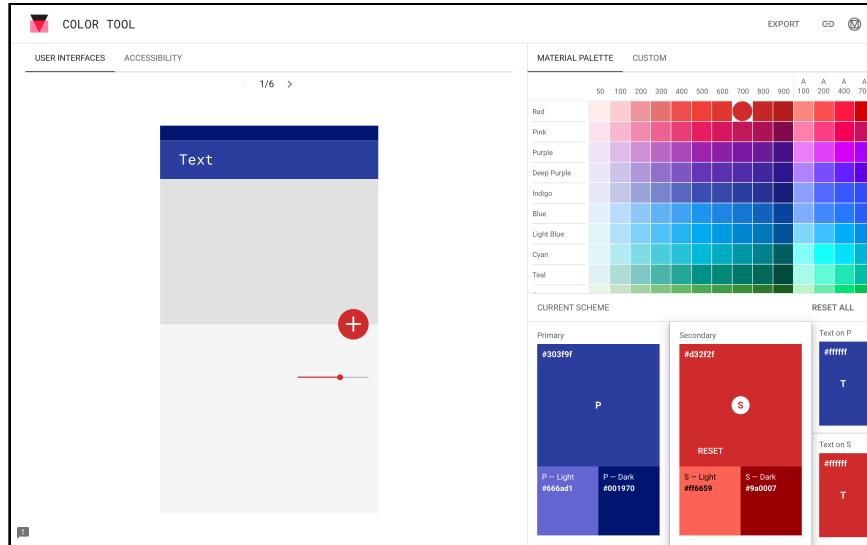
In the top-right of the Color Tool, choose a Primary Color that you like. Remember that this color is the one that's most used in your app, so choose wisely. The sample project uses an indigo color, but you can choose something else that appeals to your sense of design.



Next, in the bottom-right of the Color Tool, click the **Secondary** color scheme to set up the Secondary color.



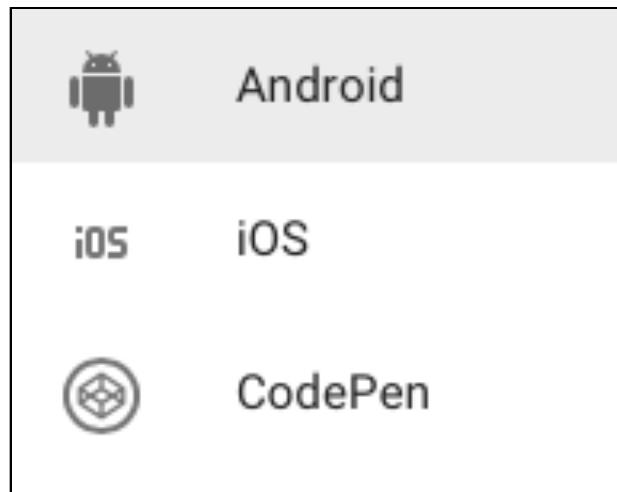
Select a different color for the Secondary Color. Remember, this color should contrast with the Primary Color. The sample project uses a dark red.



When you're happy with your selections, move the mouse cursor to the top-right of the browser window and click **Export**.



A new window appears over the export button, and gives you the option to export your theme for various platforms. Select **Android**.



The Color Tool exports your chosen colors to your computer: Check the download folder, and you'll find **colors.xml** ready for importing into your project.

Open the ListMaker project in Android Studio and navigate to **colors.xml** in **res > values**. This file is where you declare 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 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**; it's located in the same directory as **colors.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>
```

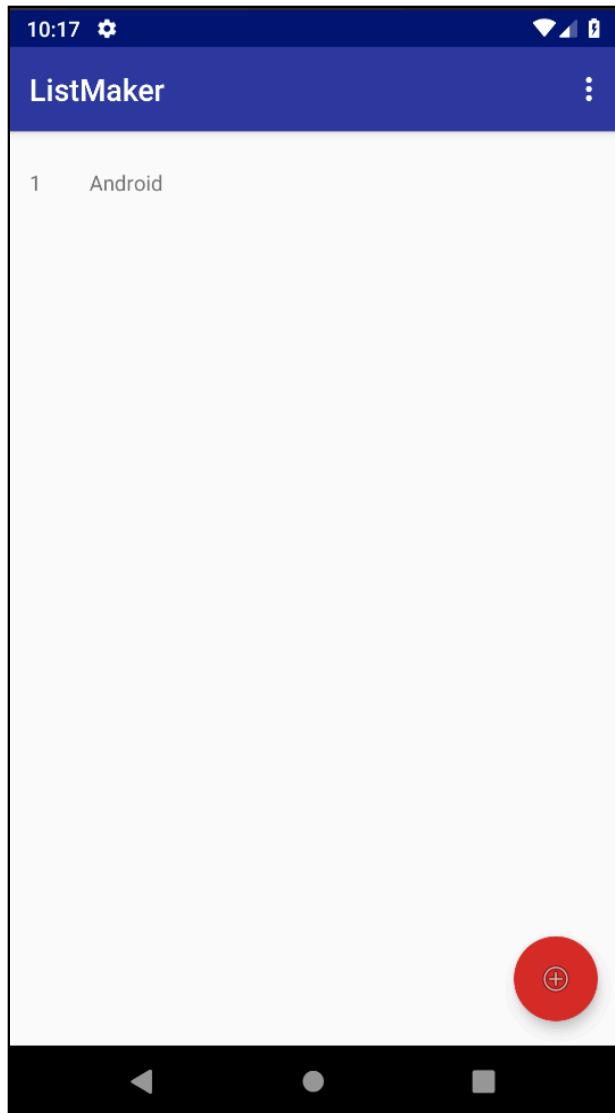
This file is responsible for declaring **Themes** for your app. A theme is a set of grouped attributes. You can create multiple themes for your app to avoid repeatedly declaring the same attributes in your Layouts and Views. This is useful if you want all of the buttons in your app to look the same, and you want a single place from which to change things should you ever decide to do so.

You might be seeing errors with this file because the colors used in the **AppTheme** theme no longer exist. Within the AppTheme, rename the `colorPrimary`, `colorPrimaryDark` and `colorAccent` entries so they match the names of the colors you added to **colors.xml**:

```
<item name="colorPrimary">@color/primaryColor</item>
<item name="colorPrimaryDark">@color/primaryDarkColor</item>
<item name="colorAccent">@color/secondaryColor</item>
```

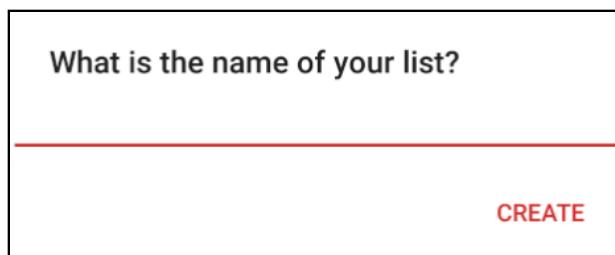
This tells your app what colors to use as the primary and secondary colors.

Time for the big reveal! Run your app and check out the difference in color.



Tap the FAB. It looks so nice!

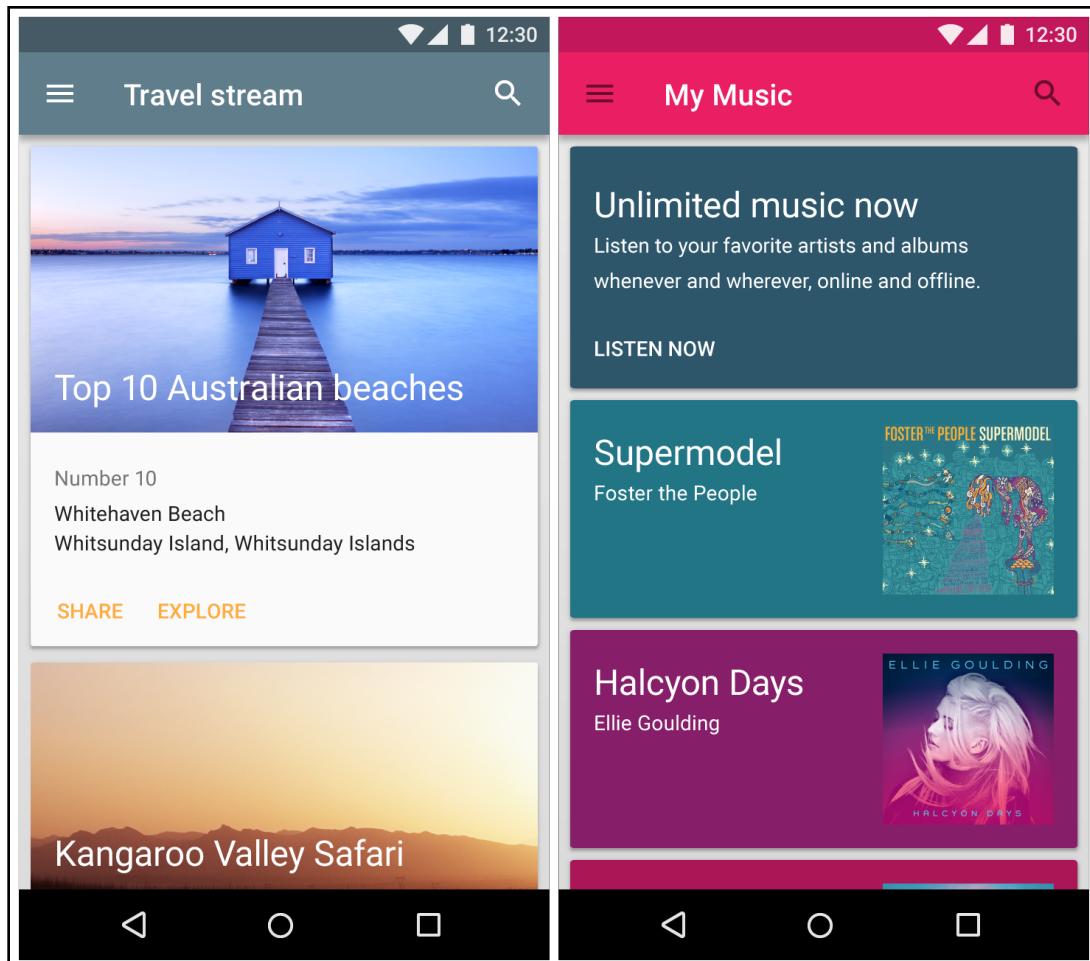
Also, notice the accent color change in the dialog when tapping the EditText:



Excellent! You just updated the app to use some Material Design colors of your own.

Card views

As you continue reading the Material Design site, you'll notice that it often emphasizes the use of **Cards**. Cards are designed as a gateway to more information.



Example of apps using Cards

You can use Cards to visually inform your users that a list contains some tasks in ListMaker.

Note: For more information about Cards read the following: <https://material.io/guidelines/components/cards.html>.

To use a Card in your app, you need to declare a new dependency. Open **build.gradle** (**Module: app**), and in the dependencies block, add the following line:

```
implementation 'com.android.support:cardview-v7:28.0.0'
```

Click the **Sync Now** button that appears when you change the Gradle script. This rebuilds your project, pulling in any new dependencies that you added from the internet.

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 the project, you can now update the RecyclerView ViewHolders to use Cards.

Open the `list_selection_view_holder.xml` Layout in `res > layout`, making sure you have the **Text** Layout view open. Update the XML layout so that it matches this:

```
<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>
```

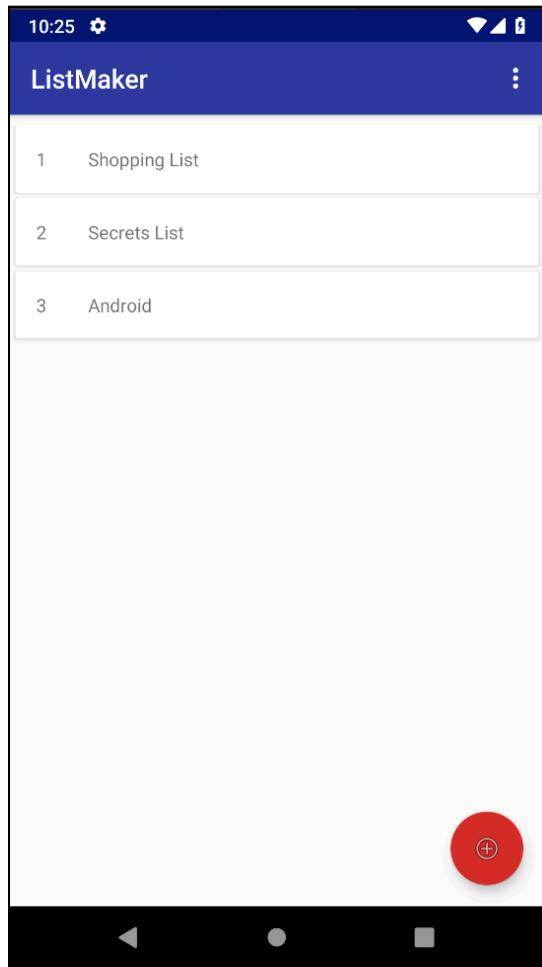
The `LinearLayout` and its containing `TextView` widgets stay the same. The significant change is that these components are now wrapped in a **CardView**.

Cards behave similar to other Layout widgets, with a few additional properties. To access these properties, you first need to assign a namespace; 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), used here.

It's also worth noting that the `CardView` has its margins pushed 4 dp from the left, right and bottom. This ensures that it doesn't hit the edge of the screen and isn't clipped by a card beneath it, which would obscure the `CardView`.

Run the app, and you'll see that now the collection of lists look more appealing.



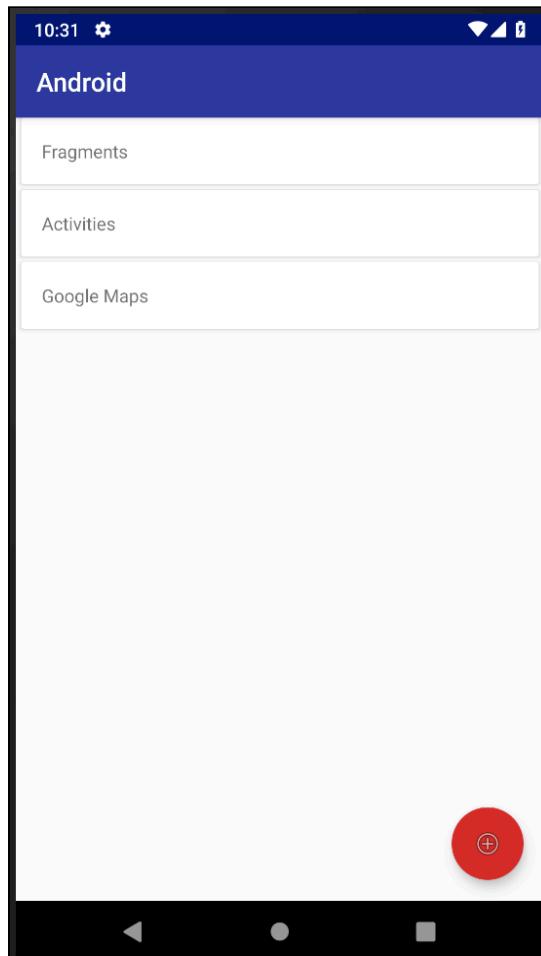
It's time to do the same for `task_view_holder`!

Open `task_view_holder.xml` Layout in `res > layout`, making sure the **Text** tab is selected. Update the XML to wrap a `CardView` around the root `LinearLayout`, like so:

```
<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_marginLeft="4dp"  
    android:layout_marginRight="4dp"  
    android:layout_marginBottom="4dp"  
    card_view:cardCornerRadius="2dp">
```

```
<LinearLayout  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:orientation="vertical">  
  
    <TextView  
        android:id="@+id/textview_task"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:layout_marginStart="16dp"  
        android:layout_marginTop="16dp"  
        android:layout_marginBottom="16dp"  
        android:text="TextView" />  
    </LinearLayout>  
</android.support.v7.widget.CardView>
```

Run the app, and click on a list that already contains some tasks. Notice how the CardView makes your tasks pop off the screen.



Where to go from here?

You only had a peek into the benefits Material Design brings to an app. With Material Design, you can provide your users with a great experience. In future apps, it's worth reading more about the Material Design guidelines and finding ways to incorporate it into your app. Don't forget to 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 Namrata Bandekar

Have you ever been on a road trip and wanted to make notes about the places you've visited; needed to warn your future self about some heartburn-inducing greasy food from a roadside diner; or you wanted to keep reminders about the best menu items at your favorite local restaurants?

If you answered "yes" to any of those questions, 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.

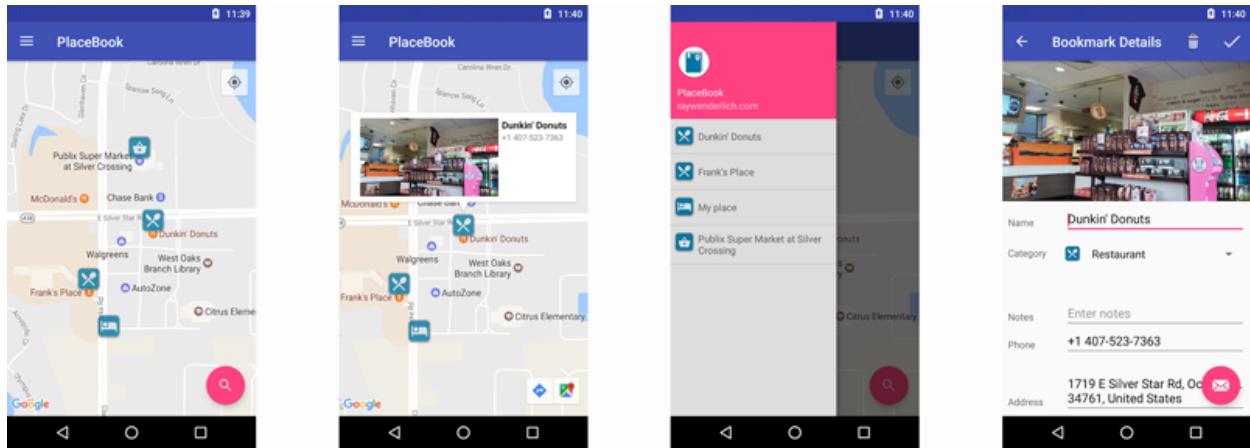
Getting started

While building PlaceBook, you'll use familiar techniques from the previous sections and learn about several new Android APIs. Along the way, you'll use:

- **Google Maps API** to display a map, track the user's location and add custom markers.
- **Google Places API** to display place information and search for nearby places.
- **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, but in the end, 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 corresponding 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 best to think about them in bite-sized pieces. From there, you can slowly build up to the finished product. The steps you'll take to accomplish this are as follows:

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:

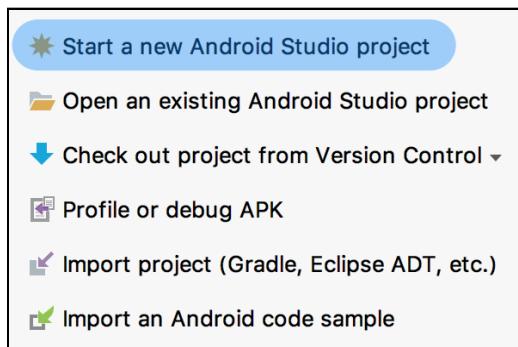
- **Framework Location APIs:** Known collectively as the **location framework**, this framework has been around the longest and is the traditional means for getting the user's current location. However, it's also the most difficult to use.
- **Google Maps API:** The Google Maps API makes it easy to display 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 a user's location. You'll be using the **FusedLocationProviderApi** component of this API in the book.

Map wizard walk-through

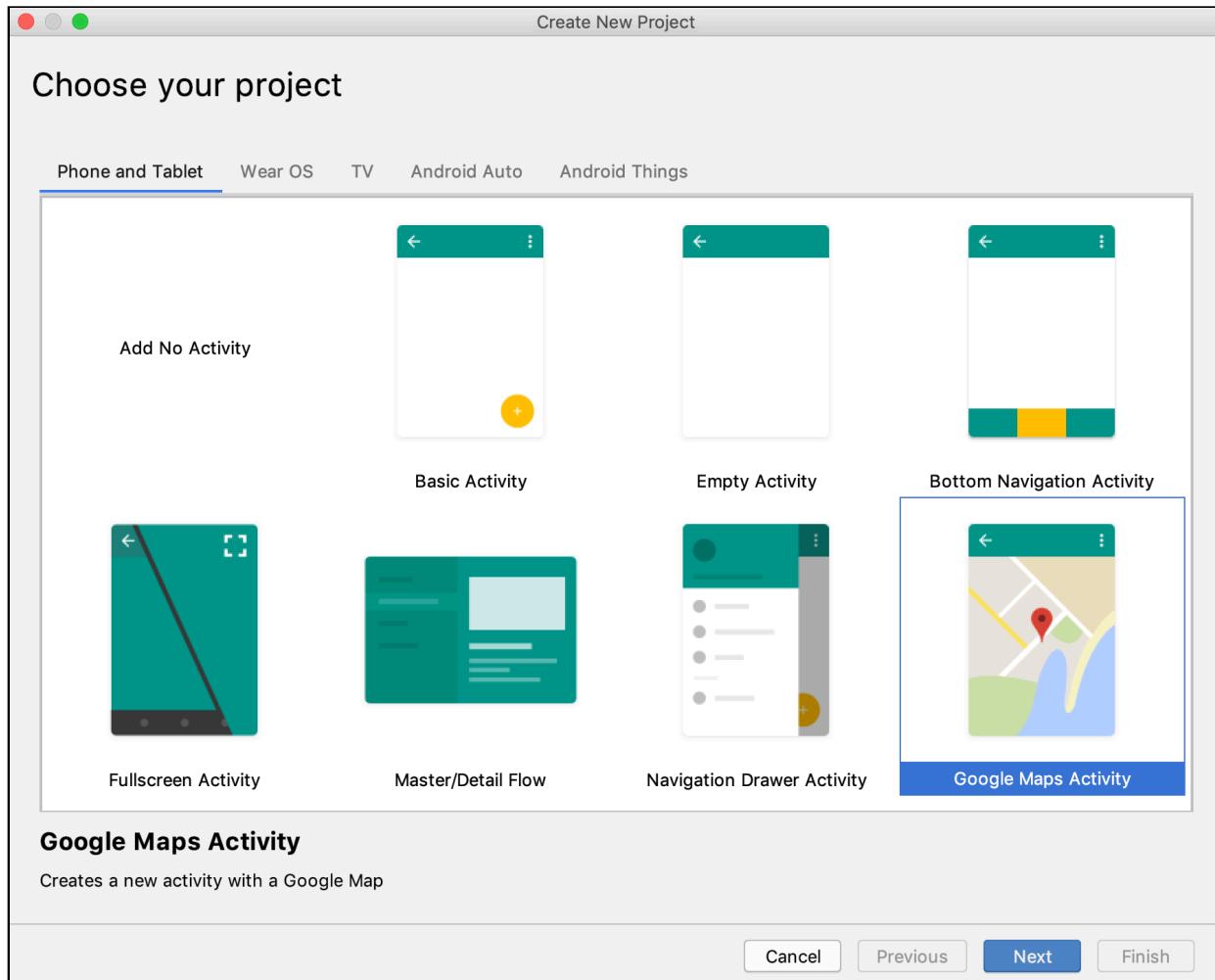
To save time, you'll use the Maps Activity project template to generate an app with a single Activity that displays a map.



To begin, launch Android Studio and select **Start a new Android Studio Project**.

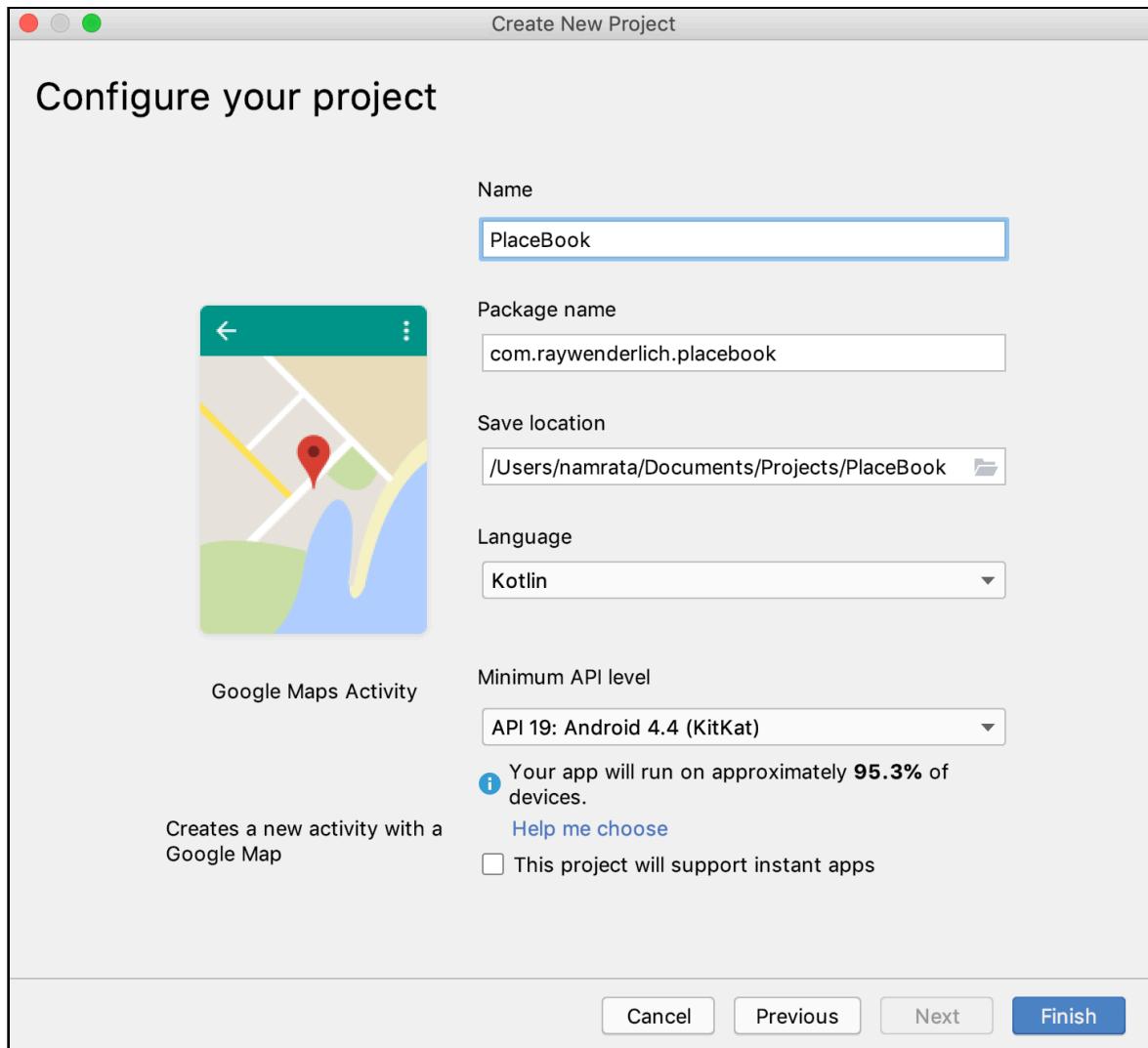


Select **Google Maps Activity** under **Phone and Tablet**. You'll find this on the **Choose your project** dialog. Once selected, click **Next**.



Fill out the **Configure your project** dialog with the information below:

- Name: PlaceBook
- Package name: com.raywenderlich.placebook
- Save location: select a directory for the project files
- Language: Kotlin
- Minimum API level: API 19
- Leave everything else unchecked.



Click **Finish**.

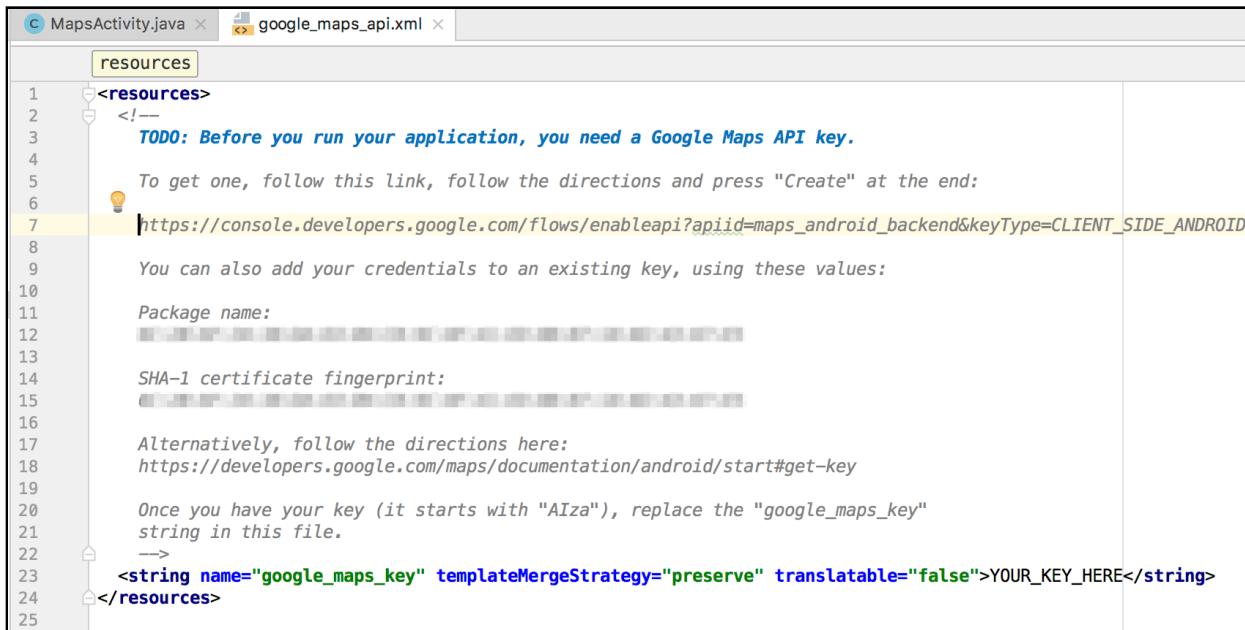
Android Studio automagically generates your new project and performs an initial build. If all goes as planned, you're left viewing the `google_maps_api.xml` file.

Google Maps API key

Before your app will work, you need to generate an API key using the free Google Developer Console, which requires a Google account.

The Google Maps API communicates with the Google Map servers and only works if a valid key is provided by the app. Android Studio generates the `google_maps_api.xml` file to make things easier. It also provides important information to help you create the Google Maps API key.

The easiest way to create an API key is to use the link at the top of `google_maps_api.xml`, shown here and highlighted in yellow:



```
MapsActivity.java x google_maps_api.xml x
resources
1  <resources>
2    <!--
3      TODO: Before you run your application, you need a Google Maps API key.
4
5      To get one, follow this link, follow the directions and press "Create" at the end:
6      https://console.developers.google.com/flows/enableapi?apiid=maps_android_backend&keyType=CLIENT_SIDE_ANDROID
7      You can also add your credentials to an existing key, using these values:
8
9      Package name:
10     [REDACTED]
11
12      SHA-1 certificate fingerprint:
13     [REDACTED]
14
15      Alternatively, follow the directions here:
16      https://developers.google.com/maps/documentation/android/start#get-key
17
18      Once you have your key (it starts with "AIza"), replace the "google_maps_key"
19      string in this file.
20      -->
21      <string name="google_maps_key" templateMergeStrategy="preserve" translatable="false">YOUR_KEY_HERE</string>
22
23  </resources>
```

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're not familiar with SHA-1. SHA-1 is a method for generated secure hashes. Just like a real fingerprint uniquely identifies an individual, each SHA-1 fingerprint uniquely identifies a set of bytes.

The fingerprint in `google_maps_api.xml` is an SHA-1 hash of the certificate from your debug keystore file. A keystore file contains everything you need 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.

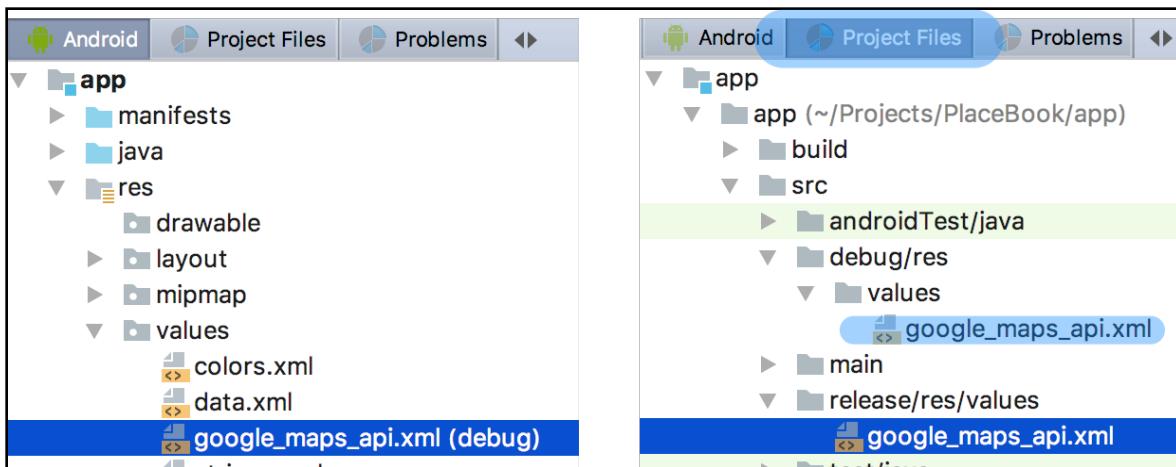
The debug keystore file is automatically generated when you first install Android Studio and is shared among all of your projects. Using a release keystore is covered in detail in Chapter 30, “Preparing for Release”.

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.

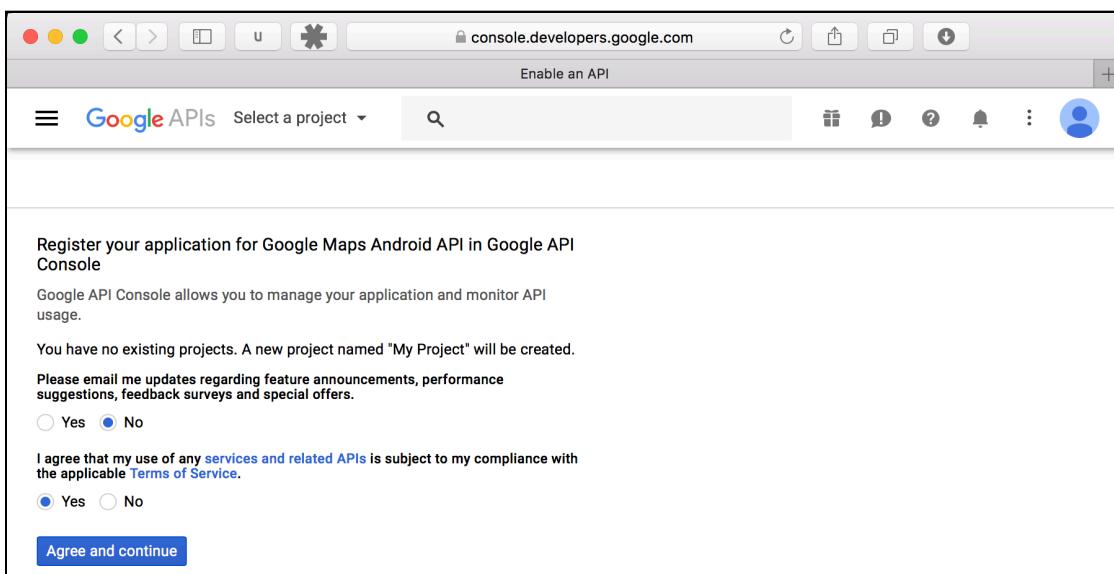
There are actually two versions of `google_maps_api.xml` in your project. One version is used only when building the debug version, while the other is used only for the release version.

If you're using the **Android View** in the **Project View**, you'll only see one version of the file in the `app/res/values` folder; however, you'll see **(debug)** after the filename.

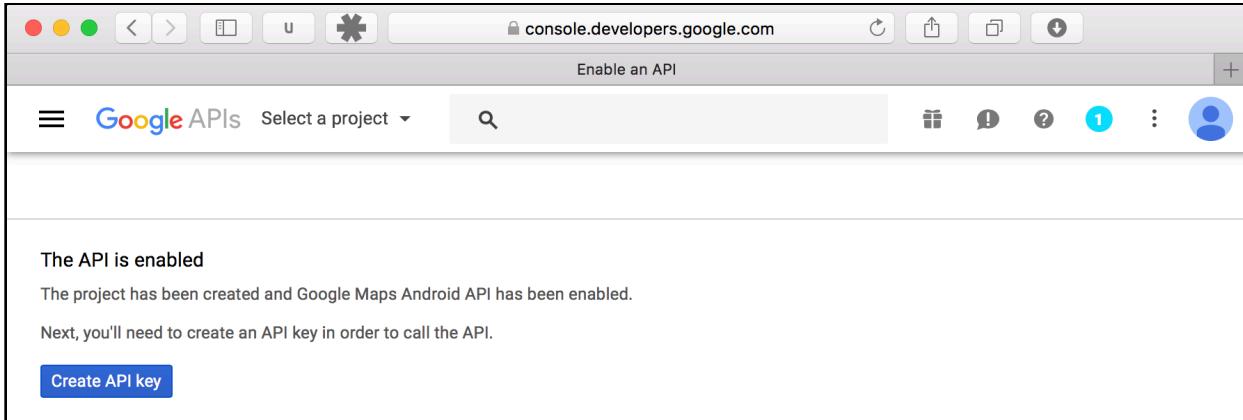
To see both versions, switch over to the Project Files view by selecting the **Project Files** tab in the **Project View** window. Open the `app/app/src/debug/res/values` and `app/app/src/release/res/values` folders and you'll notice that there's a `google_maps_api.xml` file in each one. By placing files in these build-specific folders, Android Studio can apply them separately to debug or release builds as appropriate.



Follow the link provided in `google_maps_api.xml` and you'll see the following page after signing in to your Google account.

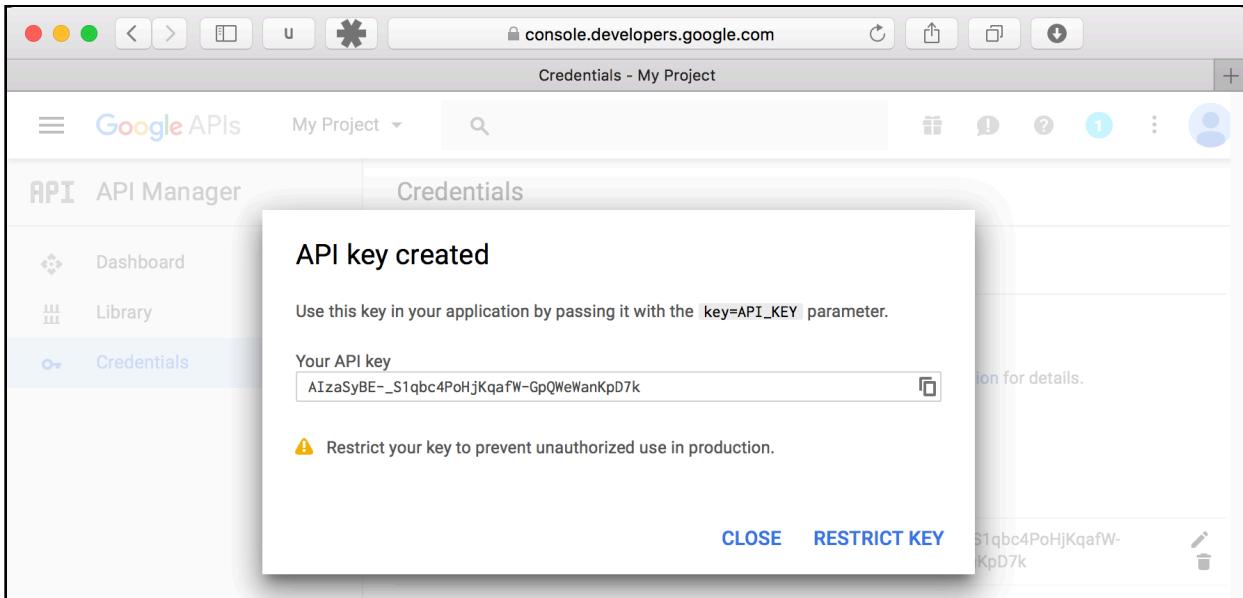


Click **Agree and continue**, and you'll come to a page displaying **The API is enabled**.



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 edit it later.

Click **Create API key** 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 is already 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 reads `YOUR_KEY_HERE`. The resulting file will look something like this, but with your key instead:

```
<string name="google_maps_key"
    templateMergeStrategy="preserve"
    translatable="false">
    AIza5sD-_G2dq7PjafW-Ad4pKpU5a</string>
```

Getting the keystore fingerprint

Although Android Studio conveniently placed your debug keystore fingerprint in the XML file, it's helpful to know how to get the fingerprint yourself should you ever need to regenerate it. The following instructions work for debug builds; getting the SHA1 key for release builds is covered in Section VI, "Submitting Your App".

First, locate your keystore file.

- On macOS, the keystore file is located in `~/.android/`.
- On Windows, you'll find the keystore file 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 produces output similar to this:

```
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 will match what's already in the XML file.

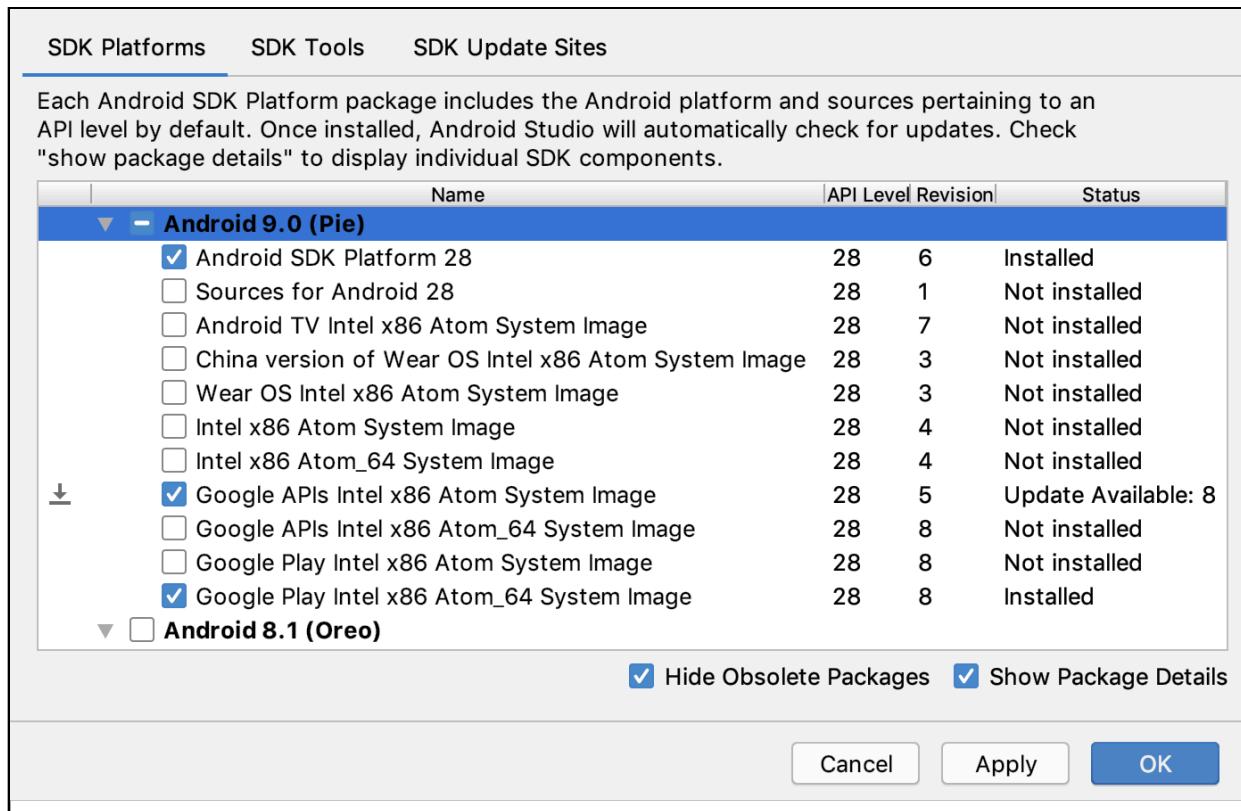
Maps and the emulator

If you're installing on a device, that's all you need. However, 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, use the following steps to create an emulator with API Level 19 or newer that includes the Google APIs.

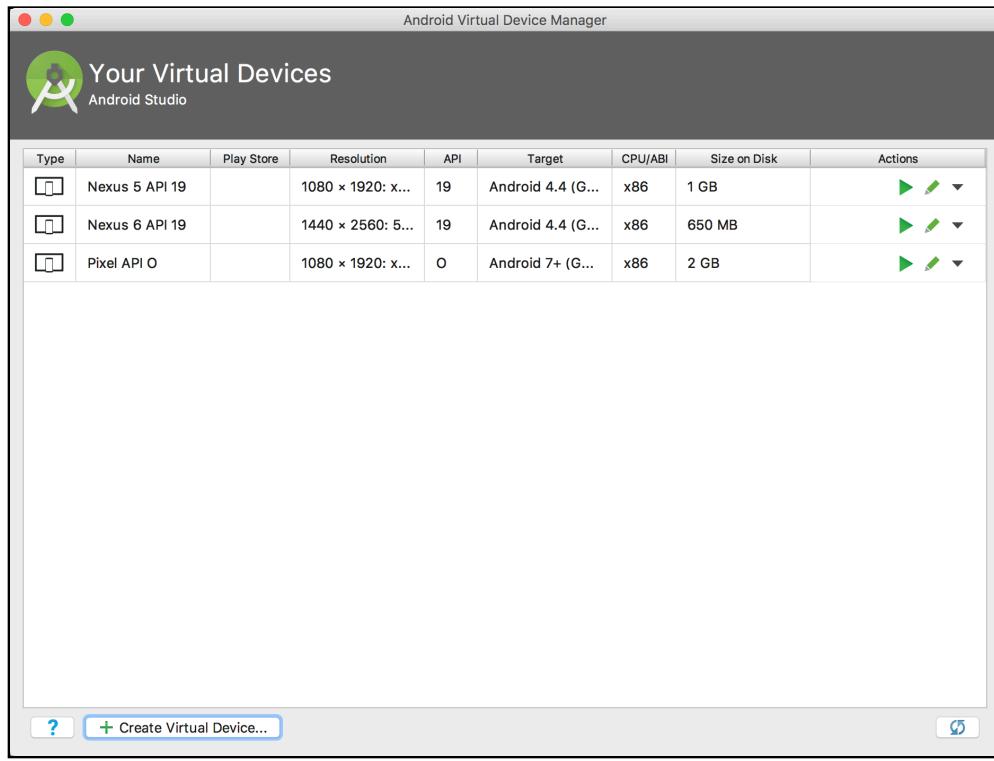
Select **Tools > SDK Manager**. Under the **SDK Platforms** tab, select **Show Package Details**.

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, select it as well. The following shows Android 9.0 (Pie) with the necessary items selected.

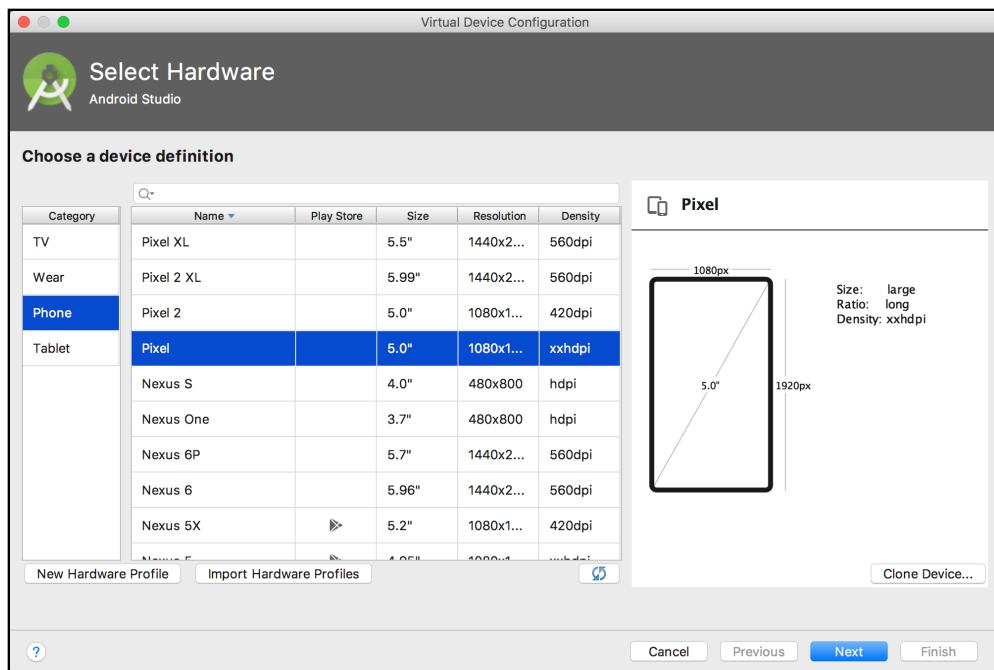


Click **OK** to install the platform files.

Once the installation is complete, select **Tools > AVD Manager**, and then click **Create Virtual Device**.

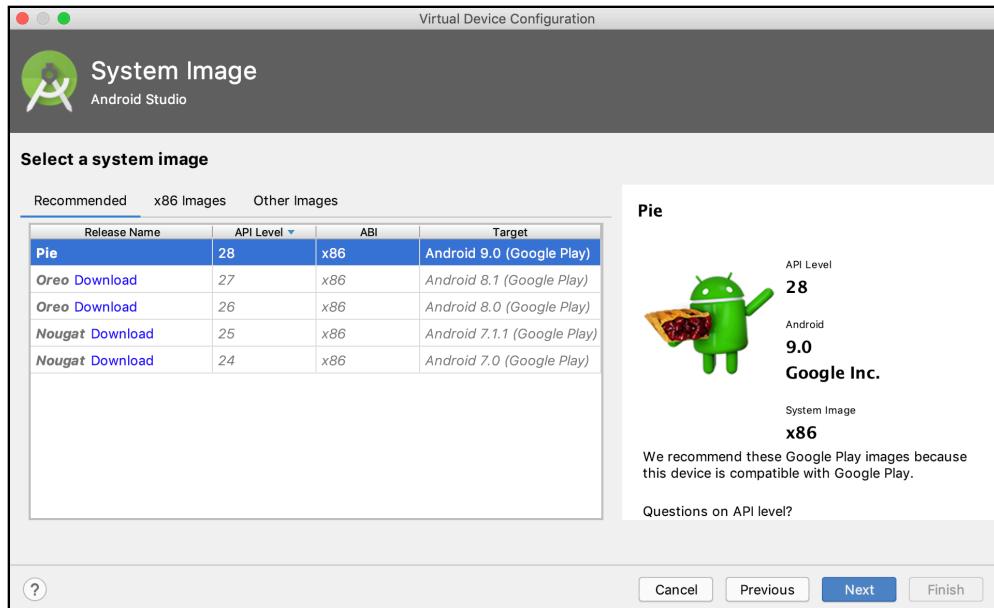


Select your preferred device and click **Next**. For demonstration purposes, the following screen shot shows an emulator set up for a Pixel device.

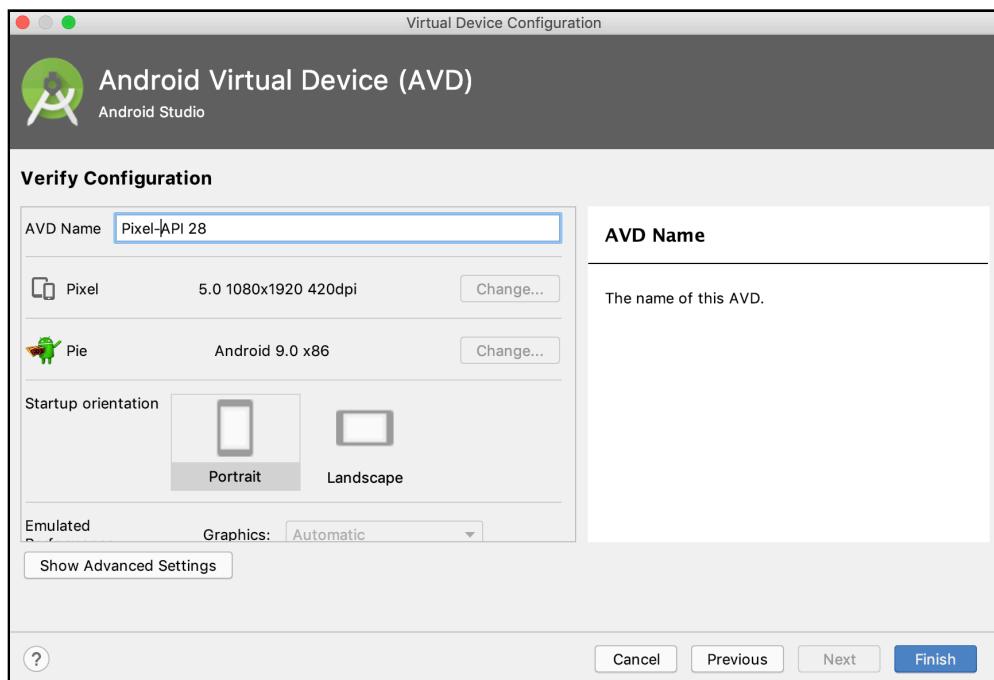


The **Recommended** tab displays a choice that matches the SDK platform files you downloaded in the previous step. Make sure you select the one with the Google APIs option. If you don't see one on the Recommended tab, then try the **x86 images** tab.

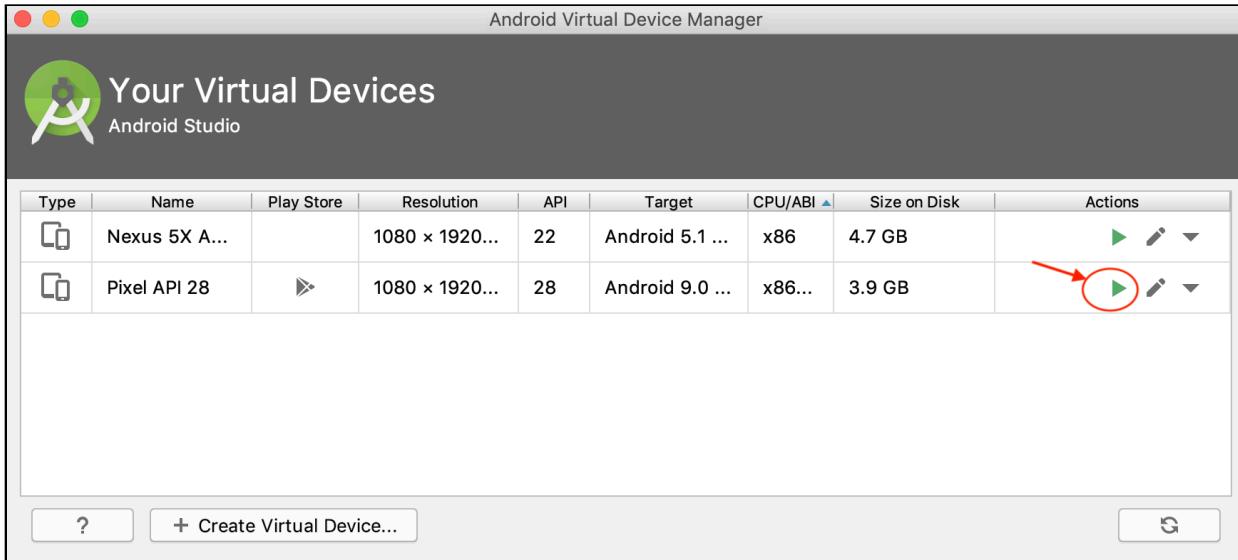
Click **Next**.



On the Configuration screen, leave the default settings as-is and click **Finish**.



You'll 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'll see a map on the screen. If you see a blank screen, check Logcat for error messages.

If you see an error message in Logcat that looks like the one shown here, double check that you 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 ensure that it matches the one you received when you created your key.

Once you have the correct key, you'll 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 but that will change soon enough!

Before moving on, take a moment to review the files Android Studio created for you.

Project dependencies

Before you can use maps in your app, you have to add the two required dependencies. To find the first one, open **build.gradle** from your application module folder. In the dependencies section, you'll see the following line:

```
implementation 'com.google.android.gms:play-services-maps:16.0.0'
```

This instructs the Gradle build system to include the Maps API in your build and is required to use maps.

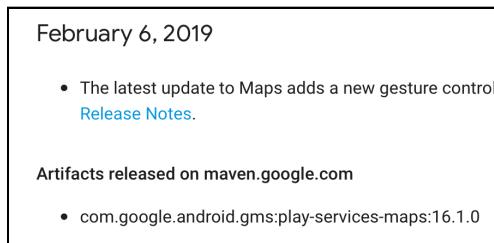
You may be wondering, “How do I know which version of the library to include?”

Good question! There are at least three ways to find the latest version:

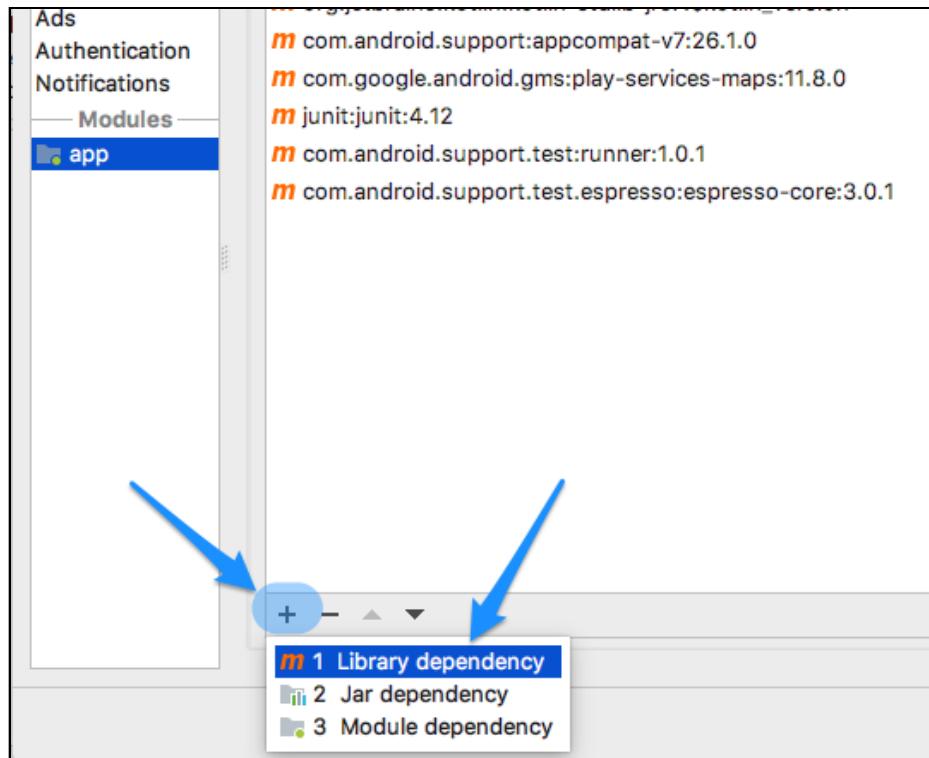
1. Go to <https://developers.google.com/android/guides/setup>. Scroll down to see the list of APIs. This list is dynamically generated and reflects the most recent version of each API.

API	Description in build.gradle
Google+	com.google.android.gms:play-services-plus:16.0.0
Google Account Login	com.google.android.gms:play-services-auth:16.0.1
Google Actions, Base Client Library	com.google.android.gms:play-services-base:16.1.0
Google Sign In	com.google.android.gms:play-services-identity:16.0.0
Google Analytics	com.google.android.gms:play-services-analytics:16.0.6

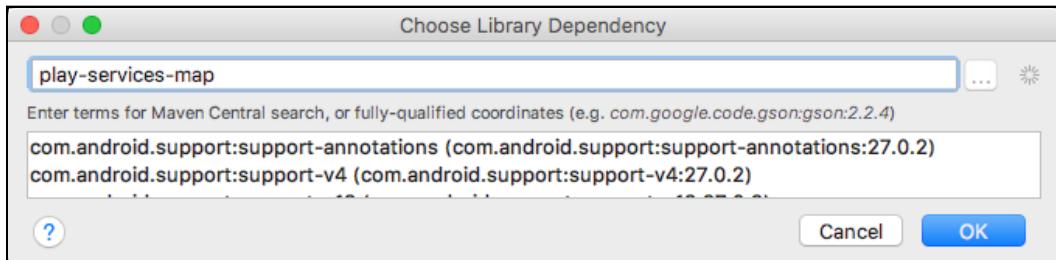
2. Go to <https://developers.google.com/android/guides/releases>. Note the latest release of Maps SDK for Android.



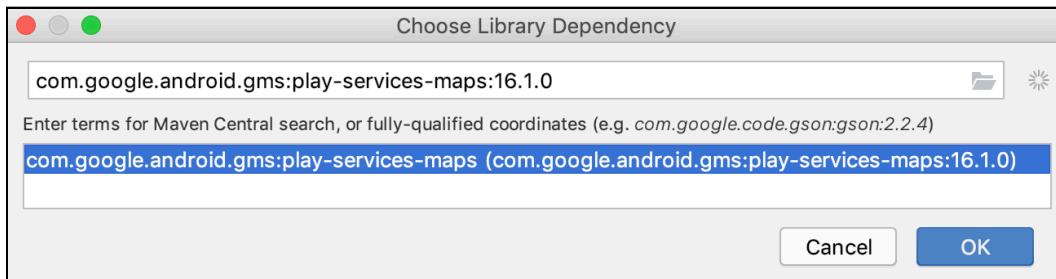
3. Select **File > Project Structure**. Select **app** under Modules, then select the **Dependencies** tab, click **+**, and select **1 Library Dependency**.



Type **play-services-maps** and press **Enter**.



You'll see 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 only the subset of play services required for your app, such as the Google Maps API.

The manifest

First, from **app/manifests**, open **AndroidManifest.xml**. It'll look like the following, with your API key displayed in place of `@string/google_maps_key`:

```
<?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 indicates, the `ACCESS_FINE_LOCATION` permission is not required to show the map. You *could* remove this line, and your app would continue to run 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'll show you the key.

The activity and layout

Open `MapsActivity.kt`. This is the startup Activity created from the Maps template. Note that it inherits from `AppCompatActivity` 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 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 the main UI in the **ListMaker** 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're responsible for forwarding lifecycle methods to the MapView. In lite mode, forwarding the lifecycle events is optional.

The template uses the MapFragment option. 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 it finds the map Fragment from the Layout and uses it to initialize the map using `getMapAsync()`.

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'll use to control and query the map.

If you're 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!

Look at `OnMapReady()`:

```
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()` is called by the `SupportMapFragment` object when the map is ready to go. You passed in a `GoogleMap` object that's 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 that's the case, the `SupportMapFragment` object prompts 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 some methods are used to add a marker and zoom the map to it. For now, don't worry about how the methods work; 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, you'll just use sensible naming, so go ahead and rename `mMap` to `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 include:

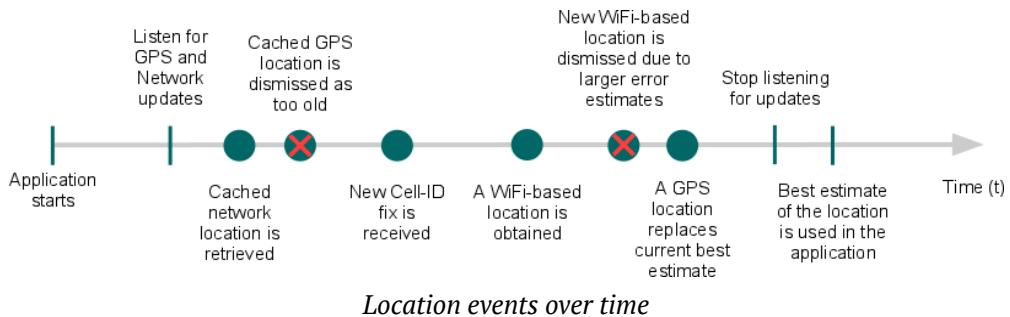
- **Dealing with multiple methods for determining location:** Your mobile device has several ways to determine your location, and each has its own benefits and disadvantages. Your phone uses the GPS chip, Wi-Fi location and cell towers to zero in on your location. You have to decide which one to use 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 provide 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 might look like this:

1. At some point after the application starts, begin listening for updates from the 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're done to preserve power.
4. Make use of the best location in your app logic.

The following graphic illustrates the location signals that your app might receive as it goes along. Note that the graph shows a time sequence of location events.



Note: Image credit [Android Open Source Project](#), used according to terms described in the [Creative Commons 2.5 Attribution License](#). The original image appears in [Location Strategies](#).

There are many decisions to make to determine how to best calculate the user's location:

- **Determine precisely 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.

Where to go from here?

As you can see, there are many moving parts, and there is 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.

Although you will continue to extend this app over the next few chapters, you can access more extensive information on the developer pages for the [Google Places API for Android](#).

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 Namrata Bandekar

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 is 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're following along with your own app, open it and keep using it with this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **PlaceBook** app inside the **starter** folder. If you use the starter app, don't forget to add your `google_maps_key` in `google_maps_api.xml`. Review 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 its dependencies.

The first order of business is to fix the starting location. 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, whereas this chapter takes a more in-depth look at how it works.

Fused location provider

The job of the fused location provider is to take all of 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. Directly ask 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 you can use to get the last known location of the device. If the device has not yet retrieved a location, this may return null.

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 uses 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's 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 if possible.

Note: The Google Play services API libraries are designed to be independent and the versions may be different. Currently, `play-services-maps` is at 16.1.0 and `play-services-location` is at 16.0.0, but to keep things simple, you'll use 16.0.0 for both.

```
implementation 'com.google.android.gms:play-services-location:16.0.0'
```

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 your Gradle dependency management. The `app/build.gradle` dependencies section now has two entries for `play-services` that both use the **16.0.0** 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 = '1.3.21'
```

Note: Your version might be different. Regardless, remove whatever `ext.kotlin_version` is in your file.

Update the first part of the `buildscript` section to match this:

```
buildscript {
    ext {
        kotlin_version = '1.3.21'
        play_services_version = '16.0.0'
```

```
}
```

You now have two properties defined within the build script domain that you can access from any .gradle file within 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.

Creating the location services client

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 `onMapReady()`:

```
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'll 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 an 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 app's manifest file. Android divides permissions into two main categories; **Normal** and **Dangerous**.

- **Normal permissions:** Permissions in this category are considered less harmful and are granted automatically if they're listed in the manifest. Examples of normal permissions include **BLUETOOTH**, **ACCESS_NETWORK_STATE**, **INTERNET** and **SET_ALARM**.
- **Dangerous permissions:** Permissions in this category can affect user's privacy or system stability. For these permissions, the system explicitly asks the user to allow the permissions. Examples of dangerous permissions include **READ_CALENDAR**, **READ_CONTACTS**, **CALL_PHONE** and **SEND_SMS**.

Android handles 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 is asked to approve the permissions when the app is first installed. If an app update adds new permissions, then the user is 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 won't display the specific permission when asking the user for permission; it'll 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 **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 `MapsActivity`:

```
companion object {
    private const val REQUEST_LOCATION = 1
    private const val TAG = "MapsActivity"
}
```

`REQUEST_LOCATION` is a request code passed to `requestPermissions()`. It's used to identify the specific permission request when the result is returned by Android.

`TAG` is passed into the `Log.e` method in the next code block. `Log.e()` is used to print information to the Logcat window to help 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 {
            val location = it.result
            if (location != null) {
                // 4
                val latLng = LatLng(location.latitude, location.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 that it centers on the location.

Here's how it works:

1. Check if the `ACCESS_FINE_LOCATION` permission was granted before requesting a location.
2. If permission was not 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 runs in the background to fetch the location. You request to receive notification when the location is ready by adding an `OnCompleteListener` to the `lastLocation` Task.

When the Task completes, 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 `location` is not `null`, you create a `LatLng` object from `location`. `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 gets displayed if tapped. 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.

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 viewing. 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:** You can show maps 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-widescreen. 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, you can call it once the map is ready.

Replace `onMapReady()` with the following code.

```
override fun onMapReady(googleMap: GoogleMap) {  
    map = googleMap  
    getCurrentLocation()  
}
```

Here, you initialize `map` when the map is ready to be displayed and then call `getCurrentLocation()`.

Finally, define the callback method to handle the user's response to the permission request. When `requestLocationPermissions()` is called, the system displays a permission dialog to the user. It then calls `onRequestPermissionsResult()` with the results.

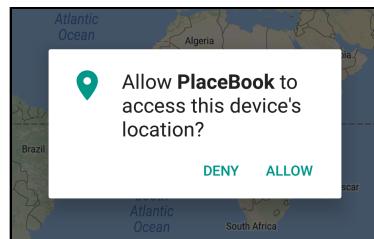
```
override fun onRequestPermissionsResult(  
    requestCode: Int,  
    permissions: Array<String>,  
    grantResults: IntArray) {  
    if (requestCode == REQUEST_LOCATION) {  
        if (grantResults.size == 1 && grantResults[0] ==  
            PackageManager.PERMISSION_GRANTED) {  
            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, you can use the granted permission and 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

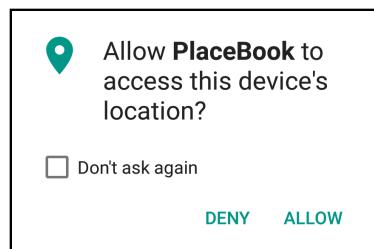
Run the app on a device or emulator running Android 6.0 or newer, and you'll see the following prompt:



Click DENY, and the **Location permission denied** message appears in Logcat.

```
06-21 16:20:02.114 5178-5178/com.raywenderlich.placebook E/MapsActivity: Location permission denied
```

Rotate the device, and the prompt displays again with one small change, offering the user a chance to tell the system “Don’t ask again”.



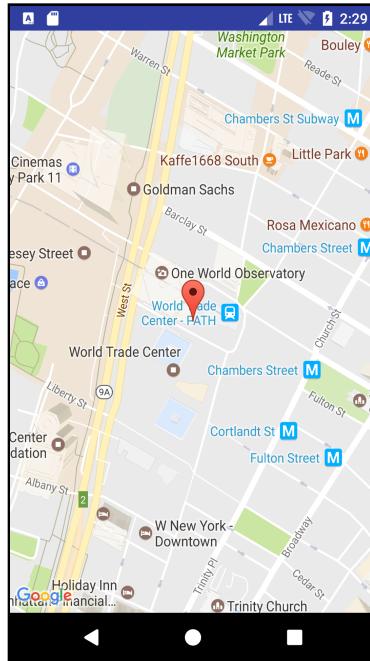
If you choose “Don’t ask again”, the dialog won’t be displayed again within the app. The only way to then grant permissions is to manually turn them on in device settings by tapping on **Apps>PlaceBook>Permissions**.

Note: Google recommends that you display a more detailed reason for asking for permission if the user denies it multiple times. There’s 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 will return your current location and then zoom the map to your current location.

If you're running on a 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, however, 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
```

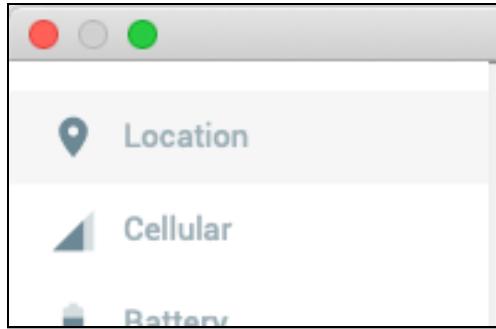
This is because the emulator hasn't simulated a user location. An emulator doesn't have access to GPS hardware, so you need another way to supply GPS locations.

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 does not have any location data from which to pull. 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 the 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

A screenshot of the "Extended controls" dialog. The title bar says "Extended controls" with a close button. The main area is titled "GPS data point".

Coordinate system	Decimal	Longitude
		-78.8704978

Currently reported location:

Longitude: -122.0840
Latitude: 37.4220
Altitude: 0.0

Latitude
42.90237

Altitude (meters)
0.0

At the bottom right is a "SEND" button with a green icon.

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 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 PlaceBook 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 is a little finicky, but eventually, it'll 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'll update to the 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()`

To request updates from the location client, you need a `LocationRequest` object to describe the level of accuracy you want 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 uses very little power and only polls 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 uses 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 set `priority` to `LocationRequest.PRIORITY_HIGH_ACCURACY` so it'll 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 gets 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 calls `LocationCallBack.onLocationResult` when it has a new location ready. You define a `LocationCallBack` object with `onLocationResult()`. 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()` is 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 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'll 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, you can accomplish changes #1 and #2 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()`:

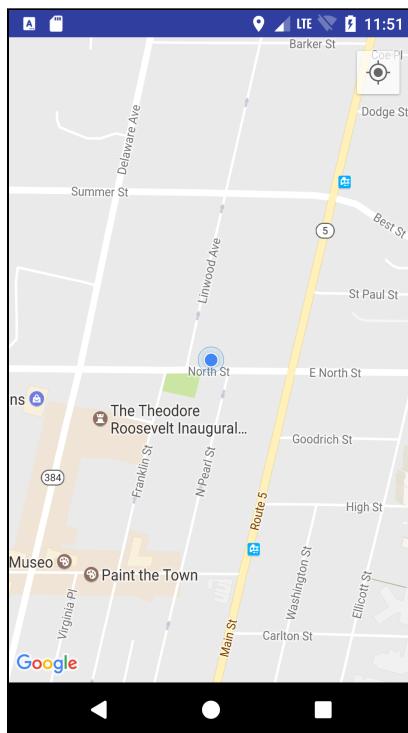
```
if (locationRequest == null) {  
    locationRequest = LocationRequest.create()  
    locationRequest?.let { locationRequest ->
```

```
// 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 completed everything needed for the basic map controls! In the next chapter, you'll start working with Google Places.

Chapter 15: Google Places

By Namrata Bandekar

Before you can achieve the 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're following along with your own app, open it and keep using it with this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **PlaceBook** app inside the **starter** folder. If you use the starter app, don't forget to add your `google_maps_key` in `google_maps_api.xml`. Read 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 its dependencies.

If you're following along with your own app, you'll also need to copy `default_photo.png` from `src/main/res/drawable-xxx`, which is included with the starter project, into your project:

Make sure to copy the files from all of the drawable folders (hdpi,mdpi,xhdpi,xxhdpi).

Before using the Google Places API, you 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 this book might be slightly different than what you

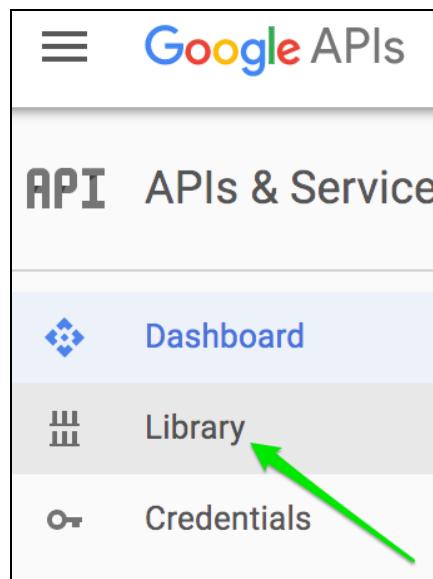
see on the Google developer portal since Google changes these often.

Enable the places API

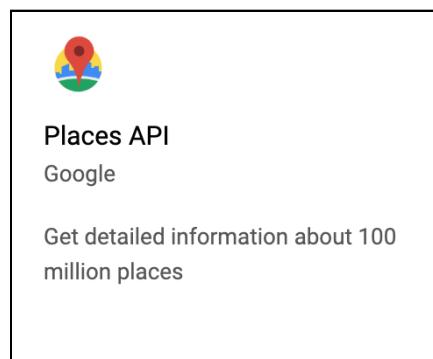
The Maps SDK for Android was enabled on your Google developer account when you created the initial Google Maps key. However, you need to turn on the Google Places API manually.

Log into your Google developer account at <https://console.developers.google.com>

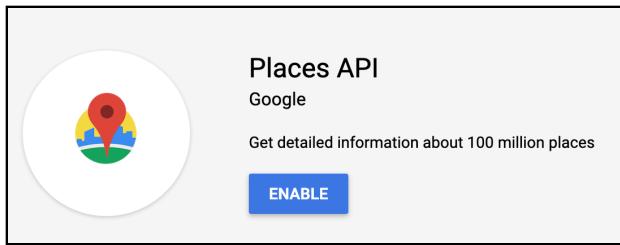
Ensure the project containing the Maps API key you created previously is selected. Switch to the Library tab on the left.



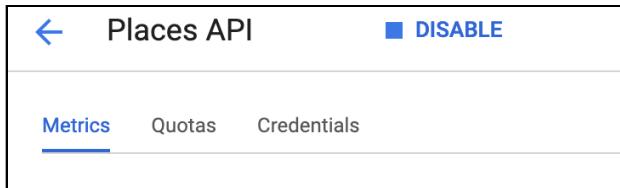
Click on **Places API**.



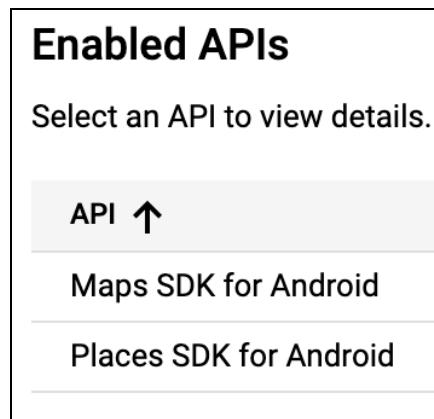
You'll see the following screen with an **Enable** button:



Click on **ENABLE** and wait while Google enables the API. After the API is enabled, the screen changes to show your app's metrics for this API.



Click on the back arrow next to Places API until you get back to the main Dashboard. Both the Maps SDK and Places SDK for Android are listed.



One last thing. To use the Places SDK for Android, you must enable billing on each of your projects that use the SDK. To do this sign up for billing at <https://console.cloud.google.com/projectselector2/billing/enable>.

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. PlaceBook 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 addresses, geographic locations, phone numbers, reviews and photos.

Note: Google Places for Android enforces a default limit of 70,000 requests per month. You can see the details of the pricing model at <https://cloud.google.com/maps-platform/pricing>. 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 location API, you'll have to add the Places API dependency yourself.

Open **app/build.gradle** and add the places library to the dependencies section as follows:

```
implementation "com.google.android.libraries.places:places:1.0.0"
```

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 **POIs**, 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 taps on a POI. You 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()  
}
```

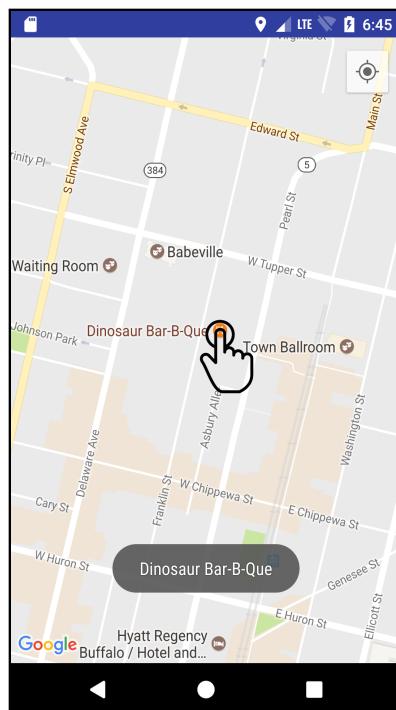
Here, you call `setOnPoiClickListener()` on `map` and provide it a lambda that implements the single `onPoiClick()` method of the `PoiClickListener` interface.

The `map` object will call your lambda anytime it detects that the user has tapped on a POI. The lambda 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 normally matches what's 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 toast messages 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 the Places SDK.

Before using the Places SDK, you need to create a `PlacesClient`. This client is your gateway to all of the available APIs provided by Places API.

In `MapsActivity.kt`, add the following import statements to import the places client library and the `PlacesClient`.

```
import com.google.android.libraries.places.api.Places  
import com.google.android.libraries.places.api.net.PlacesClient
```

Next, add a new private member below the `map` member:

```
private lateinit var placesClient: PlacesClient
```

Add the following method to `MapsActivity` under `onMapReady()`:

```
private fun setupPlacesClient() {  
    Places.initialize(getApplicationContext(), "YOUR_API_KEY");  
    placesClient = Places.createClient(this);  
}
```

This creates the `PlacesClient`. Every API call to the Places API must contain your API key. Fill in the API key that you obtained in Chapter 13 to replace the string `YOUR_API_KEY`. You'll notice that you're passing the application context to initialize the Places library. You also pass the current activity context to create the `PlacesClient`.

Note: Here, the API key is hard-coded for simplicity. In a production environment, you should use a secure mechanism to manage API keys.

Now, add the following at the end of `onCreate()` to ensure `PlacesClient` gets initialized only once when the activity is created.

```
setupPlacesClient()
```

Next, you want to use `PlacesClient` to fetch details about a place. Add the following method to `MapsActivity.kt`:

```
private fun displayPoi(pointOfInterest: PointOfInterest) {
```

```
// 1
val placeId = pointOfInterest.placeId

// 2
val placeFields = listOf(Place.Field.ID,
    Place.Field.NAME,
    Place.Field.PHONE_NUMBER,
    Place.Field.PHOTO_METADATA,
    Place.Field.ADDRESS,
    Place.Field.LAT_LNG)

// 3
val request = FetchPlaceRequest
    .builder(placeId, placeFields)
    .build()

// 4
placesClient.fetchPlace(request)
    .addOnSuccessListener { response ->
        // 5
        val place = response.place
        Toast.makeText(this,
            "${place.name}, " +
            "${place.phoneNumber}",
            Toast.LENGTH_LONG).show()
    }.addOnFailureListener { exception ->
        // 6
        if (exception is ApiException) {
            val statusCode = exception.statusCode
            Log.e(TAG,
                "Place not found: " +
                exception.message + ", " +
                "statusCode: " + statusCode)
        }
    }
}
```

Let's break this down:

1. First, you retrieve the `placeId` which uniquely identifies your place of interest.
2. Next, you create a field mask which contains only the attributes of a place you are interested in retrieving. This ensures that you only request data that you use and keeps your app's network usage under control. Notice that you're requesting a bunch of fields which you'll use in the latter part of this chapter.
3. You then use these two objects to create a fetch request. You use the familiar builder pattern to create this request.
4. Then, you fetch the place details using `placesClient`, which handles your request.

5. You add a success listener which is called if the response is successfully received. You then retrieve the place object which contains the requested details. You display the name and phone number for the selected place on the screen.
6. You also add a failure listener which catches any exception that could occur in the case the request fails. More specifically, you may want to know if there was an API error that occurred. You also log the status code and message to use for debugging the error.

Note: Many of the Places API methods like `fetchPlace` make network calls and can take a long time to return. For this reason, the places library offloads these tasks to the background and returns a Task. The network call completes asynchronously and then calls either your `OnSuccessListener` or `OnFailureListener` with a callback method.

Now, 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.

Build and 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 **Place not found: 9010: You have exceeded your daily request quota for this API**, then check that you have enabled billing for your project in the developer console.

You have a lot of details about the place, but wouldn't it be nice to also show a photo?

Getting a photo is not as simple as getting the basic place details, but armed with your newfound knowledge of result callbacks, you're up to the task!

You'll use the same callback pattern to get a photo for the selected place with a separate call to `fetchPhoto`. Once you retrieve the place in your success callback for `fetchPlace`, you can use the requested `PHOTO_METADATA` field in the place object to create a `FetchPhotoRequest` object and subsequently make another call to the `placesClient`. As you can see, this code can quickly become deeply nested and messy.

To avoid this, you're going to do a bit of clean up!

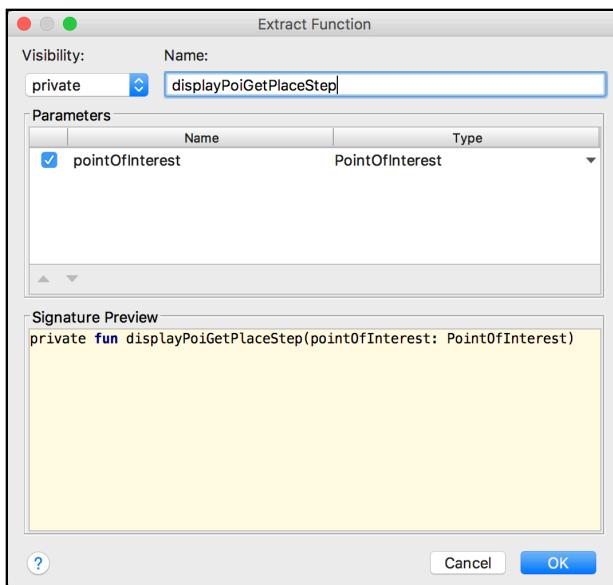
Refactoring in Android Studio

You'll place each main step in its own method to keep things nice and clean. You start by refactoring `displayPoi()` to kick off the first step. You take the code inside of `displayPoi()` and move it into a new method that takes a single argument. You then add a call to the new method inside `displayPoi()`. This is a 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 macOS 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'll need in the new method.
4. Click **OK**.

Voilà! The method is created, and the call is added to `displayPoi()`.



Your refactored code looks like this:

```
private fun displayPoi(pointOfInterest: PointOfInterest) {  
    displayPoiGetPlaceStep(pointOfInterest)  
}
```

```
private fun displayPoiGetPlaceStep(pointOfInterest: PointOfInterest) {
    val placeId = pointOfInterest.placeId

    val placeFields = listOf(Place.Field.ID,
        Place.Field.NAME,
        Place.Field.PHONE_NUMBER,
        Place.Field.PHOTO_METADATA,
        Place.Field.ADDRESS,
        Place.Field.LAT_LNG)

    val request = FetchPlaceRequest
        .builder(placeId, placeFields)
        .build()

    placesClient.fetchPlace(request)
        .addOnSuccessListener { response ->
            val place = response.place
            Toast.makeText(this,
                "${place.name}, " +
                    "${place.phoneNumber}",
                Toast.LENGTH_LONG).show()
        }.addOnFailureListener { exception ->
            if (exception is ApiException) {
                val statusCode = exception.statusCode
                Log.e(TAG,
                    "Place not found: " +
                        exception.message + ", " +
                    "statusCode: " + statusCode)
            }
        }
    }
}
```

Fetching a place photo

Now, you'll add a step to retrieve a photo using the place details you requested in the previous step.

Add the following new method to **MapsActivity**:

```
private fun displayPoiGetPhotoStep(place: Place) {
    // 1
    val photoMetadata = place
        .getPhotoMetadata()?.get(0)
    // 2
    if (photoMetadata == null) {
        // Next step here
        return
    }
    // 3
    val photoRequest = FetchPhotoRequest
        .builder(photoMetadata as PhotoMetadata)
        .setMaxWidth(resources.getDimensionPixelSize(
            R.dimen.default_image_width))
        .setMaxHeight(resources.getDimensionPixelSize(
            R.dimen.default_image_height))
}
```

```
.build()
// 4
placesClient.fetchPhoto(photoRequest)
    .addOnSuccessListener { fetchPhotoResponse ->
        val bitmap = fetchPhotoResponse.bitmap
        // Next step here
    }.addOnFailureListener { exception ->
        if (exception is ApiException) {
            val statusCode = exception.statusCode
            Log.e(TAG,
                "Place not found: " +
                exception.message + ", " +
                "statusCode: " + statusCode)
        }
    }
}
```

You use the following steps to get a photo for the selected place:

1. Get the first and only PhotoMetaData object from the retrieved photo metadata array for the selected place.
2. If there's no photo for the place, skip directly to the next step.
3. Then, you use the builder pattern again to create the FetchPhotoRequest. You pass the builder the photoMetaData, a maximum width and a maximum height for the retrieved image.
4. You call fetchPhoto passing in the photoRequest and let the callbacks handle the response from Place Photos service. If the response is successfully received, assign the photo to bitmap. Otherwise, check if an ApiException occurred and log an error.

In `displayPoiGetPhotoStep()`, you pass a maximum height and width to get a scaled-down version of the original photo. The image is scaled proportionally to match the smaller of the two dimensions.

There are two benefits to restricting the image height and width in the `photoRequest`.

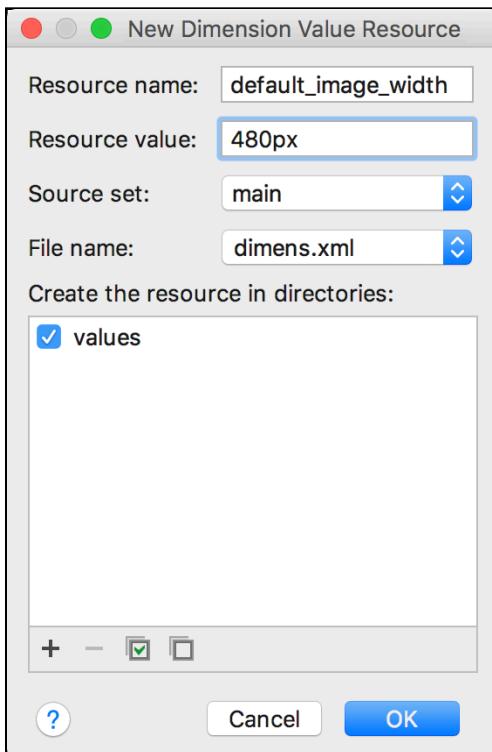
1. **Memory savings:** In general, you never want to load photos into memory that are larger than required. Here, you limit the possibility of memory issues that can happen on lower-end devices.
2. **Bandwidth savings:** Since `photoRequest` sends the maximum height and width in the API call, the scaling happens on the server-side and only the final scaled down version is sent to the device.

Now, replace the `Toast` call in `displayPoiGetPlaceStep()` with a call to this new method:

```
displayPoiGetPhotoStep(place)
```

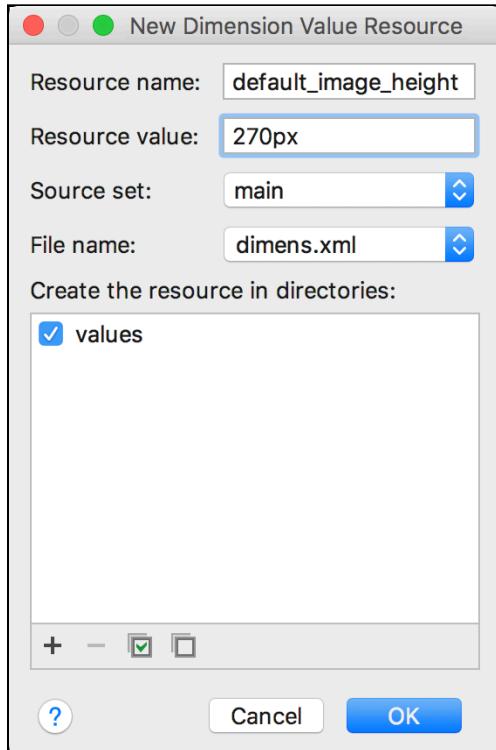
Next, fix the unresolved references for `R.dimen.default_image_width` and `R.dimen.default_image_height` that are displayed in red. To resolve the errors, follow these steps:

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 creates 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.

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 **dimens.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.

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 as LatLng)  
        .icon(iconPhoto)  
        .title(place.name)  
        .snippet(place.phoneNumber)  
    )  
}
```

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 displayPoiGetPhotoStep()` to call your new step:

```
displayPoiDisplayStep(place, null)
```

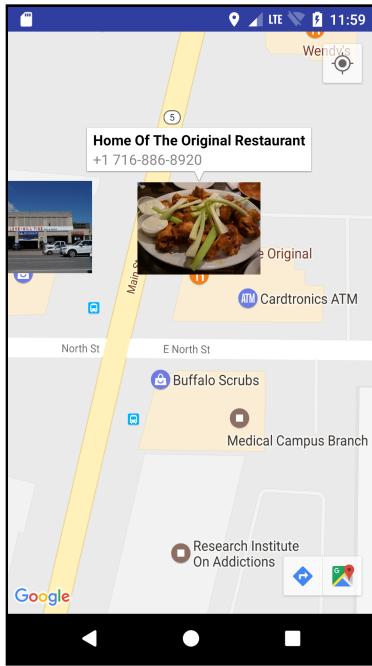
Here, you pass along the `place` object and a `null` bitmap image.

Replace the second commented line `// Next step here in displayPoiGetPhotoStep()` to call your new step:

```
displayPoiDisplayStep(place, bitmap)
```

Here, you pass along the `place` object and the bitmap image `bitmap`.

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 can 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 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 `InfoWindowAdapter`:

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 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:adjustViewBounds="true"
        android:maxWidth="200dp"
        android:scaleType="fitStart"
        android:contentDescription="@string/bookmark_image"
        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 `InfoWindowAdapter` and populate the `ImageView` and both `TextViews`.

Open `res/values/strings.xml` and add the following content description string for the bookmark image view.

```
<string name="bookmark_image">Bookmark image</string>
```

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:

```
// 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
}
```

Here's what's happening:

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're 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:

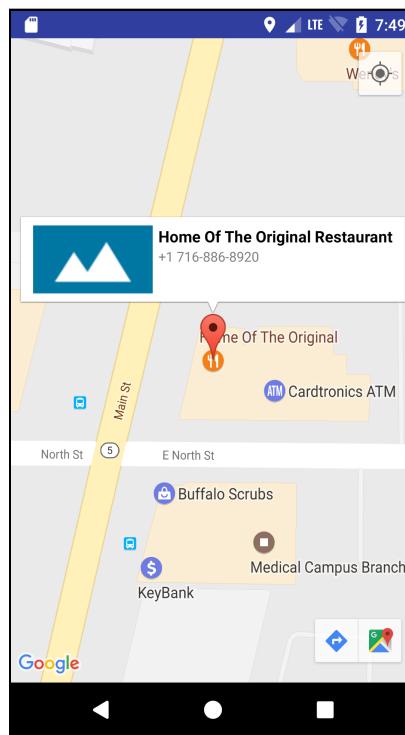
```
map.setInfoWindowAdapter(BookmarkInfoWindowAdapter(this))
```

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 as LatLng)
    .title(place.name)
    .snippet(place.phoneNumber)
)
```

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.



You'll finish off the `BookmarkInfoWindowAdapter` by adding the place image.

Marker tags

So, how do you associate the image with the marker? There are 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 as LatLng)
    .title(place.name)
    .snippet(place.phoneNumber)

)
marker?.tag = photo
```

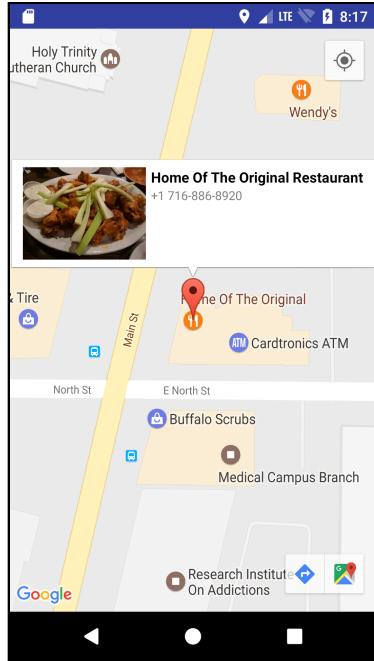
Here, `addMarker()` returns a Marker object and you assign it to `marker`. You then assign `photo` to the tag property.

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?

Pat yourself 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 update the View automatically.

Getting started

If you were following along with your own app, open it, and keep using it with 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 `google_maps_api.xml` and in the method `setupPlacesClient()` in `MapsActivity.kt`. Read 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 its dependencies.

In ListMaker, you used **Shared Preferences** to store data permanently. 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:

- Works directly with **Plain Java Objects** (POJOs) with minimal effort.
- Provides advanced search and sorting through SQL queries.
- Manages relationships between different data types.
- Efficiently stores large amounts of data.

Room overview

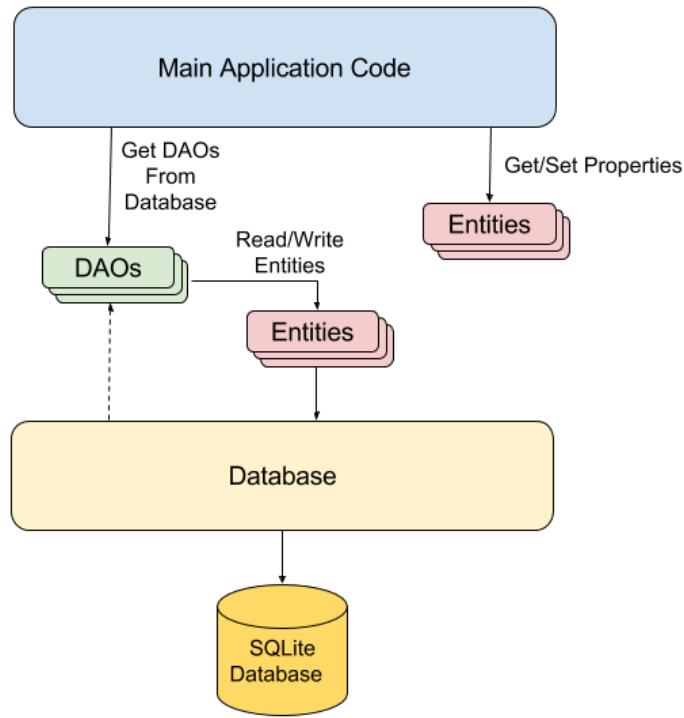
Before diving into the code, it's important to understand the three basic components of Room.

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 creates 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 DAO annotated interface.

The following diagram illustrates how these three components fit into PlaceBook.



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:

- **Lifecycle management**: Provides several classes to help build lifecycle-aware objects.
- **LiveData**: Holds data that can be observed for changes and respects lifecycles.
- **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'll be covered in more detail as you build out the app.

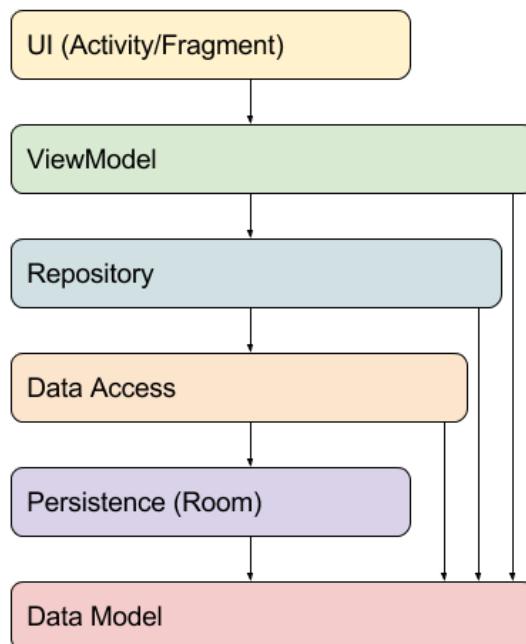
PlaceBook architecture

Before creating your first Room classes, you need to organize the app to achieve a clean overall architecture. You can separate the app into distinct areas of responsibility along these lines:

- Data access and persistence (Room).
- Data model (Model).
- Data abstraction (Repository).
- Business/Domain logic (ViewModel).
- User interface (Activity/Fragments).

One key goal is to ensure that communication only flows in one direction between these layers. This will result in a loosely coupled architecture that's easy to modify without causing side effects.

The overall architecture will look like this:



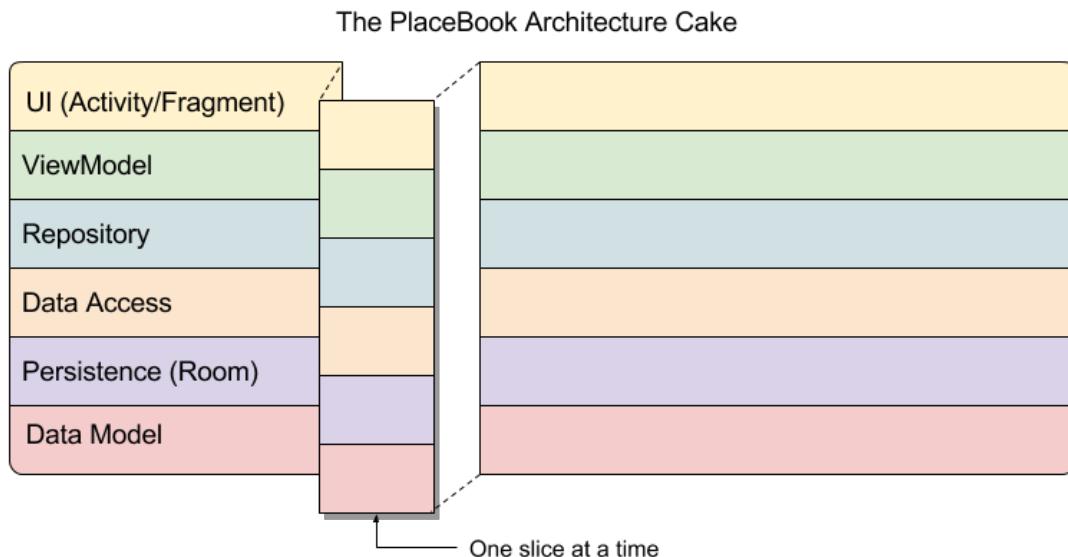
The arrows represent lines of communication and visibility. Notice that the UI layer is completely independent of all other layers except for the ViewModel. The ViewModel layer knows nothing about the UI layer.

As the rest of the app is built out, you'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:

- The way you store data in Room can be completely replaced with minimal impact. The only layers affected are the **Persistence** layer itself and its immediate parent, the **Data Access** layer.
- The UI layer can be fully replaced without any other layer being any the wiser.
- You can easily test all of the layers without any active UI running.

Development approach

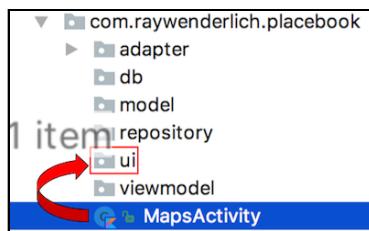
Think about the architecture as a multi-layered cake. Have you ever seen somebody eat a cake one layer at a time? That would be a little odd! Likewise, you're not going to build out the app one layer at a time. You're going to take one slice at a time. Each slice may cut through all of the layers as you slowly build out the final product.



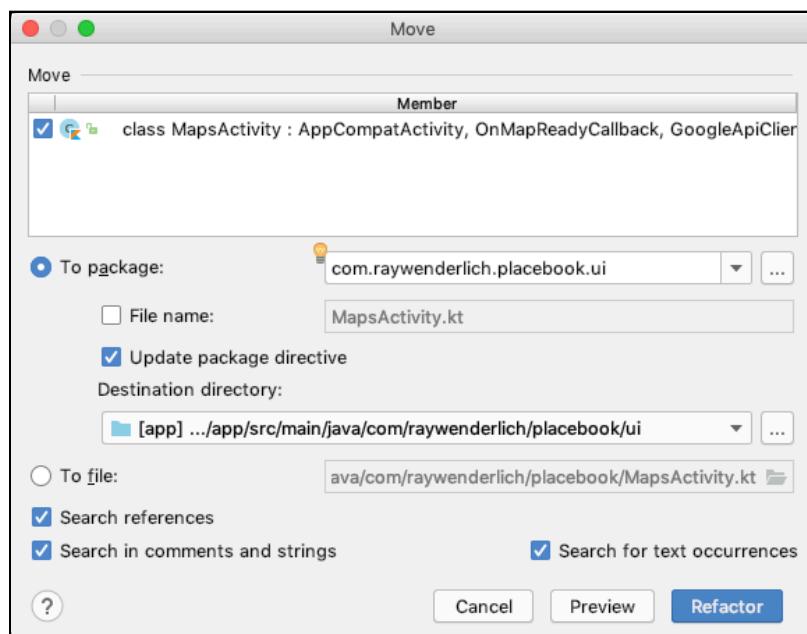
In the Project navigator, click **java/com.raywenderlich.placebook** and select **File > New > Package** to create the following packages. This will help organize the project to match the architecture:

- **db**: Data access and persistence. You'll keep the **Room Database** and **DAO** objects here.
- **model**: Model objects. This includes all **Room Entities** as POJOs.
- **repository**: Data abstraction. This provides a layer of abstraction for all data access.
- **ui**: User interface. All Views and View control logic belong here.
- **viewmodel**: Business/Domain logic. Contains ViewModel classes that drive the user interface and app logic.

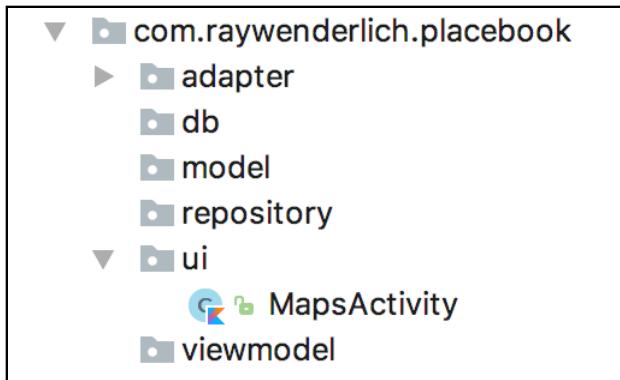
In the Project navigator, drag the **MapsActivity** class from the root package to the **ui** package.



Accept the default settings from the dialog and click **Refactor**.



The 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 = '1.1.1'  
room_version = '1.1.1'
```

It's time to 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.lifecycle:compiler:  
$architecture_version"  
// 4  
kapt "android.arch.persistence.room:compiler:$room_version"
```

Let's go through the above dependencies:

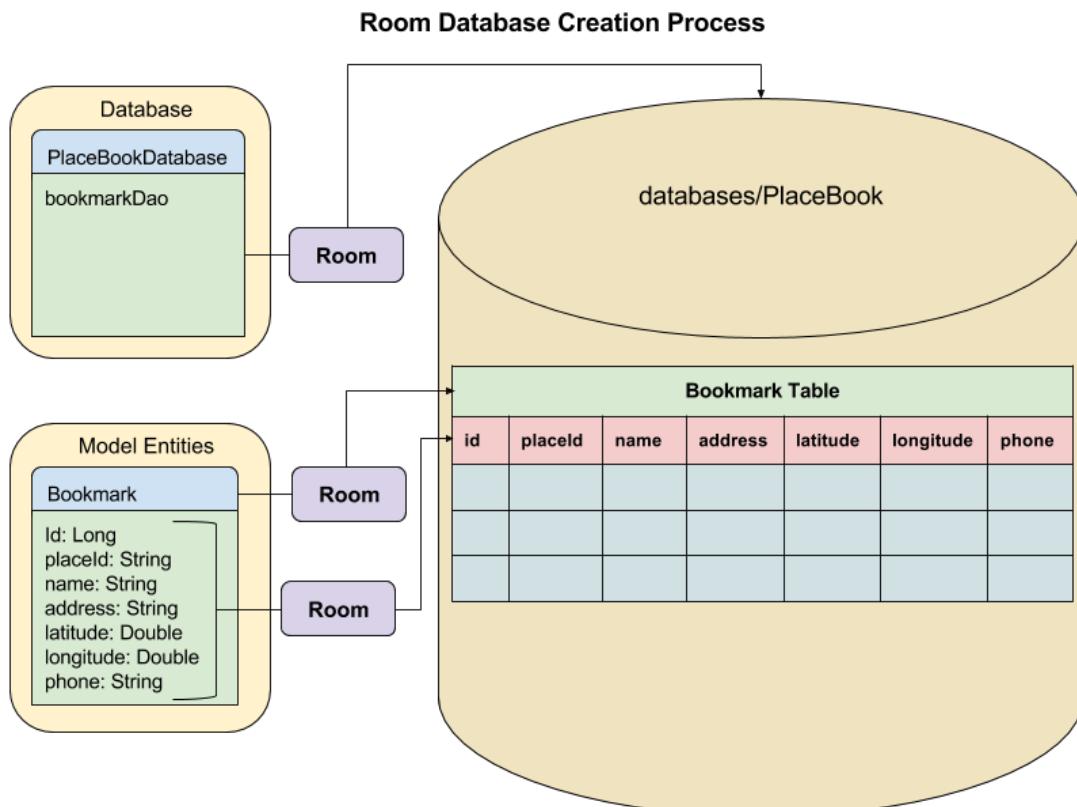
1. Adds the main Lifecycle classes along with extensions such as LiveData.
2. Adds the Room library.

3. Adds the annotation processor for the lifecycle classes.
4. Adds the Kotlin annotation processor for the Room library.

Room classes

Now you're ready to add the basic classes required by Room. This includes the Entities, DAOs, and the Database. Behind the scenes, Room takes your class structure and does all of the hard work to create an SQLite database with tables and column definitions.

Room names the database `PlaceBookDatabase`, and the model class `Bookmark`. The following diagram will help visualize the process that Room uses to convert your classes into the underlying database:



Entities

PlaceBook 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 = ""
)
```

Here's what's going on in the code above:

1. The `@Entity` annotation tells Room that this is a database entity class.

Note: Although not used in this example, 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's 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 looks for arguments on the constructor and class properties when defining fields for the table. In this case, you're only using properties to define the table fields.

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.

In database terminology, this would be considered a surrogate or synthetic key and provides a unique identifier for each Bookmark record.

4. The rest of the fields are defined with default values.

When creating the new class in **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 = :bookmarkId")
    fun loadBookmark(bookmarkId: Long): Bookmark

    @Query("SELECT * FROM Bookmark WHERE id = :bookmarkId")
    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 macOS or **Ctrl-Enter** on Windows, and select the `android.arch.persistence.room.OnConflictStrategy.IGNORE` option — you may have to add this import manually.

Place the cursor on REPLACE and press **Option-Return** on macOS or **Ctrl-Enter** on Windows and select the `android.arch.persistence.room.OnConflictStrategy.REPLACE` option.

`BookmarkDao` defines what would traditionally be known as **CRUD** database operations. The CRUD operations consist 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:

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 an SQL statement to read all of the 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 a lot of SQL to build out PlaceBook. If you want to learn more about SQL, and specifically the syntax used for SQLite, read <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` notifies 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. This method returns a single `Bookmark` object. Here the `@Query` annotation is used to tell Room how to retrieve a single `Bookmark`. This method loads a `Bookmark` based on the `bookmarkId`. To do the actual database query, Room takes the arguments passed into your method and replaces the matching `:?` strings in the query, where `?` matches an argument name on the method. In this case, `:bookmarkId` is replaced with the value of the `bookmarkId` argument passed into `loadBookmark()`.

You also define an asynchronous version named `loadLiveBookmark` that returns a `LiveData` wrapper around a single `Bookmark`.

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 is 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:

```
// 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
        }
    }
}
```

Here's how this code works:

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. This database will store a single entity type of `Bookmark`. If you were storing multiple entity types, they would be separated by commas inside the `arrayOf` construct.

Room requires your database class to 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. You are declaring this as abstract because Room takes care of implementing the actual `BookmarkDao` class for you based on the `BookmarkDao` interface you defined earlier.

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 results in a compile error that reads “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 exposes 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 IV makes full use of the Repository pattern.

To manage your bookmarks, you'll create a single repository class named `BookmarkRepo`. This class will internally use `BookmarkDao` from `PlaceBookDatabase` to access the underlying bookmarks in the database. It will define some basic methods for saving and loading bookmarks.

Create a Kotlin file named `BookmarkRepo.kt` in the `repository` package, and replace the contents with the following:

```
// 1
class BookmarkRepo(context: Context) {
    // 2
    private var db = PlaceBookDatabase.getInstance(context)
    private var bookmarkDao: BookmarkDao = db.bookmarkDao()
    // 3
    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()
        }
}
```

Here's the code breakdown:

1. Define the `BookmarkRepo` class with a constructor that passes in an object named `context`. A `Context` object is required to get an instance of the `PlaceBookDatabase` class.
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, although none is required in this case.

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.

You'll see how this class is used in detail as you build out the `ViewModel`.

The ViewModel

The **ViewModel** layer serves as the intermediary between your app Views and the data provided by the **BookmarkRepo**. The `ViewModel` drives the UI based on the repository data and updates the repository data based on user interactions.

You'll typically have one `ViewModel` for each View (Activity or Fragment) in your app. The naming convention used for `ViewModel` classes is to simply append `ViewModel` to the View class prefix. Your first View model will be used to manage the `MapsActivity` View.

Create a Kotlin file named **MapsViewModel.kt** in the `viewmodel` package to go along with the **MapsActivity**. Replace the contents with the following:

```
// 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 ?: 0.0
        bookmark.latitude = place.latLng?.latitude ?: 0.0
        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.")
    }
}
```

Here's what's happening:

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, passing in the application context. `getApplication()` is provided by the base `AndroidViewModel` class.
3. Declare the method `addBookmarkFromPlace` that takes in a Google Place and a `Bitmap` image. This will be called by the `MapsActivity` when it wants to create a bookmark for a Google Place that has been identified by the user.
4. Use `BookmarkRepo.createBookmark()` to create an empty `Bookmark` object and then fill it in using the Place data. If the `latLng` property is `null`, you use the `?:` operator to set the `longitude` and `latitude` values to `0.0`.
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 before `onCreate()`.

```
private lateinit var mapsViewModel: MapsViewModel
```

You're declaring a private member to hold the `MapsViewModel`. This is initialized when the map is ready.

Add a new private method to `MapsActivity` 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` creates a new `mapsViewModel` only the first time the Activity is created. If a configuration change happens, such as a screen rotation, `ViewModelProviders` returns 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. You can solve this by creating a private class to hold both pieces of information.

Add the following internal class to the bottom of the `MapsActivity` class before the final closing `}`:

```
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 "`marker?.tag = photo`" 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**, in `getInfoContents()`, update the line that calls `setImageBitmap` 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) {  
        mapsViewModel.addBookmarkFromPlace(placeInfo.place,  
            placeInfo.image)  
    }  
    marker.remove()  
}
```

This method handles 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()`:

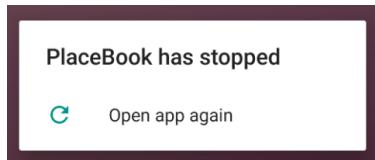
```
map.setOnInfoWindowClickListener {  
    handleInfoWindowClick(it)  
}
```

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 calls `handleInfoWindowClick()` which in turn calls `mapsViewModel.addBookmarkFromPlace()`, and adds a bookmark to the database.

Build and run the app.

Tap on a place so that 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 you get the following exception:

```
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 thrown on the call to `addBookmarkFromPlace()` and as the message explains that 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 to 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 launch (or start) a coroutine. Coroutines are always started in the context of a **CoroutineScope**. The CoroutineScope defines the lifetime of the coroutine. Kotlin provides a **GlobalScope** context that applies to the lifetime of the whole application. When launching a coroutine, you provide a block of code known as a suspending lambda expression. The **GlobalScope** context 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 in the **GlobalScope**.

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:1.1.0"
```

Note: After making the above change to the gradle file, don't forget to click **Sync Now** so that Gradle loads the new dependencies.

Creating a Coroutine

Open **MapsActivity.kt** and replace the call to `addBookmarkFromPlace` in `handleInfoWindowClick()` with the following:

```
GlobalScope.launch {
    mapsViewModel.addBookmarkFromPlace(placeInfo.place,
        placeInfo.image)
}
```

You use the `launch` coroutine builder to launch a coroutine in the `GlobalScope`. The `GlobalScope` 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; 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

Remember that `MapsViewModel` is used to model the View seen by the user. You want to show the user a marker at each saved bookmark location, so you'll create a class in `MapsViewModel` to hold the data for each visible bookmark marker.

Add the following internal class to `MapsViewModel.kt` before the final }:

```
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.

You can now store a list of these bookmark Views. Add the following property at the top of `MapsViewModel.kt` inside the class definition:

```
private var bookmarks: LiveData<List<BookmarkMarkerView>>?  
    = null
```

Here, you're defining a `LiveData` object that wraps a list of `BookmarkMarkerView` objects.

As mentioned earlier, `LiveData` is an observable data holder class provided as part of the Android Architecture Components. When another object observes the `LiveData` object, it will be notified when any data maintained by the `LiveData` is changed. You'll see how to observe the `LiveData` object in the next section.

Now that you have an object to store the bookmark marker views, you need to populate them from the bookmarks stored in the database. This is done by reading the bookmarks from the bookmark repo and converting them into `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 from the repo into a `BookmarkMarkerView` object. This is used by the next method.

Now, add the following method:

```
private fun mapBookmarksToMarkerView() {  
    // 1  
    bookmarks = Transformations.map(bookmarkRepo.allBookmarks)  
    { repoBookmarks ->  
        // 2  
        repoBookmarks.map { bookmark ->  
            bookmarkToMarkerView(bookmark)  
        }  
    }  
}
```

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`, it's best not to expose `MapsActivity` to the details of the underlying `Bookmark` object.

1. Use the `Transformations` class to dynamically map `Bookmark` objects into `BookmarkMarkerView` objects as they get updated in the database. `Transformations` is provided by the `Lifecycle` package as a convenient way to transform values in a `LiveData` object before they are returned to the observer.

`Transformations.map` takes in an argument for a `LiveData` object and returns the transformed `LiveData` object. It's your job to define the mapping method that converts from one data type to another. This mapping method is described in Step 2 below.

2. `Transformations.map` provides you with a list of Bookmarks returned from the bookmark repo. You store these in the `repoBookmarks` variable.

Keep in mind that this is a dynamic function and will get called anytime the bookmark data changes in the database.

You take the `repoBookmarks` list provided by the `Transformations.map` function and convert them into `BookmarkMarkerViews`. You do this by using the `map` function on the `repoBookmarks` list. Using `map` on a list is an easy way to convert a list of objects from one type to another.

The class property `bookmarks` is assigned to the result of `Transformations.map`. This results in the `bookmarks` property sending out notifications to any observers when any data changes in the Bookmarks table.

To finish up this class, you need a method to initialize and return the bookmark marker views to the `MapsActivity`.

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` are `null` the first time this method is called. If it's `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. First, you need a method to add a bookmark marker to the map.

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.

Next, you'll need a method to display all of the bookmark markers. 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.

Next, you'll create a method that observes the changes to the bookmark marker views in the maps View model.

Add the following method to **MapsActivity.kt**:

```
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 `LiveData` object, you call the `observe` method. The first argument is `this`, and it represents the LifeCycle Owner. You're telling the observer to follow the lifecycle of the current activity.

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 as represented by the `it` variable.

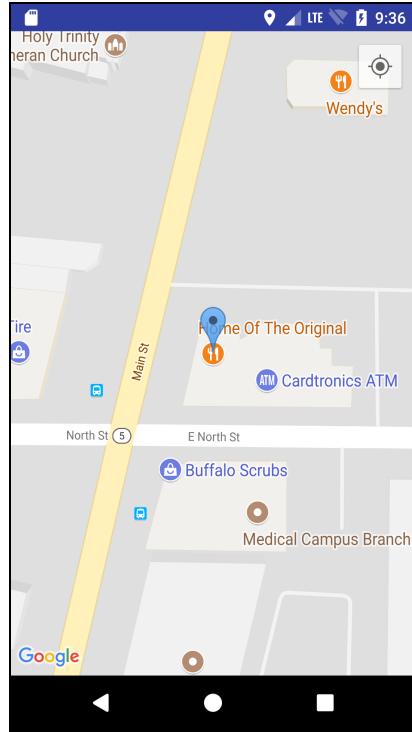
The only item left is to call `createBookmarkMarkerObserver()` when setting up the model View.

Add the following line to the end of `setupViewModel()`:

```
createBookmarkMarkerObserver()
```

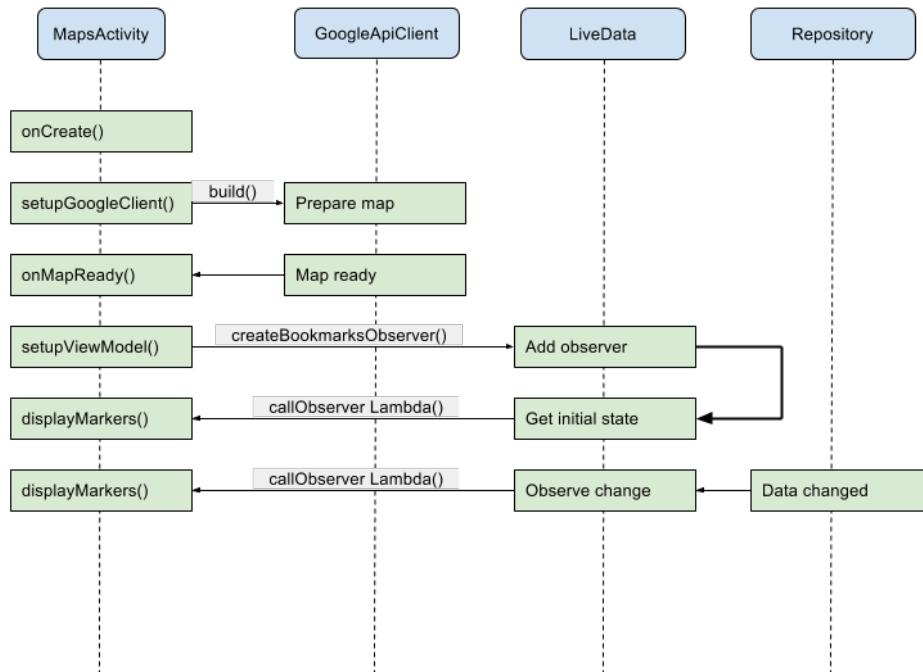
Build and run the app. If you previously added some places to the database by tapping on the info windows, you'll see blue markers appear on the map.

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 happens 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 calls your observer immediately with the current set of data. From then on, the observer is notified anytime the underlying data changes.

Where to go from here?

There's one problem with this new implementation: If you tap on any of the blue markers, the app will crash. Can you guess why? If not, don't worry! You'll fix this crash in the next chapter, and you'll add some new features to `MapsActivity`, giving 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 involves creating a new Activity to display the bookmark details with editable fields.

Getting started

If you're following along with your own app, open it and copy **res/drawable/ic_action_done.png** from the **starter** project into your project. Also, make sure to copy the files from all of the drawable folders, including everything with the **.hdpi**, **.mdpi**, **.xhdpi** and **.xxhdpi** extensions.

If you want to use the starter project instead, locate the **projects** folder for this chapter and open the **PlaceBook** app inside the **starter** folder. If you do use the starter app, don't forget to add your **google_maps_key** in **google_maps_api.xml** and in the method **setupPlacesClient()** in **MapsActivity.kt**. Read 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 its dependencies.

Fixing the info window

Before moving on, you need to track down and fix that pesky bug left over from the previous chapter where the app crashes when tapping on a blue marker. The desired behavior is:

- 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, you save a bookmark 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, look at Logcat. Find for the most recent stack trace line that has your app's package name, and you'll find the following line:

```
com.raywenderlich.placebook.adapter.BookmarkInfoWindowAdapter.getInfoContents
```

```
java.lang.ClassCastException: com.raywenderlich.placebook.viewmodel.MapsViewModel$BookmarkMarkerView cannot be cast to  
at com.raywenderlich.placebook.adapter.BookmarkInfoWindowAdapter.getInfoContents(BookmarkInfoWindowAdapter.kt:33)
```

This error is a **ClassCastException** error informing you that a `BookmarkMarkerView` cannot be cast to a `MapActivityPlaceInfo` class.

To find out what's going on, click the blue link for **BookmarkAdapter.kt**, which takes 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`. However, that's not always the case because a marker can now represent two types of places: one is a temporary place that isn't bookmarked yet, and the other is a place that has an existing bookmark.

To fix this, you need to 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 haven't saved images with the bookmarks yet.

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's linked to the record in the database.

Android doesn't provide a simple way to save images to a file, so you first need to create a new image utility class, and add a method to save an image to a file.

In the Project navigator, click **java/com.raywenderlich.placebook**, select **File > New > Package** and create a new package named **util**. Inside **util**, create a new Kotlin class named **ImageUtils.kt**.

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()
    }
}
```

Here's the code breakdown:

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 parameter 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 `saveBitmapToFile()` set up, 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 `Bookmark.kt` inside `model` and add the following code to the bottom of the file:

```
{  
    // 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"  
        }  
    }  
}
```

Previously `Bookmark` was a simple data class with only default functions provided by Kotlin. This adds a body to the class.

1. `setImage()` provides the public interface for saving an image for a `Bookmark`.
2. If the bookmark has an `id`, then the image gets 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's 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 appends the bookmark ID to the word “bookmark”. For example, for bookmark ID 5, the associated image is named `bookmark5.png`. Since you can always infer the bookmark image filename from the bookmark ID, there's no need to save the filename as a separate field in the database.

Adding the image to the bookmark

Next, 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 `setBookmarkRepo.addBookmark()`.

```
image?.let { bookmark.setImage(it, getApplication()) }
```

Here, you update `addBookmarkFromPlace()` to call the new `setImage()` method if the image is not `null`.

It's important to call this after the bookmark is saved to the database so that the bookmark has a unique ID assigned.

`setImage()` is used to save the image to the bookmark, and 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 automatically displays 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.

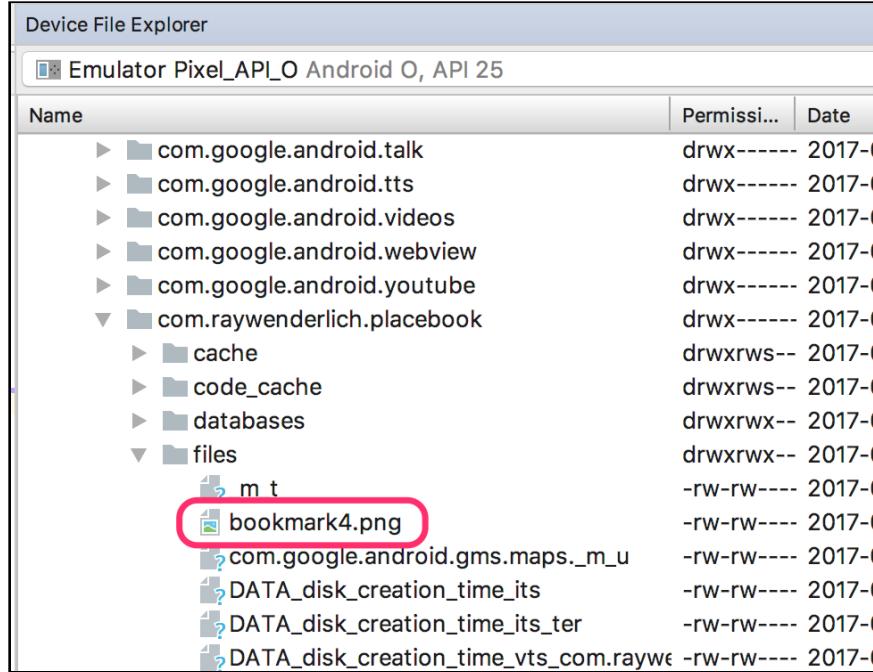
Build and run the app. Tap on a new place to display a marker and the Info window. Then, tap on the Info window, which triggers a new bookmark for saving — and this time it stores the bitmap image to a file.

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 the **Device File Explorer** tool on the right side of the Android Studio window. If you don't see it, 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

It's time to load the image from a file. This is considerably easier than saving an image because Android provides a method on **BitmapFactory** 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 can load the image from where it's stored, it's time to 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 replace the `BookmarkMarkerView` data class definition with the following:

```
data class BookmarkMarkerView(var id: Long? = null,
                               var location: LatLng = LatLng(0.0, 0.0))
{
    fun getImage(context: Context): Bitmap? {
        id?.let {
            return ImageUtils.loadBitmapFromFile(context,
                Bookmark.generateImageFilename(it))
        }
        return null
    }
}
```

Previously, `BookmarkMarkerView` was a simple data class with only default functions provided by Kotlin. This adds a body to the class with the new `getImage` function.

In `getImage()`, you first check to make sure the `BookmarkMarkerView` has a valid ID. Then, you call `generateImageFilename()` and pass in the bookmark ID represented as `id`. You call `loadBitmapFromFile()` with the current context and `Bookmark` image filename, and it returns the resulting `Bitmap` to the caller.

You need to update the Info window Adapter to load the image when it's done rendering. 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 change the constructor to the following:

```
class BookmarkInfoWindowAdapter(val context: Activity) :
    GoogleMap.InfoWindowAdapter {
```

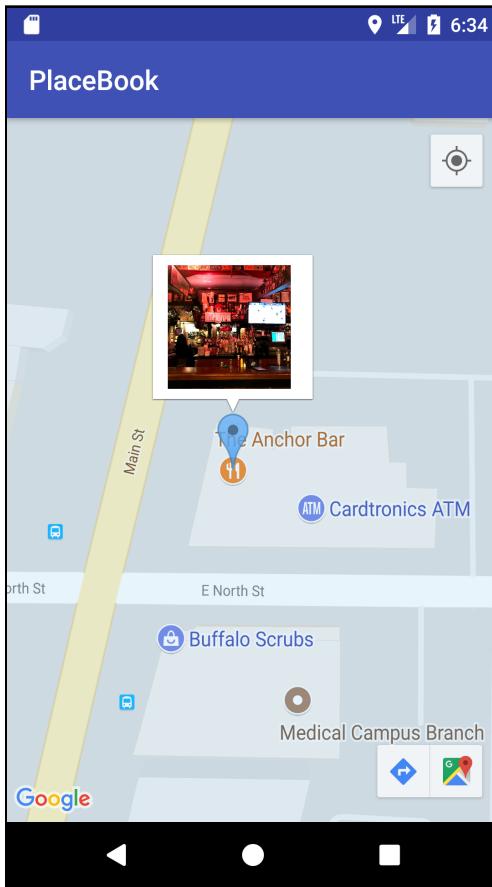
The only difference is the addition of the `val` modifier. This makes `context` a property so you can use it later to load the image.

Add the following code in `getInfoContents()` after the comment `// Set imageView bitmap here:`

```
imageView.setImageBitmap(bookMarkview.getImage(context))
```

Build and run the app. Tap on a blue marker for a bookmark that was saved after you added the ability to save images.

This displays the Info window with the bookmark image.



The image is showing, but there's no bookmark information along with it. You can fix this 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 `BookmarkMarkerView`.

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)  
}
```

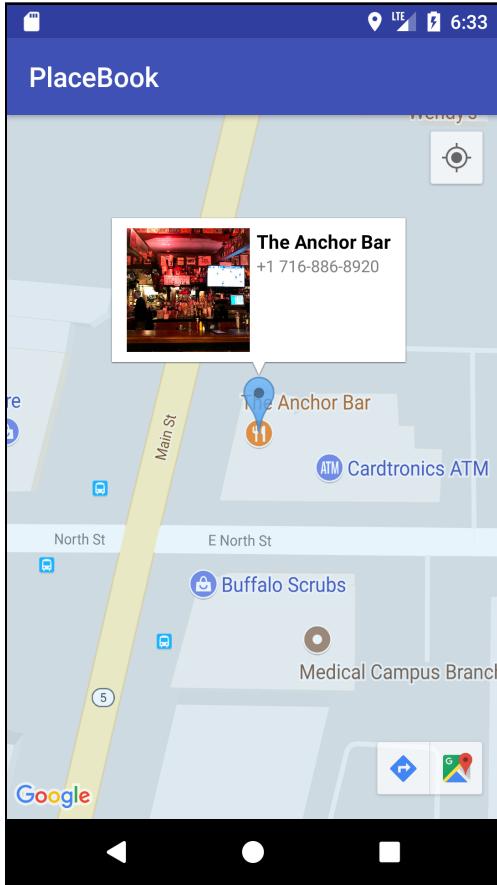
The only change is that the bookmark name and phone properties are passed into the new `BookmarkMarkerView` constructor.

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, notice it displays the name and phone number beside the image.



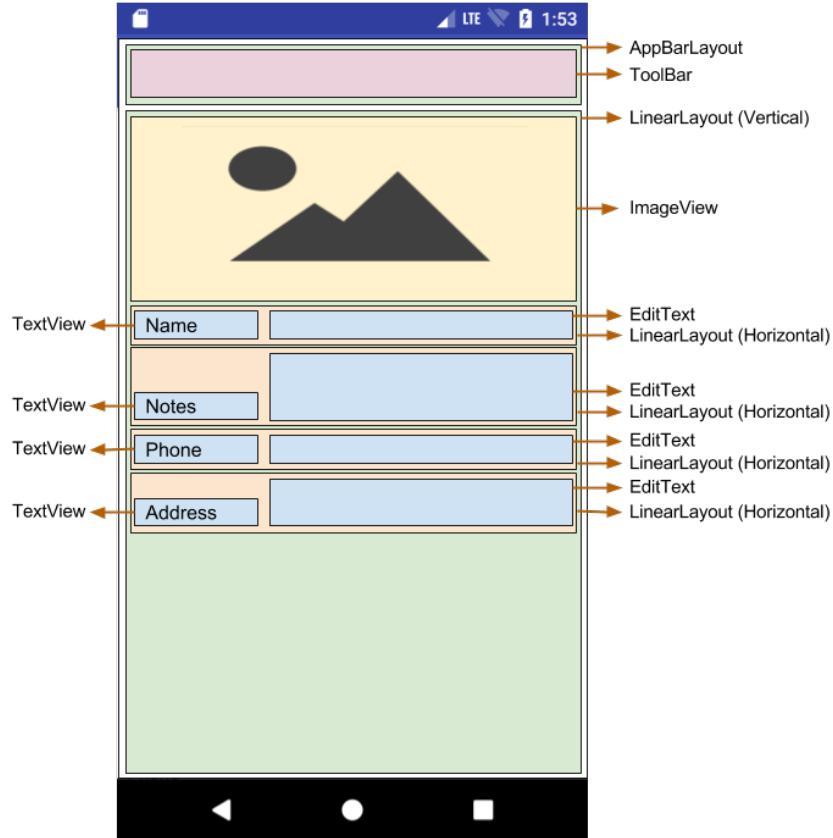
If you tap on the Info window, the app will most likely crash. You'll fix that soon!

Bookmark detail activity

You've waited patiently, and it's finally time to build out the detail Activity for editing a bookmark. For that, 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 displays 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 that the label takes 20% of the Layout width.

Defining styles

First, you need to define some standard styles that are required when using the support library version of the toolbar.

Add the following to **res/values/styles.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 is used to hide the native `ActionBar`. `AppBarOverlay` gives the Toolbar Layout a dark theme, and `PopupOverlay` gives 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 some styles that you can apply 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**:

```
<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 need to create the bookmark details Layout based on the design. The Activity will use all of the new styles you 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"/>

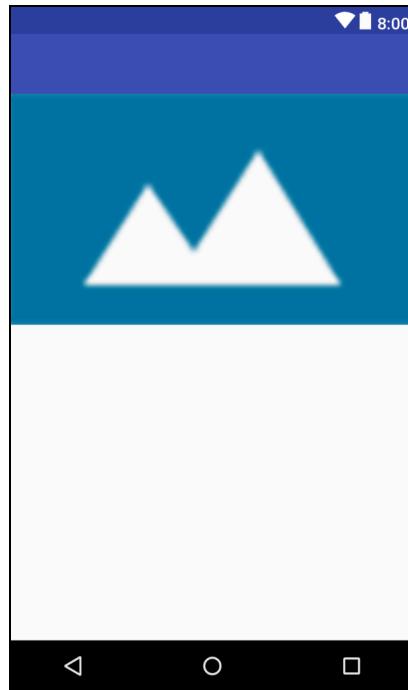
    </android.support.design.widget.AppBarLayout>

    <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 need to 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.

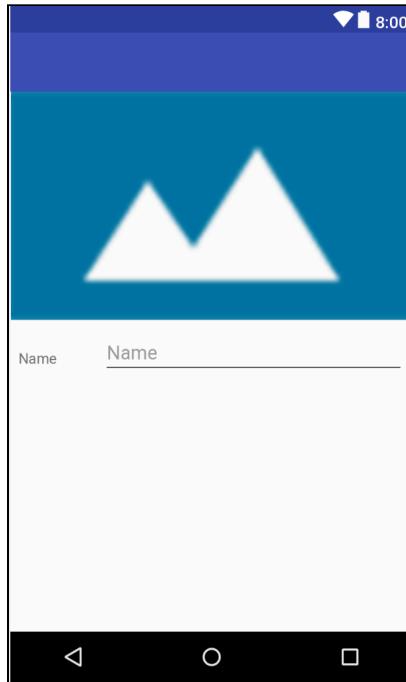
First, add a row for the bookmark name by adding the following code below the `<ImageView>` element:

```
<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 by adding 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 by adding 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>
```

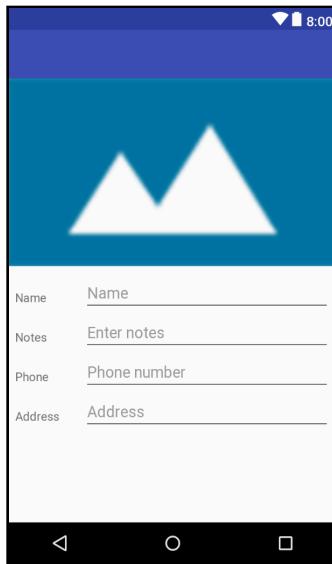
Next, add a row for the bookmark address by adding the following code after the bookmark phone number row:

```
<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.

Inside **ui**, create a new Kotlin file named **BookmarkDetailsActivity.kt**, and replace the contents with the following:

```
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. By using the support library version of the Toolbar, you ensure that it works the same across a variety of devices.

Note: Because the app **build.gradle** file contains the `kotlin-android-extensions` plugin, Android Studio automatically recognizes 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

To use `setSupportActionBar()` you need to include the support design library provided by Google.

Add the following line to the `buildscript.ext` section in the project **build.gradle** file.

```
support_lib_version = '28.0.0'
```

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.

Multidex support

As you add new libraries to your Android project, they each increase the size of your final app. Android app (APK) files are made up of **Dalvik Executable** (DEX) files. Each DEX file has a limit of 64K (65,536) methods.

That may seem like a high limit, but many of the support libraries bring in thousands of methods each, and you can pass the 64K before you know it. If you pass this limit, Android Studio won't be able to build and run your app.

If you build the project after adding the design support library, you may see an error similar to the following:

```
Error: null, Cannot fit requested classes in a single dex file (# methods: 68273 > 65536)
```

Fortunately, the fix is simple! You need to tell the build system to use something called **Multidex** support. This works behind the scenes to split your application into multiple DEX files to get past the 64K per DEX limit.

In the app **build.gradle** file, add the following line to the `defaultConfig` section:

```
multiDexEnabled true
```

A warning about changing Gradle files is shown at the top of the editor. Click **Sync Now**.

Rebuild the app again, and the DEX error will disappear.

Note: You can read all of the gory details about Android APK and DEX files at the following link: <https://developer.android.com/studio/build/multidex>.

Updating the manifest

Next, you need to make Android aware of the new `BookmarkDetailsActivity` class, so 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

You can now 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`:

```
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.

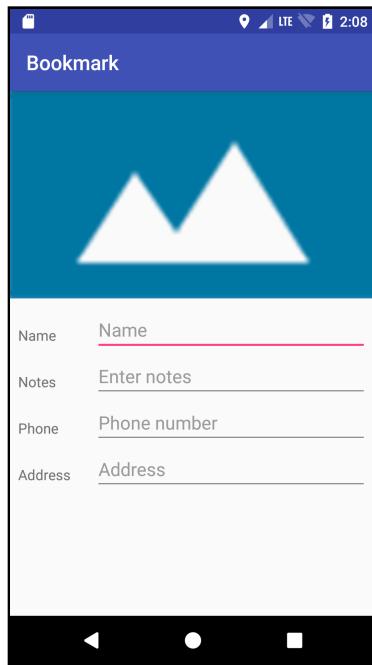
Replace `handleInfoWindowClick()` with the following:

```
private fun handleInfoWindowClick(marker: Marker) {
    when (marker.tag) {
        is MapsActivity.PlaceInfo -> {
            val placeInfo = (marker.tag as PlaceInfo)
            if (placeInfo.place != null && placeInfo.image != null) {
                GlobalScope.launch {
                    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)
            }
        }
    }
}
```

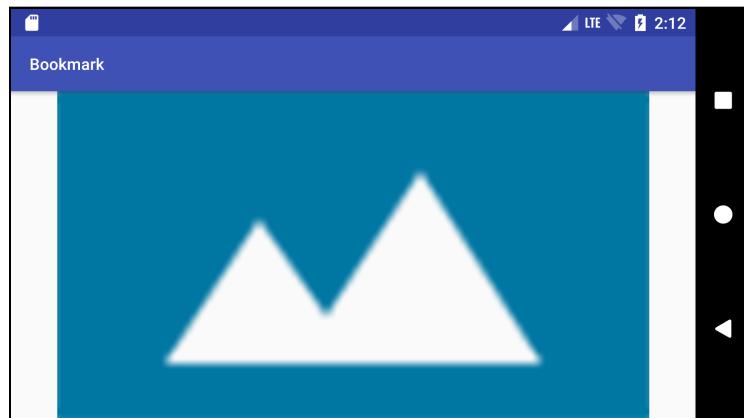
This method handles the action when a user taps a place Info window. Previously, it was designed to save the bookmark to the database. Now, it saves the bookmark if it hasn't been saved before, or it starts the bookmark details Activity if the bookmark has already been saved.

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 is 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. You can easily fix this by surrounding the main content with a **ScrollView**.

Open `activity_bookmark_details.xml` and after `</android.support.design.widget.AppBarLayout>`, add this:

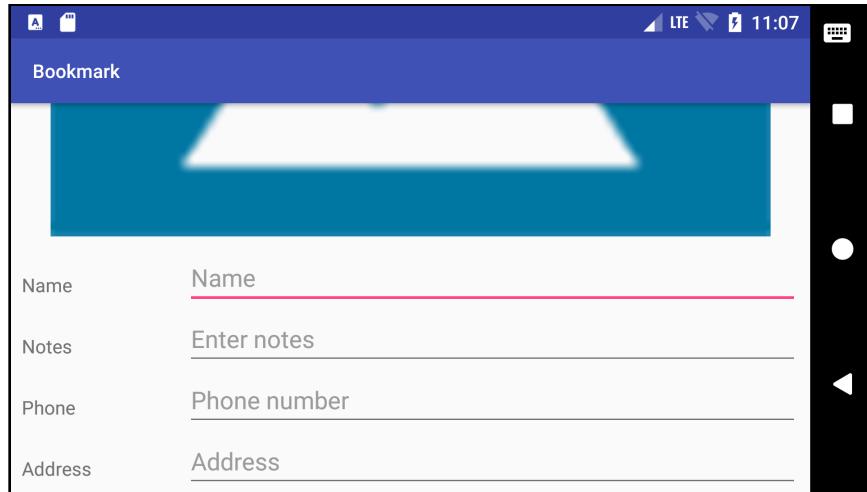
```
<ScrollView  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
  
    <LinearLayout  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:orientation="vertical">
```

Now, 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 lacks any knowledge about the bookmark. To fix this, you'll pass the bookmark ID to the Activity so that 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.

In `startBookmarkDetails()`, add the following line before the call to `startActivity()`:

```
intent.putExtra(EXTRA_BOOKMARK_ID, bookmarkId)
```

This adds the `bookmarkId` as an extra parameter on the Intent.

Next, you need to 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` uses a `ViewModel` to coordinate the data between the View and the Model.

You 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.

In **viewmodel**, create a new Kotlin file named **BookmarkDetailsViewModel.kt** and replace the contents with the following:

```
class BookmarkDetailsViewModel(application: Application) :  
    AndroidViewModel(application) {  
  
    private var bookmarkRepo: BookmarkRepo =  
        BookmarkRepo(getApplicationContext())  
}
```

`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. You can repeat this pattern anytime you need to return live data for a View; 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**:

```
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  
    }  
}
```

`BookmarkDetailsView` defines the data needed by `BookmarkDetailsActivity.getImage()` loads the image associated with the bookmark.

Adding notes to the database

Before continuing, you 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:

```
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:

```
@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 tries to migrate data from the old structure to the new structure. It does so by looking for **Migrations** that you have added to the database builder. If you haven't added any Migrations, then an exception is thrown, and the app crashes.

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's `getInstance()`, 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 and 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 a Bookmark model to a BookmarkDetailsView model. Now, you 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 allows 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 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) { repoBookmark ->
        bookmarkToBookmarkView(repoBookmark)
    }
}
```

Here, you get the live Bookmark from the BookmarkRepo and then transform it to the live `BookmarkDetailsView`. See the previous chapter for details about how `Transformations.map()` works.

Finally, you can bring it all together by exposing a method to return a live bookmark View based on a bookmark ID. 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 in the View Activity.

First, you 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` provided 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 passes the bookmark ID to the details Activity.

You now need to add a method in **BookmarkDetailsActivity** 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 is called when the Activity is created. Here's how it works:

1. You pull the `bookmarkId` from the Intent data.
2. You retrieve the `BookmarkDetailsView` from `BookmarkDetailsViewModel` and then observe it for changes.
3. Whenever the `BookmarkDetailsView` is loaded or changed, you assign the `bookmarkDetailsView` property to it, and populate the bookmark fields from the data. You call the previously defined functions to populate the fields.

Finishing the detail activity

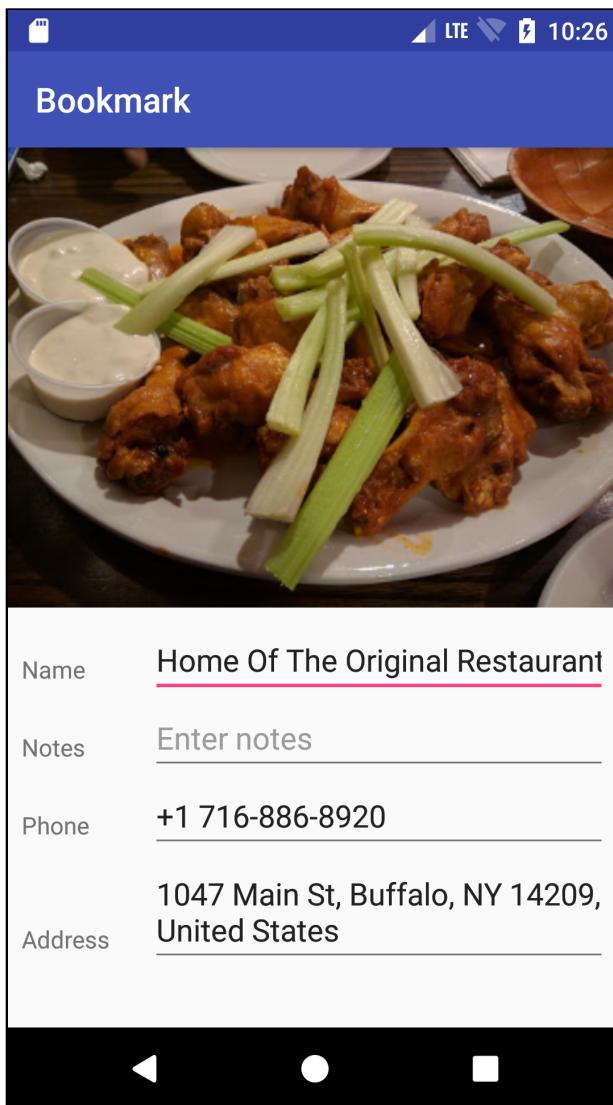
You're ready to pull everything together by adding the following calls to the end of `onCreate()` in **BookmarkDetailsActivity**.

```
setupViewModel()  
getIntentData()
```

When the bookmark details Activity starts, it creates the view model and process the Intent data passed in from the maps Activity.

Build and run the app. The previous data is cleared out because of the database schema change.

Add a new bookmark and view the details. The bookmark info is now displayed.

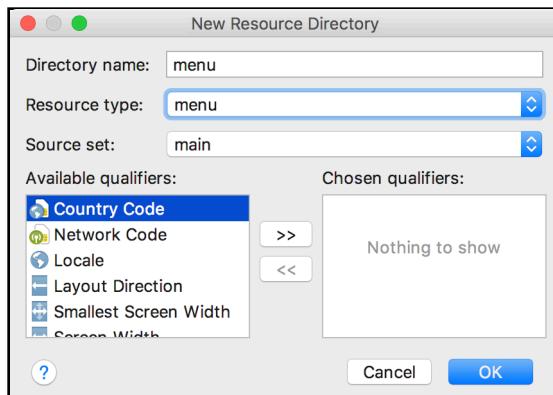


Saving changes

The only major feature left is to save the user's edits. For that, you'll add a checkmark Toolbar item to trigger the save.

First, you 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 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:

```
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 need some new methods in `BookmarkRepo`. Open **BookmarkRepo.kt** and add the following methods:

```
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 need to update the bookmark view model class. For that, you need a method to convert a bookmark view model to the database bookmark model.

Open **BookmarkDetailsViewModel.kt** and add the following method:

```
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`. You load the original bookmark values 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:

```
fun updateBookmark(bookmarkView: BookmarkDetailsView) {  
    // 1  
    GlobalScope.launch {  
        // 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 updates 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:

```
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 `editTextName` 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.

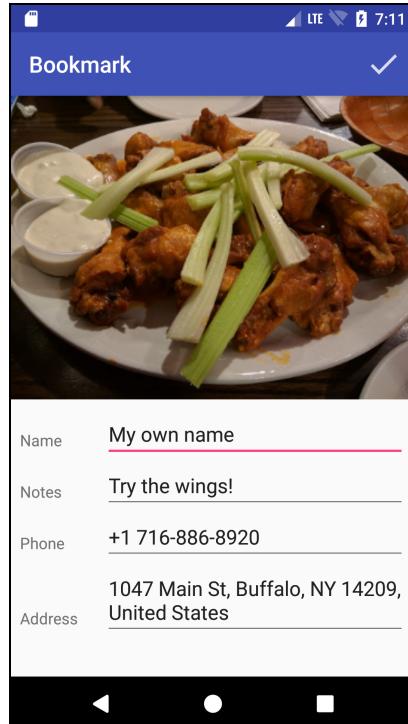
Next, you need to add code to respond to the user tapping the checkmark menu item and then call `saveChanges()`.

Add the following method:

```
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 of 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.



Where to go from here?

Congratulations! You can now edit bookmarks, but there's still more work to do. The next chapter wraps 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 also replace the photo for a bookmark.

Getting started

The starter project for this chapter includes an additional icon that you need to complete the chapter. You can either begin this chapter with the starter project or copy `src/main/res/drawable/ic_other.png` from the starter project into yours.

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` and in the method `setupPlacesClient()` in `MapsActivity.kt`. Read Chapter 13 for more details about the Google Maps key.

Bookmark navigation

At the moment, 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 they can use to jump directly to any bookmark.

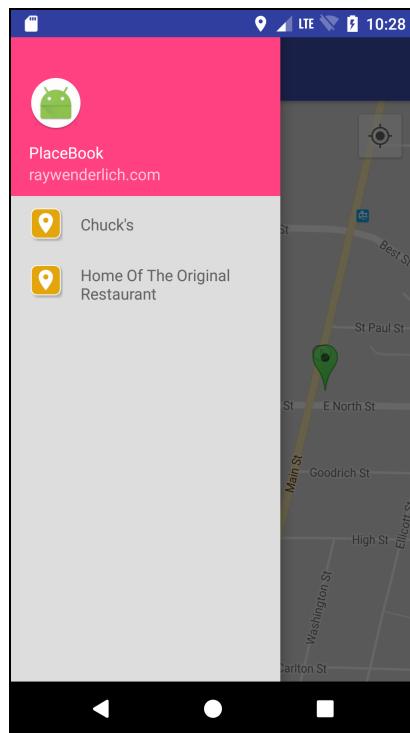
Navigation drawer design

It's difficult to use Android without encountering a navigation drawer. Although its 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 the top of the main content and slides back in once an action has been taken by the user.

You can add a navigation drawer in three steps:

- Make `DrawerLayout` the root view of the Layout.
- Make the first view within `DrawerLayout` the main content.
- Make the second view within `DrawerLayout` the navigation drawer content.

The final navigation drawer will look like this:



Navigation drawer layout

To create the drawer Layout, you need to 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 need to move the map fragment to a separate Layout.

Create a new Layout resource file in **res/layout**, and name it **main_view_maps.xml**. Then, replace the 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: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 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 **res/layout**, and name it **drawer_view_maps.xml**. Then, replace the contents with the following:

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawerView"
    android:layout_width="240dp"
    android:layout_height="match_parent"
    android:layout_gravity="start"
    android:orientation="vertical"
    android:background="#ddd">

    <LinearLayout
        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 important key elements:

- 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 places 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 is 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 `res/layout`, and name it `bookmark_item.xml`. Then, replace the contents with the following:

```
<?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 you need for the navigation drawer. Next, you need to update the main maps Activity to use the new Layouts. Open **activity_maps.xml** and replace the contents with the following:

```
<?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"
    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>
```

The 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:

```
<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**:

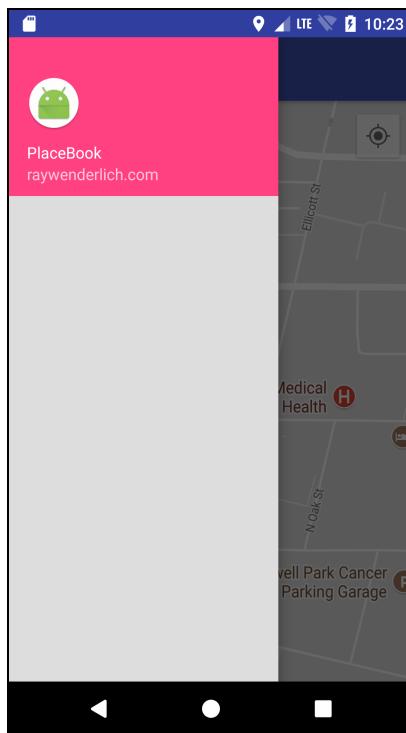
```
private fun setupToolbar() {  
    setSupportActionBar(toolbar)  
}
```

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 slides 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` takes your `drawerLayout` and `toolbar` and fully manages the display and functionality of the toggle icon. You call `toggle.syncState` 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`.

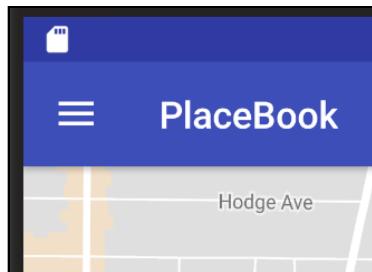
All that's left to do is call `setupToolbar()` when the Activity is created.

Add the following lines before `setupGoogleClient()` in `onCreate()`:

```
setupToolbar()
```

This calls the two new methods to bind the controls and set up the toolbar with the toggle icon.

Build and run the app. Tap the toggle (hamburger) icon to test the navigation drawer slide.



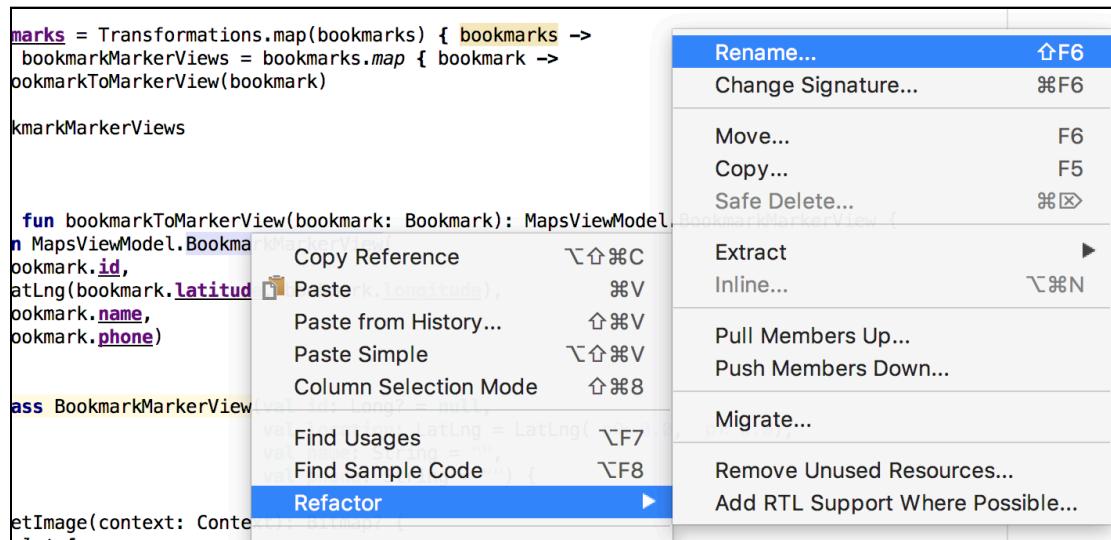
Populating the navigation bar

To populate the navigation bar, you need to provide an Adapter to the RecyclerView and use LiveData to update the Adapter any time bookmarks change in the database.

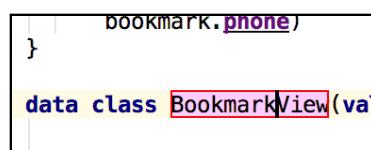
The Adapter requires some view data — one option is to create a new data class in MapsViewModel. You already have the BookmarkMarkerView class used by the MapsActivity for the map markers, so you can take advantage of the existing class and the code that observes changes to the data.

Since you'll be using BookmarkMarkerView to display markers and the navigation drawer items, it needs a more generic name. This is a great opportunity to use Android Studio's available refactoring capabilities.

Open **MapsViewModel.kt** and locate 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 is highlighted. Change the name to **BookmarkView** and press Enter.



This automatically updates all references to use `BookmarkView` instead of `BookmarkMarkerView`. This is a great feature that can save a lot of time when renaming classes, methods or variables.

Note: When performing a refactor, there might be an additional step in some cases. If you notice the ‘Refactoring Preview’ window appears on the bottom left of Android Studio, you will need to review the changes, and if they match your intent, click ‘Do Refactor’ to perform the operation.

Use the same rename refactor to change the following:

- `getBookmarkMarkerViews()` \rightarrow `getBookmarkViews()`.
- `mapBookmarksToMarkerView()` \rightarrow `mapBookmarksToBookmarkView()`.
- `bookmarkToMarkerView()` \rightarrow `bookmarkToBookmarkView()`.

In **MapsActivity.kt**, use the rename feature to change `createBookmarkMarkerObserver()` to `createBookmarkObserver()`.

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:

```
// 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 that you learned about in Chapter 7, “Recyclers”.

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. `onCreateViewHolder` 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.

You can now 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()
```

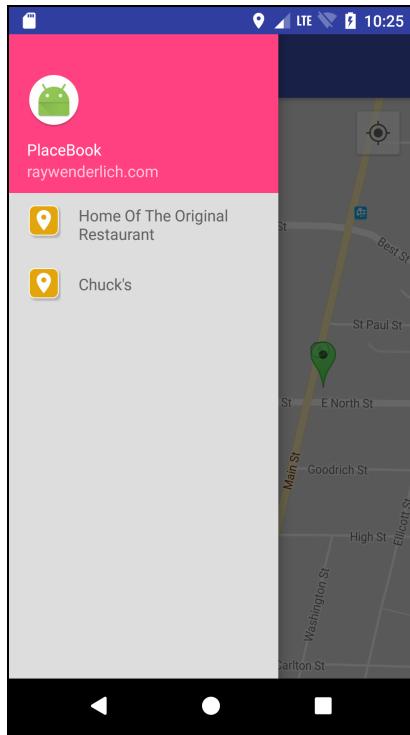
Also, you need to make sure the list Adapter is updated any time the list of bookmarks changes. This can be handled in **createBookmarkObserver()**.

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 causes 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. It's time to add the ability to zoom to a bookmark when the user taps an item in the navigation drawer.

First, you need to 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 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 adds 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` are 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)
}
```

Here's how it works:

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 is 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)  
    }  
}
```

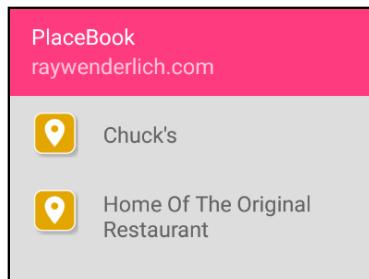
This method is called when a `ViewHolder` is initialized. It sets an `onClickListener` on the `ViewHolder`. When the click event is fired, you get the `bookmarkView` associated with the `ViewHolder` and call `moveToBookmark()` to zoom the map to the bookmark.

Before wrapping up this feature, you need to 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 closes, and the map zooms 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 inside **ui**, and name it **PhotoOptionDialogFragment.kt**. Then, set the contents as follows:

```
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
        val context = activity as Context
        // 8
        if (canCapture(context)) {
            options.add("Camera")
            captureSelectIdx = 0
        }
        // 9
        if (canPick(context)) {
            options.add("Gallery")
            pickSelectIdx = if (captureSelectIdx == 0) 1 else 0
        }
        // 10
        return AlertDialog.Builder(context)
            .setTitle("Photo Option")
            .setItems(options.toTypedArray<CharSequence>()) {
                // 12
                which ->
                if (which == captureSelectIdx) {
                    // 13
                    listener.onCaptureClick()
                } else if (which == pickSelectIdx) {

```

```
        // 12
        listener.onPickClick()
    }
    .setNegativeButton("Cancel", null)
    .create()
}

companion object {
    // 13
    fun canPick(context: Context) : Boolean {
        val pickIntent = Intent(Intent.ACTION_PICK,
            MediaStore.Images.Media.EXTERNAL_CONTENT_URI)
        return (pickIntent.resolveActivity(
            context.packageManager) != null)
    }
    // 14
    fun canCapture(context: Context?) : Boolean {
        val captureIntent = Intent(
            MediaStore.ACTION_IMAGE_CAPTURE)
        return (captureIntent.resolveActivity(
            context.packageManager) != null)
    }
    // 15
    fun newInstance(context: Context?):
        PhotoOptionDialogFragment? {
        // 16
        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 shows 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 is included. If the device has a camera to capture new images, then a `Camera` option is 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 are defined dynamically, because the position of the Gallery and Camera options may change based on the device capabilities.
6. An options `ArrayList` is defined to hold the `AlertDialog` options.
7. The next few calls require a `Context` object. You'll use the `activity` property of the `AlertDialog()` class as the context. Since the `activity` property has a getter method and may change between calls, you set a temporary un-mutable local variable and use it to prevent compiler errors.
8. 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.
9. 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.
10. The `AlertDialog` is built using the options list, and an `onClickListener` is provided to respond to the user selection.
11. If the Camera option was selected, then `onCaptureClick()` is called on `listener`.
12. If the Gallery option was selected, then `onPickClick()` is called on `listener`.
13. `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.
14. `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.
15. `newInstance` is a helper method intended to be used by the parent activity when creating a new `PhotoOptionDialogFragment`.
16. 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 causes 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.

While in **BookmarkDetailsActivity.kt**, add the following method:

```
private fun replaceImage() {
    val newFragment = PhotoOptionDialogFragment.newInstance(this)
    newFragment?.show(supportFragmentManager, "photoOptionDialog")
}
```

When the user taps on the bookmark image, you call `replaceImage()`. You attempt to create the `PhotoOptionDialogFragment` fragment. If `newFragment` is not null, then it's displayed.

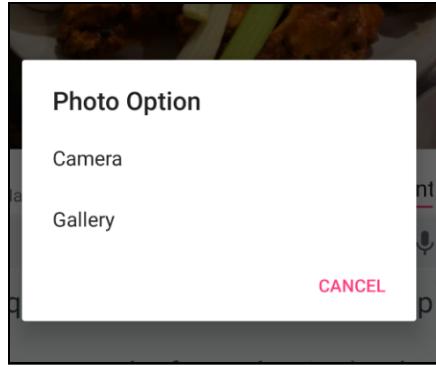
All that's left is to listen for the `imageViewPlace` to be tapped and call `replaceImage()`.

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 is located at the filename Uri you provided.

Generate a unique filename

First, you need to create a helper method to generate a unique image filename.

Open `ImageUtils.kt` and add the following method:

```
@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 `java.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 need to add a property to the details Activity to keep track of the image File.

Open **BookmarkDetailsActivity.kt** and add the following property:

```
private var photoFile: File? = null
```

This is 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 can 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 captures an image. Replace the contents of `onCaptureClick()` with the following:

```
// 1
photoFile = null
try {
    // 2
    photoFile = ImageUtils.createUniqueImageFile(this)
} catch (ex: java.io.IOException) {
    // 3
    return
}
// 4
photoFile?.let { photoFile ->
    // 5
    val photoUri = FileProvider.getUriForFile(this,
        "com.raywenderlich.placebook.fileprovider",
        photoFile)
    // 6
```

```
val captureIntent =  
    Intent(android.provider.MediaStore.ACTION_IMAGE_CAPTURE)  
    // 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)  
}
```

Here's the code breakdown:

1. Any previously assigned `photoFile` is cleared.
2. You call `createUniqueImageFile()` to create a uniquely named image File and assign it to `photoFile`.
3. If an exception is thrown, the method returns without doing anything.
4. You use the `? . let` to make sure `photoFile` is not `null` before continuing with the rest of the method.
5. `FileProvider.getUriForFile()` is called to get a Uri for the temporary photo file.
6. 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.
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:

```
<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.getUriFromFile()`.

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 creates a new `res` directory named `xml` containing the new `file_paths.xml` file.

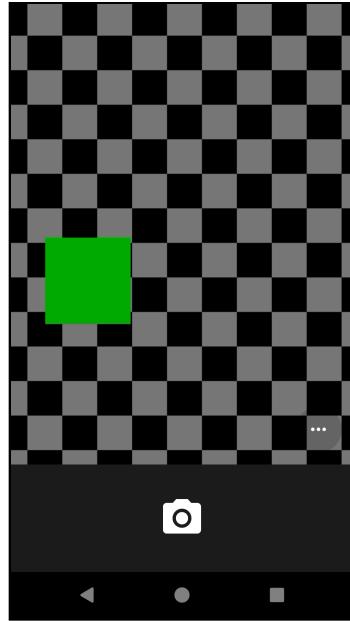
Now, you can fill in the `file_paths.xml` with the allowed file paths.

Replace the contents of `file_paths.xml` with the following:

```
<?xml 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 can snap a photo.



Emulator Camera View

The photo won't 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:

```
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:

```
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 is 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 to load the full image this time.
4. `BitmapFactory.decodeFile()` loads the downsampled image from the file returns it.

The `BookmarkView` class now needs a new method to replace the image for a bookmark.

Add the following method to `BookmarkDetailsView` in `BookmarkDetailsViewModel.kt`:

```
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` can replace its own image, you need to 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()`.

To read in and process the image captured by the system, you need a method that takes a file path and returns 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:

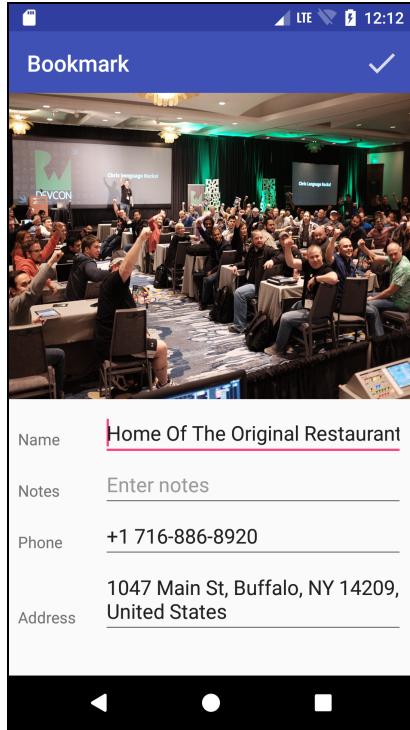
```
override fun onActivityResult(requestCode: Int, resultCode: Int,
    data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    // 1
    if (resultCode == android.app.Activity.RESULT_OK) {
        // 2
        when (requestCode) {
            // 3
            REQUEST_CAPTURE_IMAGE -> {
                // 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 which call 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 updates 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:

```
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.

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)
```

To process the results of the image selection, you 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.

Next, you 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 updates 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 PlaceBook. Even though PlaceBook is perfectly functional as-is, it's often the subtle enhancements that make an app go from good to great. With that in mind, you'll wrap things up by making the following changes:

- Adding categories for bookmarks.
- Displaying category specific icons on the map.
- Adding place search.
- Adding ad-hoc bookmark creation.
- Adding bookmark deletions.
- Adding bookmark sharing.
- Updating the color scheme.
- Displaying progress using indicators.

Getting started

The starter project for this chapter includes additional resources and an updated app icon. You can either begin this chapter with the starter project or copy the following resources from the starter project into your project:

- 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** and in the method `setupPlacesClient()` in **MapsActivity.kt**.

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, so 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 **Bookmark**.

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, Room will throw an exception. When you change the version number, Room creates a new database on the first run using the new version number.

Converting place types

If you examine `Place` defined by the Google Play Services, you'll notice that it provides a fairly long list of place types:

```
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 will get assigned to the **Other** category.

To get started, you need a method that maps a Google Place type to a supported PlaceBook category. However, 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 `Place` types to category names. Any type 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()
```

You initialize `categoryMap` 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 takes in a `Place` type and converts it to a valid category. `category` is initialized to "`Other`" by default. If `categoryMap` contains a key matching `placeType`, 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 a string value. In this case, you use `containsKey()` to ensure that the key is in the `HashMap`, so you're safe.

It's time to make use of the new icons provided in the starter project. The icons correspond to the categories, like so:

- ic_other = Other
- ic_gas = Gas
- ic_lodging = Lodging
- ic_restaurant = Restaurant
- ic_shopping = Shopping

First, you need to map the category names to the drawable resource files.

Add the following method to `BookmarkRepo`:

```
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`:

```
private var allCategories: HashMap<String, Int> =
    buildCategories()
```

You initialize `allCategories` to hold the mapping of category names to resource IDs.

Add the following method:

```
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

You're ready to update the map's view model to support bookmark categories.

Open **MapsViewModel.kt** and add the following private method:

```
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 getting assigned to a single place.

1. The category defaults to "Other" in case there's no type assigned to the place.
2. The method first checks the `placeTypes` List to see if it's populated.
3. If so, you extract the first type from the List and call `placeTypeToCategory()` to make the conversion.
4. Finally, you return the category.

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) {
```

This adds `categoryResourceId` 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

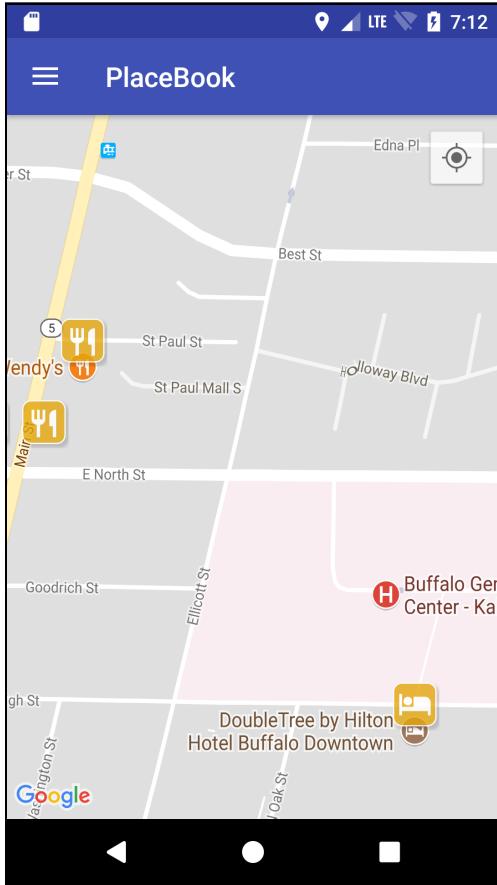
You can now update the user interface to show the category icons.

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 you load from the `categoryResourceId` property on the bookmark.

Build and run the app, and add bookmarks for a variety of place types. Notice the different icons that are displayed on the map.



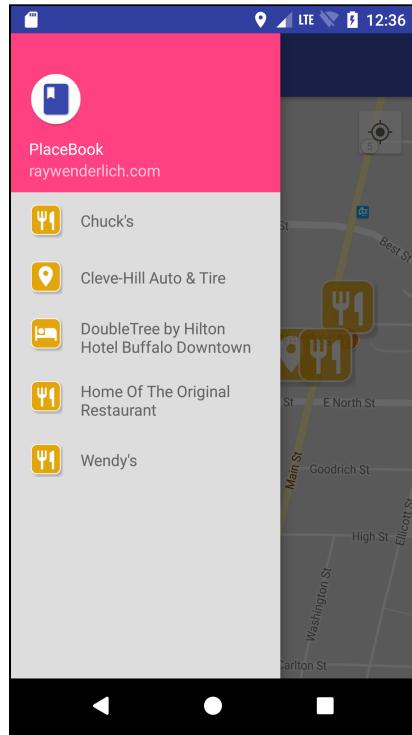
Next, you need to update the navigation drawer to display the new category icons.

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` is set; 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 to add before moving on: You need 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:

```
<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 below the bookmark name that displays the currently selected category icon using an ImageView. It also 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:

```
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:

```
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.

To fill the spinner with options, you 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 `categories` that takes all of the `HashMap` keys, which are the category names, and returns 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))
}
```

Here's how it works:

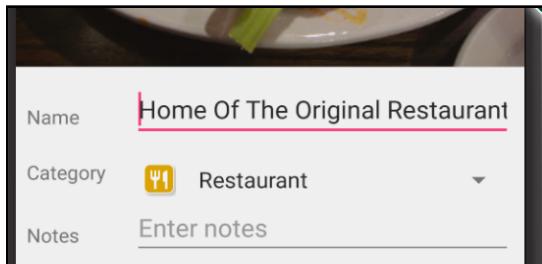
1. The method returns immediately if `bookmarkDetailsView` is null.
2. You retrieve the category icon `resourceId` from the view model.
3. If the `resourceId` is not null, you update `imageViewCategory` to the category icon.

4. You retrieve the list of categories 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(), you assign the Adapter to a standard built-in Layout resource.
6. You then assign the Adapter to the spinnerCategory control.
7. You update spinnerCategory 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 two issues: The category icon does not update when the value is changed, and the category change is not saved. Time to fix that!

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 sets 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. You assign the `spinnerCategory.onItemSelectedListener` property to an instance of the `onItemSelectedListener` class that implements `onItemSelected()` and `onNothingSelected()`.
3. When the user selects a new category, you call `onItemSelected()`. You determine the new category by the current spinner selection position, and update `imageViewCategory` 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 updates as you change selections, and the new category persists 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 displays a dynamic list of choices.

Note: If you want to customize the user experience entirely, 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 this case, a search button is provided, and the autocomplete widget shows as an Activity.

First, you need a method to kick off the search feature.

Use PlaceAutocomplete search

Open **MapsActivity.kt** and add the following property to the companion object:

```
private const val AUTOCOMPLETE_REQUEST_CODE = 2
```

Add the following method:

```
private fun searchAtCurrentLocation() {  
    // 1  
    val placeFields = listOf(  
        Place.Field.ID,  
        Place.Field.NAME,  
        Place.Field.PHONE_NUMBER,  
        Place.Field.PHOTO_METADATA,  
        Place.Field.LAT_LNG,  
        Place.Field.ADDRESS,  
        Place.Field.TYPES)  
  
    // 2  
    val bounds =  
        RectangularBounds.newInstance(map.projection.visibleRegion.latLngBounds)  
    try {  
        // 3  
        val intent = Autocomplete.IntentBuilder(  
            AutocompleteActivityMode.OVERLAY, placeFields)  
            .setLocationBias(bounds)  
            .build(this)  
        // 4  
        startActivityForResult(intent, AUTOCOMPLETE_REQUEST_CODE)  
    } catch (e: GooglePlayServicesRepairableException) {  
        //TODO: Handle exception  
    } catch (e: GooglePlayServicesNotAvailableException) {  
        //TODO: Handle exception  
    }  
}
```

Here's the code breakdown:

1. You define the fields, which informs the Autocomplete widget what attributes to return for each place.
2. You compute the bounds of the currently visible region of the map.
3. Autocomplete provides an IntentBuilder method to build up the Intent. You pass AutocompleteActivityMode.OVERLAY to indicate that the search widget can overlay the current Activity. The other option is AutocompleteActivityMode.FULLSCREEN, which causes the search interface to replace the entire screen.

You pass the map bounds to setBoundBias(). This tells the search widget to look for places within the current map window before searching other areas.

4. You start the Activity and pass a request code of AUTOCOMPLETE_REQUEST_CODE. When the user finishes the search, the results are identified by this request code.

You surrounded the code with 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 = Autocomplete.getPlaceFromIntent(data)
                // 4
                val location = Location("")
                location.latitude = place.latLng?.latitude ?: 0.0
                location.longitude = place.latLng?.longitude ?: 0.0
                updateMapToLocation(location)
                // 5
                displayPoiGetPhotoStep(place)
            }
    }
}
```

onActivityResult() is called by Android when the user completes the search.

1. First, you check the requestCode 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,

Autocomplete provides a handy method, `getPlaceFromIntent()`, that takes the data and returns a populated Place object.

4. You convert the place `latLng` to a location and pass that 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 `displayPoiGetPhotoMetaDataSet()` and pass it the found place. This loads the place photo and displays 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>
```

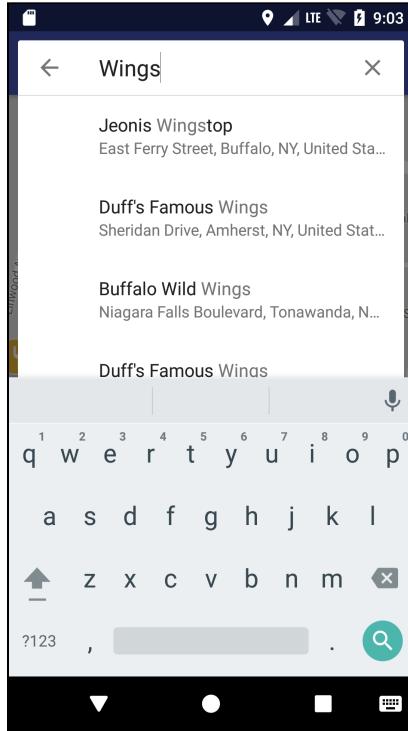
This tells the Layout engine to place the search button in the bottom-right corner of the map with a margin of 16dp on each side.

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 the end of `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 can make this possible by allowing the user to drop a pin at any location on the map.

Currently, `MapsViewModel` includes a method to create a bookmark from a place, but now you need one to create a bookmark from only a map location.

Open `MapsViewModel.kt` and add the following method:

```
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.

Next, you need a method in **MapsActivity.kt** to take advantage of this new method:

Open **MapsActivity.kt** and add the following method:

```
private fun newBookmark(latLng: LatLng) {
    GlobalScope.launch {
        val bookmarkId = mapsViewModel.addBookmark(latLng)
        bookmarkId?.let {
            startBookmarkDetails(it)
        }
    }
}
```

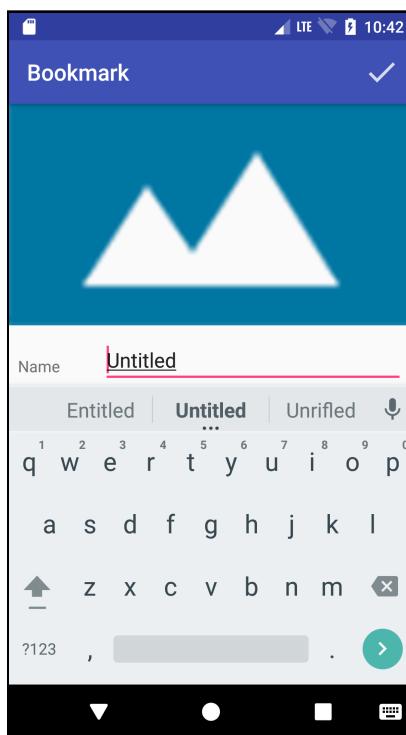
This method creates a new bookmark from a location, and then it starts the bookmark details Activity to allow editing of the new bookmark. The call to `addBookmark` runs within a coroutine block because it accesses the database and can't run on the main thread.

You now need to listen for the user to long tap on the map.

Add the following to the end of `setupMapListeners()`:

```
map.setOnMapLongClickListener { latLng ->
    newBookmark(latLng)
}
```

Build and run the app. Long tap anywhere on the map and the bookmark Activity screen pops up with a new untitled bookmark using a default photo.



Name the bookmark, assign it a category and save the changes; the new bookmark appears 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's no longer needed or one that was added by accident. For this, 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>`:

```
<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.

Next, you'll work your way up from the bottom-level code to the top, adding in basic support for deleting bookmarks.

In `utils`, create a new Kotlin file named `FileUtils.kt` 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. You'll use this 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) {
    GlobalScope.launch {
        val bookmark = bookmarkDetailsView.id?.let {
            bookmarkRepo.getBookmark(it)
        }
        bookmark?.let {
            bookmarkRepo.deleteBookmark(it)
        }
    }
}
```

This method takes in a **BookmarkDetailsView** 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, it deletes the bookmark and the Activity closes using **finish()**.

All of the support code is in place; now you need to respond to the delete menu action.

In `onOptionsItemSelected()`, add the following additional case to the `when` statement before the final `else`:

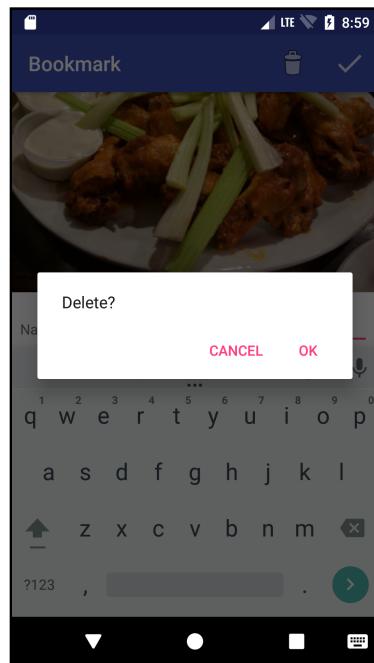
```
R.id.action_delete -> {
    deleteBookmark()
    return true
}
```

This calls `deleteBookmark()` when the delete icon is tapped. Since you're 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) { repoBookmark ->
        repoBookmark?.let {
            bookmarkToBookmarkView(bookmark)
        }
    }
}
```

This ensures that a `null` bookmark is not passed to `bookmarkToView` by using the `repoBookmark?.let` statement.

Build and run the app. Edit an existing bookmark and use the delete icon to delete it. The bookmark is deleted, and you 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 figures out the apps that support your data type and presents the user with a list of choices.

Your next step is to 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)
}
```

Here's what's happening:

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. Read the documentation at <https://developers.google.com/maps/documentation/urls/guide> for details about constructing map URLs.

There are two different styles of URLs to use depending on whether a place ID is available. If the user creates an ad-hoc bookmark, then the directions go directly to the latitude/longitude of the bookmark. If the bookmark is created from a place, then the directions 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 looks 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 contains 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 looks like this: `https://www.google.com/maps/dir/?api=1&destination=Riverstone+Plaza&destination_place_id=ChIJAAAAAAAAAAAR1tSJBrRUoKI`
5. You create the sharing Activity Intent and set the action to `ACTION_SEND`. This tells Android that this Intent is meant to share its data with another application installed on the device.
6. Multiple types of extra data can be added to the Intent. The app 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 app 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 need to 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, you’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>
```

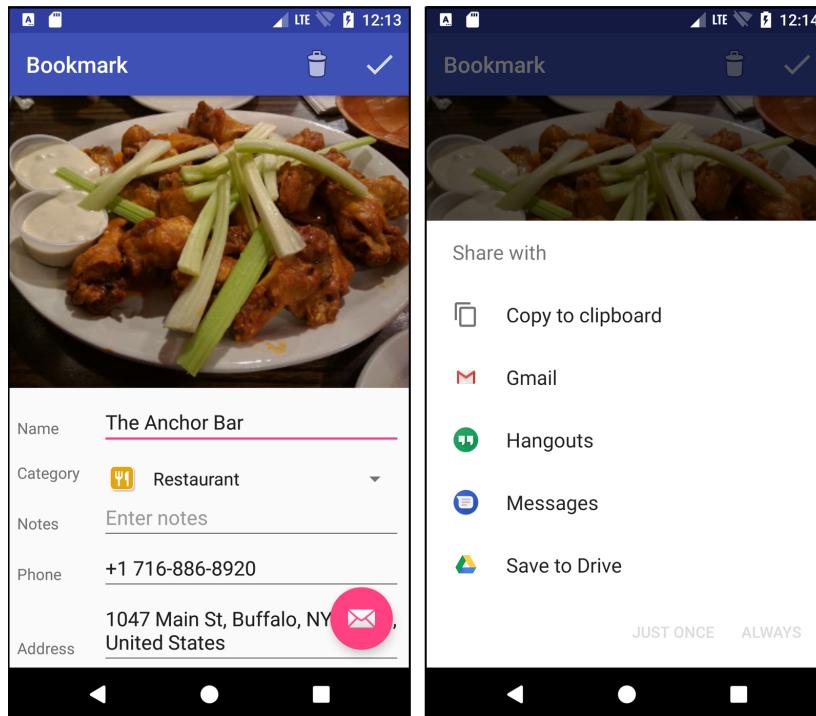
Open **BookmarkDetailsActivity.kt** and add the following method:

```
private fun setupFab() {  
    fab.setOnClickListener { sharePlace() }  
}
```

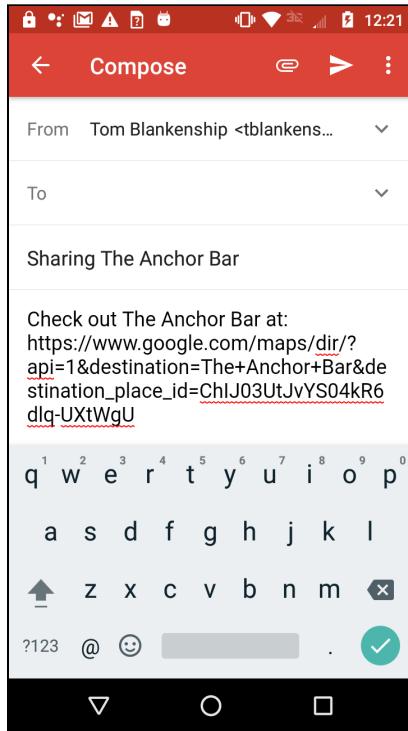
Add the following to the end of `onCreate()`:

```
setupFab()
```

Build and run the app. Open a bookmark and tap the sharing button. You'll see a share dialog similar to the following. Your choices may vary depending on the apps installed on your device.



Tap on Gmail, and it launches the Gmail app and populates the subject and message body.



Color scheme

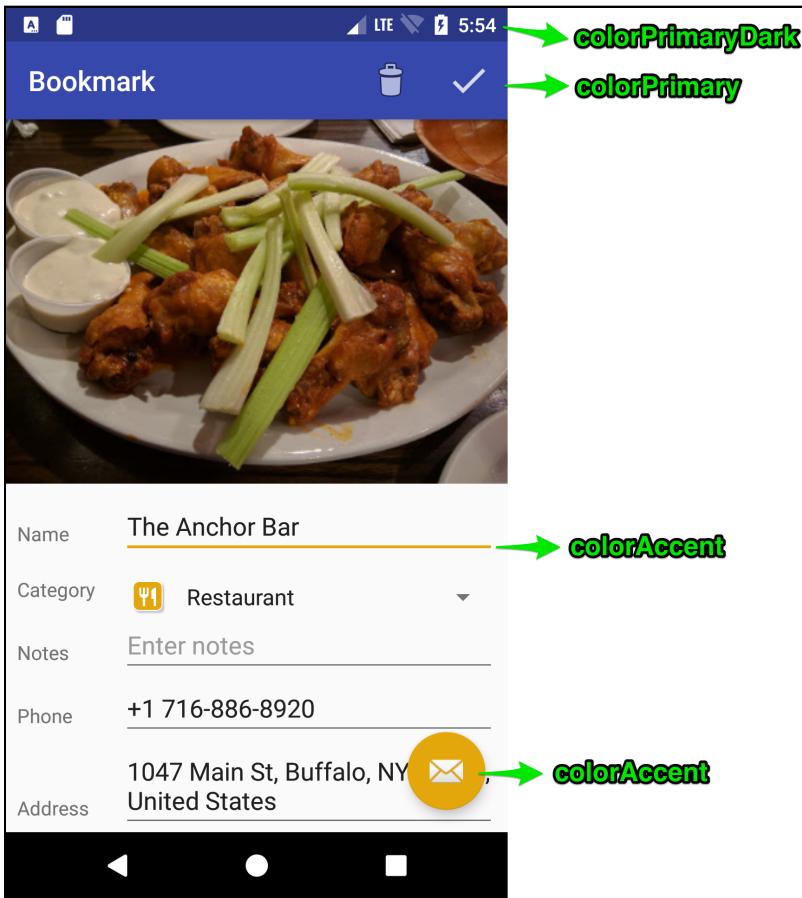
It's a minor change, but updating the color scheme to match the bookmark icon colors will make the app look much better.

Open **values/colors.xml** and update the three colors:

```
<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 is used by the main action bar. The primary dark 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's used by the floating buttons and the highlight color when a field is in focus.

Build and run the app. The overall app colors look a lot 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>`:

```
<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 a circular progress indicator.

Open **MapsActivity.kt** and add the following new methods:

```
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 main window to prevent user touches.

`enableUserInteraction()` clears the flag set by `disableUserInteraction()`.

Add the following new methods:

```
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 need to show and hide the progress bar in a few strategic locations.

You need to show progress when a place or place photo is loading. 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 displays the progress bar when a place is tapped.

Add a call to `showProgress()` in `onActivityResult()`, after the call to `updateMapToLocation()`:

```
showProgress()
```

This displays 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 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 hides the progress bar if the place cannot be retrieved and the `displayPoi` steps end here.

In `displayPoiGetPhotoStep()`, add a call to `hideProgress()` as the last line in the `addOnFailureListener` code block:

```
hideProgress()
```

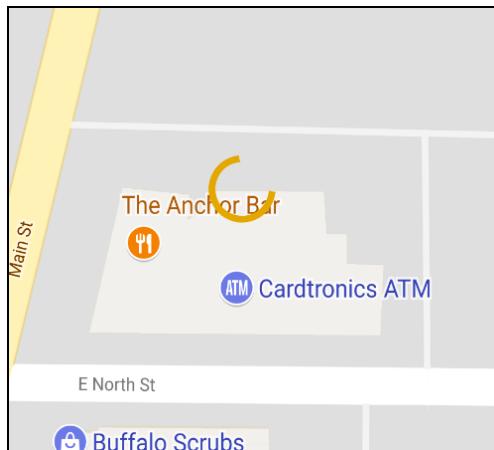
This hides the progress bar if there's an error fetching the photo and the `displayPoi` steps end here.

In `displayPoiDisplayStep()` add a call to `hideProgress()` as the first line:

```
hideProgress()
```

This hides the progress bar before the new marker is shown.

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 made it through the entire PlaceBook app section. You built a useful map-based app 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 & 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**](#)



20

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 are covered:

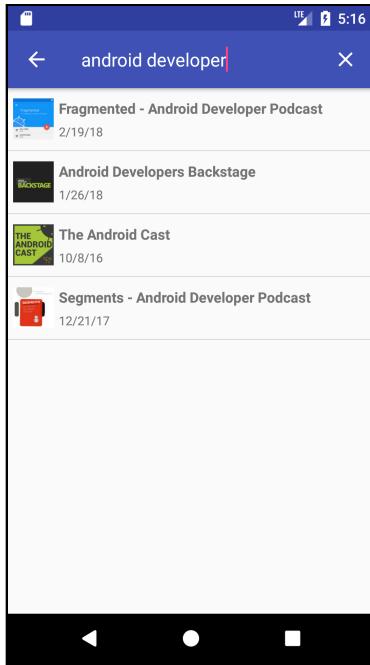
- Android networking.
- Retrofit REST API library.
- XML Parsing.
- Search activity.
- MediaPlayer library.



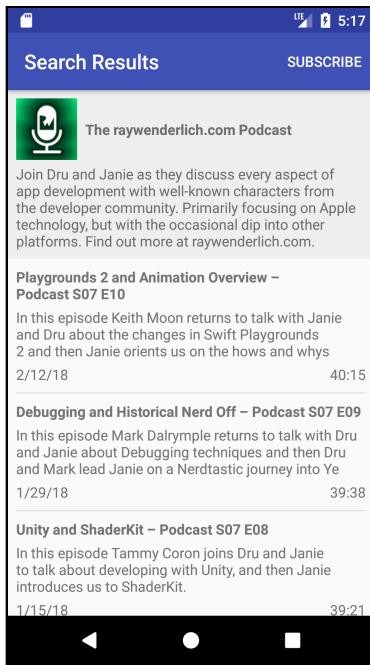
Getting started

PodPlay will contain these main features:

1. Quick searching of podcasts by keyword or name.



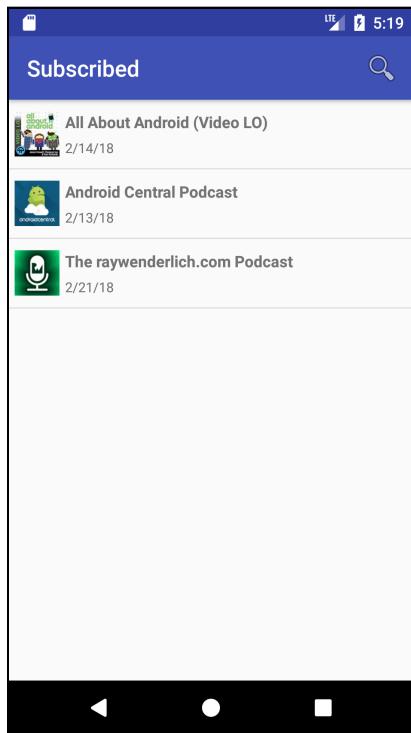
2. Display for previewing podcast episodes.



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 uses the same structure as PlaceBook, but it will also add a new services layer.

Open Android Studio and close any open projects so that the “Welcome to Android Studio” dialog is displayed.

Select **Start a new Android Studio project**.

Click the **Empty Activity** project type in the **Phone and Tablet** tab, and click **Next**.

Fill out the **Configure your project** dialog:

- Name: PodPlay
- Package name: com.raywenderlich.podplay
- Save Location: *Select your own location*
- Language: Kotlin
- Minimum API level: API 19: Android 4.4 (KitKat)
- Leave everything else unchecked.

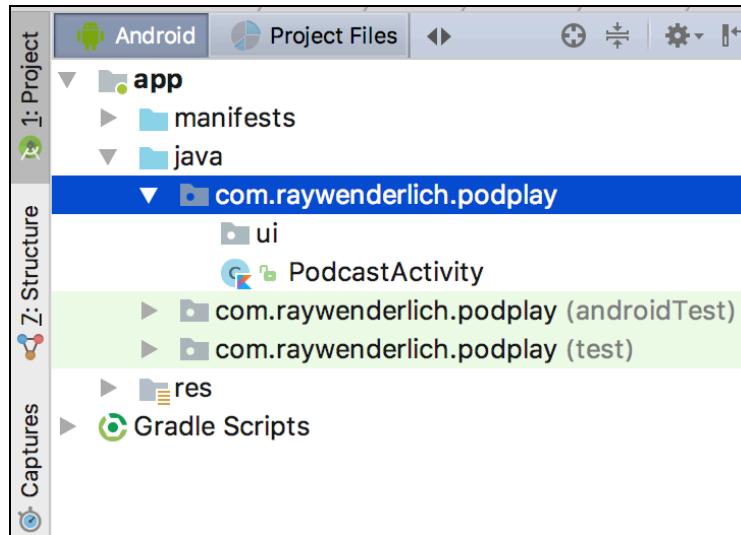
Click **Finish**.

The new project is created and the **MainActivity.kt** file is shown.

Select **MainActivity.kt** in the Project navigator and press **Shift-F6** to rename the activity. In the rename dialog, change the name to **PodcastActivity**, leave the other options to their default values and click **Refactor**.

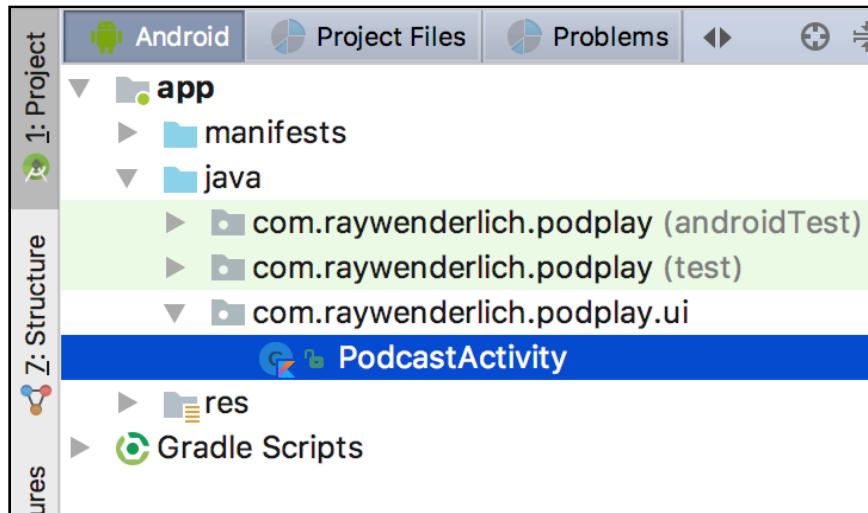
Select **activity_main.xml** in the Project navigator inside **res/layout** and press **Shift-F6** to rename the Layout. In the rename dialog, change the name to **activity_podcast.xml**, leave the other options to their default values and click **Refactor**.

Next, inside **com.raywenderlich.podcast**, create a new package named **ui**.

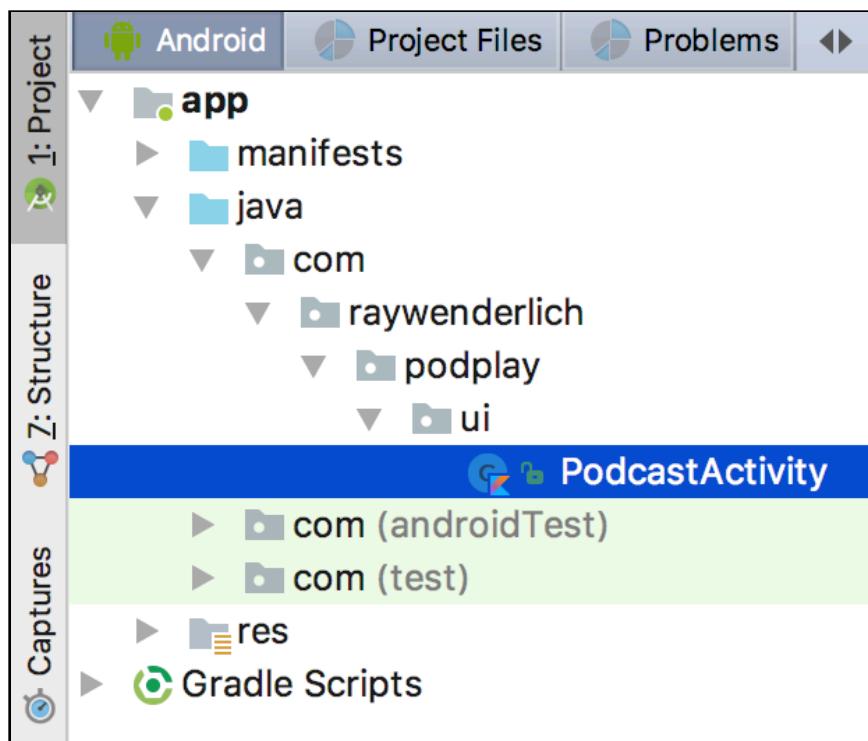


In the Project navigator, move **PodcastActivity.kt** into **ui** using drag-and-drop. In the Move refactor dialog, leave all of the options set to their defaults and click **Refactor**.

If you have the option to **Compact middle packages** turned on in the Android Project view, then you'll see something like this:



If you have the option to **Compact middle packages** turned off, you'll see this:



Where are the podcasts?

Before you get to the fun part of podcast playback, you 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 are found on the main website that promotes or produces the podcast. There's normally a feed button that provides 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; this includes the title of the podcast, the date it was published, associated artwork, a descriptive summary of the podcast and 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 the 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 can use to allow users to search for podcast by keywords, making it easy to subscribe to a podcast.

Android networking

So far, all of the apps you've built during your apprenticeship have been self-contained. They have not had to access any remote or network-based services directly. Although PlaceBook did access Google Places and download place photos, that was all handled by the Places library. That's about to change with PodPlay.

PodPlay requires 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 you concentrate on building app functionality.

There are a few choices available:

- **Volley**: Google provides a 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 returns a list of these feeds with a JSON structure.

PodPlay architecture

Continuing with the layered architecture, you'll create a service layer that handles all network access to iTunes and hides 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 almost any podcast from a variety of categories.

Apple also provides an API to allow searching the podcast directory. You can find the full API documentation 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'll use here allows searching for podcasts by titles or keywords. It looks like this:

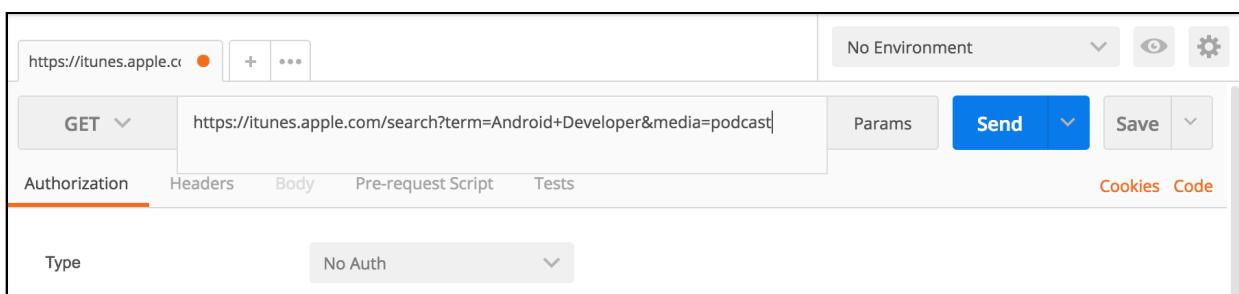
`https://itunes.apple.com/search?term=Android+Developer&media=podcast`

The `media=podcast` part tells iTunes to only search for podcasts.

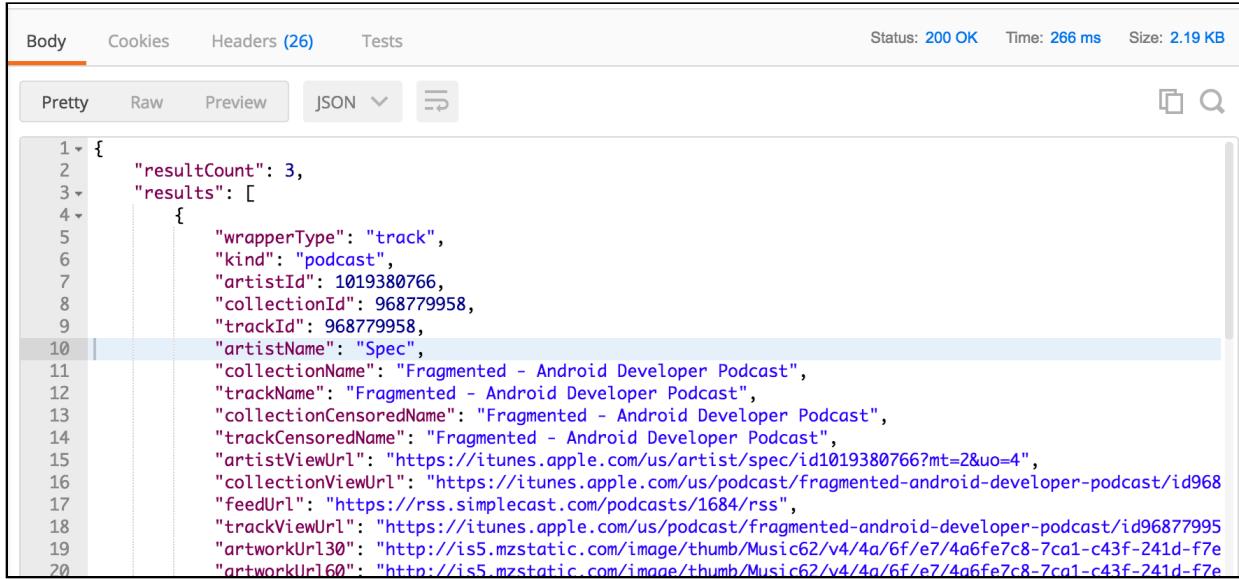
`term=Android+Developer` is the search term. The plus sign is used because the search term must be URL-encoded. URL-encoding replaces all spaces with plus symbols and encodes all other special characters except letters, numbers, periods (.), dashes (-), underscores (_) and asterisks (*).

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:



The screenshot shows a network response viewer with the following details:

- Body tab selected.
- Cookies, Headers (26), Tests tabs visible.
- Status: 200 OK, Time: 266 ms, Size: 2.19 KB.
- JSON dropdown menu open.
- Pretty, Raw, Preview buttons.
- Search icon.
- JSON content:

```
1 {  
2   "resultCount": 3,  
3   "results": [  
4     {  
5       "wrapperType": "track",  
6       "kind": "podcast",  
7       "artistId": 1019380766,  
8       "collectionId": 968779958,  
9       "trackId": 968779958,  
10      "artistName": "Spec",  
11      "collectionName": "Fragmented - Android Developer Podcast",  
12      "trackName": "Fragmented - Android Developer Podcast",  
13      "collectionCensoredName": "Fragmented - Android Developer Podcast",  
14      "trackCensoredName": "Fragmented - Android Developer Podcast",  
15      "artistViewUrl": "https://itunes.apple.com/us/artist/spec/id1019380766?mt=2&uo=4",  
16      "collectionViewUrl": "https://itunes.apple.com/us/podcast/fragmented-android-developer-podcast/id968",  
17      "feedUrl": "https://rss.simplecast.com/podcasts/1684/rss",  
18      "trackViewUrl": "https://itunes.apple.com/us/podcast/fragmented-android-developer-podcast/id96877995",  
19      "artworkUrl30": "http://is5.mzstatic.com/image/thumb/Music62/v4/4a/6f/e7/4a6fe7c8-7ca1-c43f-241d-f7e",  
20      "artworkUrl160": "http://is5.mzstatic.com/image/thumb/Music62/v4/4a/6f/e7/4a6fe7c8-7ca1-c43f-241d-f7e"  
]
```

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.

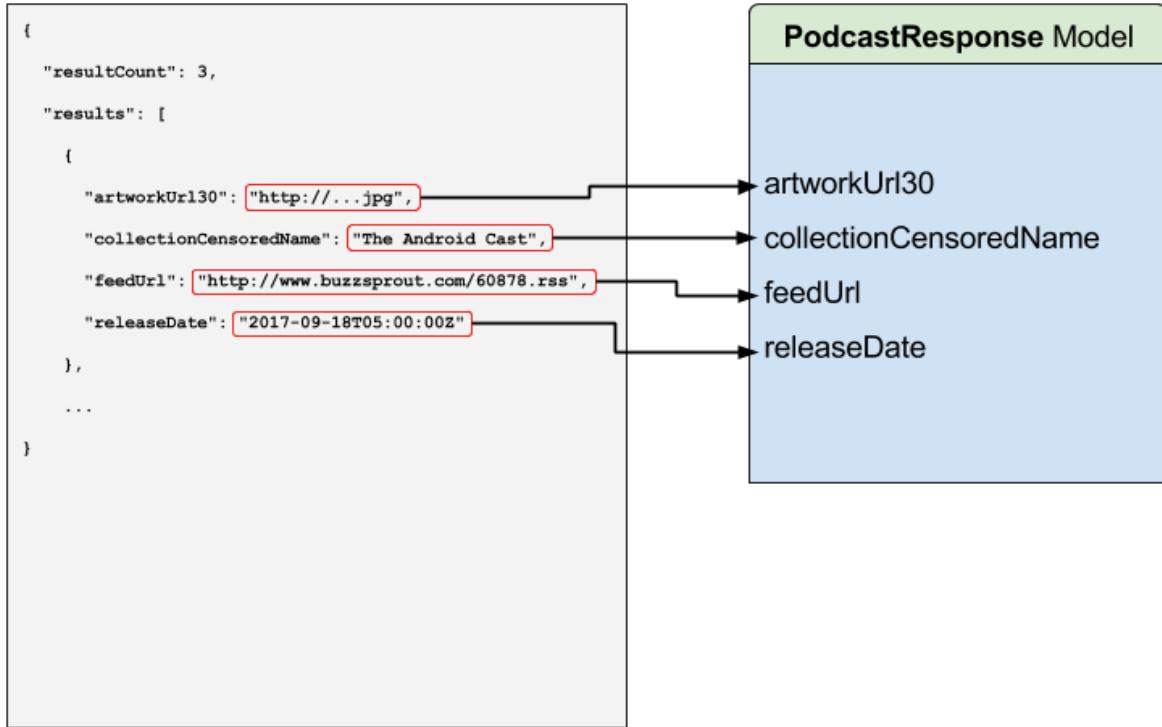
Retrofit

Now that you know how to get search results, the next step is to turn them into data models.

If you manually perform 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's 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 more comfortable! 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**. Retrofit uses the annotation data 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 = '1.3.21'  
    retrofit_version = '2.5.0'  
}
```

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.

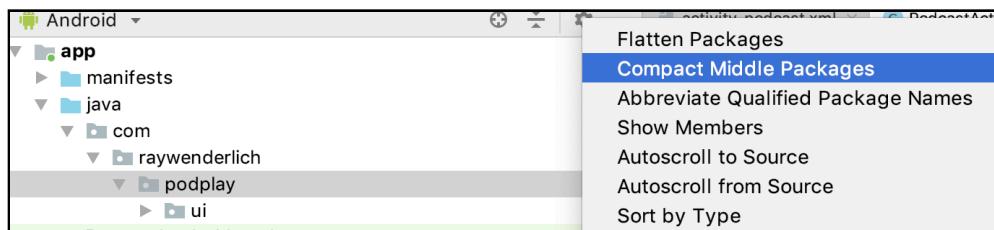
A warning about changing Gradle files is shown at the top of the editor. Click on **Sync Now**.

Creating the podcast response model

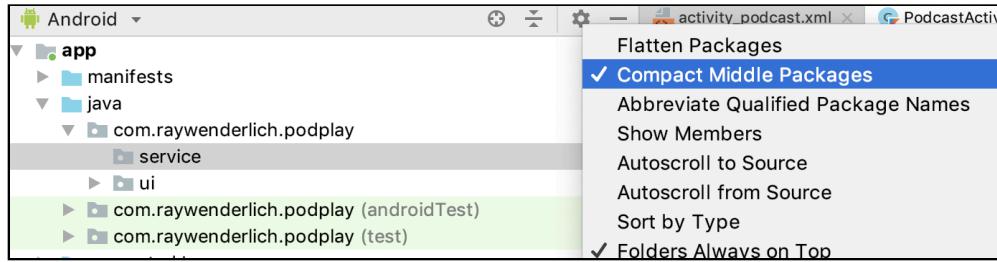
Now you'll create the model that represents a response from the iTunes service.

Create a new package named **service** inside the project root.

Note: To create the package inside `com.raywenderlich.podplay`, you may need to change the settings in the Project navigator to disable “**Compact Middle Packages**”.



Once you add the **service** package, you can re-enable the **Compact Middle Packages**, and your project structure will look like this:



In the service package, create a new Kotlin file named **PodcastResponse.kt**, and then replace the contents with the following:

```
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 ignores 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 a matter of personal preference, but this particular model is 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:

```
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.

Time to 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")` results in Retrofit using a final URL of `/search?media=podcast&term=Android+Developer`. Retrofit automatically URL-Encodes the parameter names and values when constructing the URL.

3. You define a companion object in the `ItunesService` interface.
4. The `instance` property of the companion object holds the 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 simply references `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 only performs 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's a simple example (don't type in this code):

```
class SomeClass: {
    val someProperty: String by SomeDelegateClass()
}
```

`SomeDelegateClass` must provide `setValue()` and `getValue()`. `get()` and `set()` for `someProperty` is delegated to `setValue()` and `getValue()`. Here's a simple implementation of `SomeDelegateClass` (don't type in this code):

```
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 you want to learn more, 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 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 to specify several options that let Retrofit know how it should ultimately create the concrete implementation of the `ItunesService` interface. In this case, you specify the following options:

baseUrl: Sets the base URL for the service. This is 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, you call `create<ItunesService>()` 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.

The next step is to hide the service behind a repository as you did with the database in PlaceBook. The repository is the only part of the app that touches the `ItunesService`.

Create a new package named **repository** inside the project root. Inside that package, create a new file named **ItunesRepo.kt**, and replace the contents with the following:

```
// 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)
        }
    })
}
```

Note: If you have any unresolved references with multiple choices for resolving, make sure to resolve them from the retrofit library.

1. You define the primary constructor for `ItunesRepo` to require an existing instance of the `ItunesService` interface. This is an example of the Dependency Injection principle. By passing an `ItunesService` to `ItunesRepo`, it makes it possible for the calling code to pass a different implementation for `ItunesService`. `ItunesRepo` 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 first parameter, and a method as the second parameter. The method defines a single parameter as a List of `iTunesPodcast` objects.
3. You call `searchPodcastByTerm()` and pass in the search term. This returns a Retrofit `Call` object.
4. You invoke `enqueue` on the `Call` object, and it runs in the background to retrieve the response from the web service. `enqueue` takes a Retrofit `CallBack` interface that defines two callback methods: `onFailure()` and `onResponse()`.
5. You call `onFailure()` if anything goes wrong with the call such as a network error or an invalid URL.

6. If there's an error, you call `callBack()` with a `null` value.
7. If the call succeeds, you call `onResponse()`.
8. You retrieve the populated `PodcastResponse` model with a call to `response.body()`.
9. You call `callBack()` 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 only the resulting `ItunesPodcast` object.

To test if the service is working, you can use `ItunesRepo` to search for a podcast and log the results.

Open `PodcastActivity.kt` and add the following to `onCreate()`:

```
val TAG = javaClass.simpleName  
val itunesService = ItunesService.instance  
val itunesRepo = ItunesRepo(itunesService)  
  
itunesRepo.searchByTerm("Android Developer", {  
    Log.i(TAG, "Results = $it")  
})
```

This code uses `ItunesRepo` to search for the podcast and prints the results to the Logcat window.

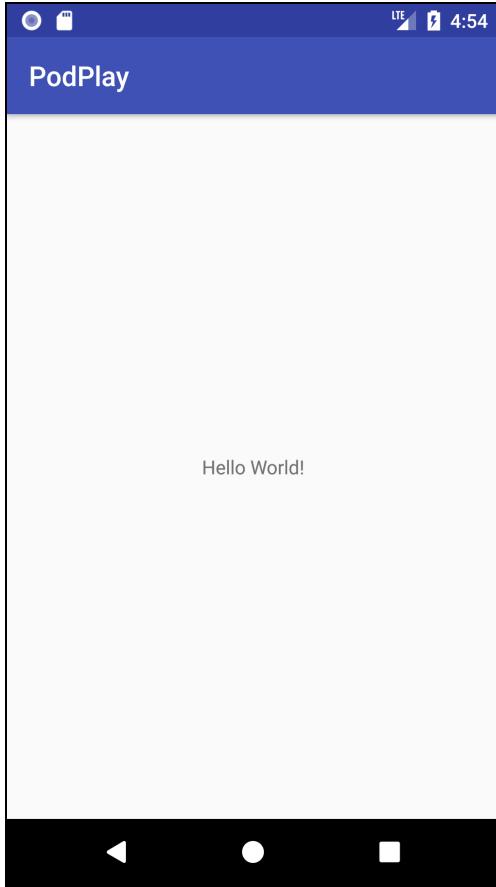
`ItunesService.instance` is called to get an instance of the `ItunesService` and it's passed to a new `ItunesRepo` instance. `searchByTerm()` is called with the search term and is passed an anonymous method to receive the results.

Before you run the app for the first time, you need to give it permission to use the internet.

Open `AndroidManifest.xml` and add the following before the `<Application>` section:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Build and run the app, and you'll see the default “Hello World” screen.



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)]
```

Congratulations, the service is working! The response displays the list of ItunesPodcast objects based on the search term.

Where to go from here?

The term **dependency injection** was mentioned briefly when you created the **iTunesRepo** class. You used a simple form of dependency injection when you passed in the **ItunesService** instance to the **iTunesRepo** constructor.

As your projects get more complicated it can be useful to have objects created and managed by a dependency injection library. Two of the most popular libraries for Android are **Dagger** and **Koin**.

Dagger is a Java based library that has been around for many years. Koin is a newer library written in Kotlin that takes advantage of Kotlin features.

You can learn more about Dagger in the following tutorials:

- <https://www.raywenderlich.com/265010-getting-started-with-dagger>
- <https://www.raywenderlich.com/265117-dagger-network-injection>

You can learn more about Koin with the following tutorial:

- <https://www.raywenderlich.com/5730-dependency-injection-with-koin>

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 for searching iTunes is complete, you’re ready to build out an interface that allows users to search for podcasts. Your goal is to provide a search box at the top of the screen where users can enter a search term. You’ll use the `ItunesRepo` you created in the last chapter to fetch the list of matching podcasts. From there, you’ll display the results in a `RecyclerView`, including the podcast artwork.

Although you can create a simple search interface by adding a text view that responds to the entered text, and then populating a `RecyclerView` with the results, the Android SDK provides a built-in search feature that helps future-proof your apps.

Android search

If you’re following along with your own app, open it and keep using it with this chapter. If not, don’t worry. Locate the **projects** folder for this chapter and open the **PodPlay** app inside the **starter** folder.

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

Android’s search functionality provides part of the search interface. You can display it either as a **search dialog** at the top of an Activity or as a **search widget**, which you can then place 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.

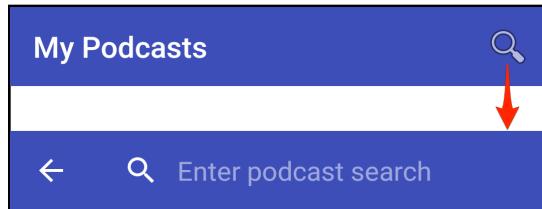
Some benefits to using Android search include:

- Displaying suggestions based on previous queries.
- Displaying suggestions based on 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 what you'll do in PodPlay. In other words, 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 you add a search widget as an action view, it displays a collapsible 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` with some hint text and a back arrow that's used to close the search.



Implementing search

To implement search capabilities, 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 app.
- **hint**: A hint that 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 label is the only required attribute. Because you're implementing a basic search for PodPlay, you'll 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, you need to name the search configuration file **searchable.xml**, and you must store it in **res/xml**.

To create this file in the proper location, right-click on **res** 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

File name:	searchable
Resource type:	XML
Root element:	searchable
Source set:	main
Directory name:	xml

Click **OK**. This creates the file and the **xml** resource directory. Now, replace the contents of **searchable.xml** with this:

```
<?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 displays an error for the missing `@string/search_hint` resource. To fix this, open `res/values/strings.xml` and add the following line:

```
<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 user'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 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 the 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.

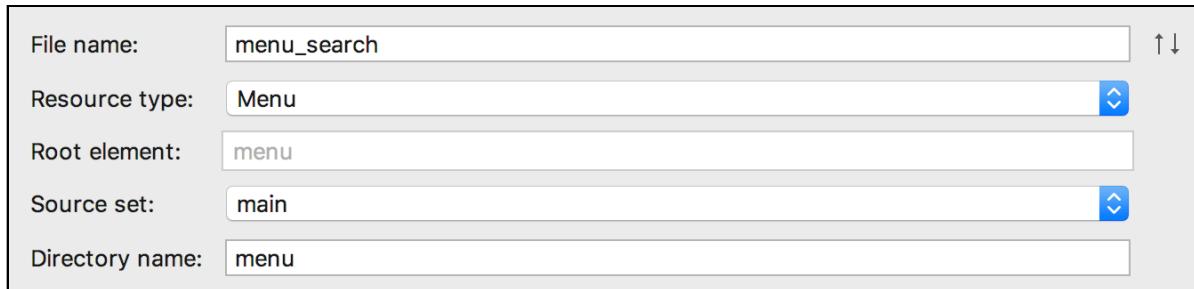
Open `app/manifests/AndroidManifest.xml` and update the `PodcastActivity` element to match this:

```
<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 you'll show the search widget as an action view in the app bar, you need to define an options menu with a single search button item. To do this, right-click on **res**, then select **New ▶ Android Resource File**.

Set the resource type to **Menu**, which automatically sets the root element type to **menu** and the folder to **menu**. Name the file **menu_search**:



Click **OK**, then open **res/menu/menu_search.xml** and replace the existing contents with the following:

```
<?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">

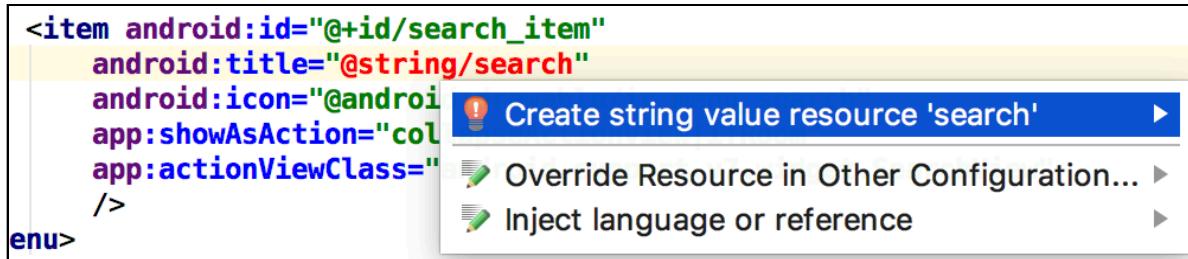
    <item android:id="@+id/search_item"
        android:title="@string/search"
        android:icon="@android:drawable/ic_menu_search"
        app:showAsAction=
            "collapseActionView|ifRoom"
        app:actionViewClass=
            "android.support.v7.widget.SearchView"/>
</menu>
```

This defines an options menu with a single `menu_search` item that's shown as an action view and uses the built-in `ic_menu_search` icon from the Android operating system.

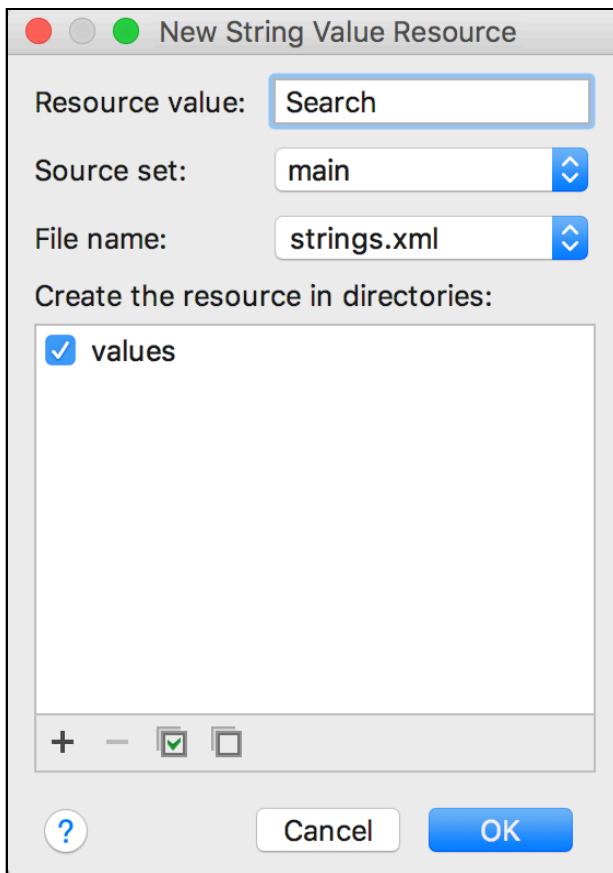
The `showAsAction` pipe-separated options are set to collapse the action view by default and display in the app bar if there's room. The `actionViewClass` must be set as `android.support.v7.widget.SearchView` since you want your shiny search bar to be backward-compatible with older versions of Android.

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 macOS or **Alt-Enter** on Windows to bring up the context menu, and select **Create string value resource 'search'**:



In the dialog that appears, type **Search** for the **Resource value** and click **OK**.



This adds the appropriate line to **strings.xml**, and the menu file updates so that all of 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 `onCreateOptionsMenu()`; note that you do not need to call `super`:

```
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.

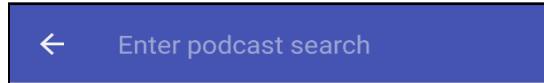
What's happening in this code?

1. First, you inflate the options menu. 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. You use `searchManager` to load the search configuration and assign it to the `searchView`.

Build and run the app, and you'll see a search icon in the app bar.



Tap the search icon, and it expands 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 disappears, 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()`, 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)
    }
}
```

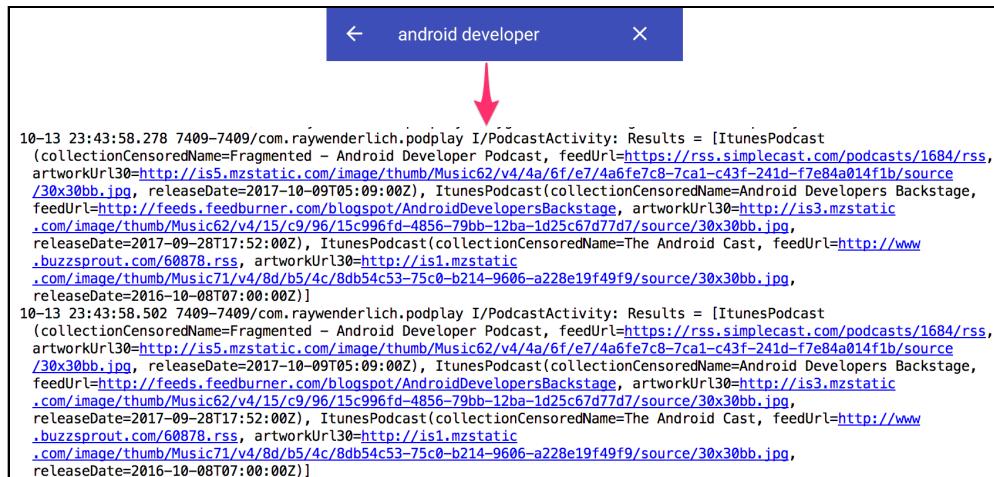
This method takes in an Intent and checks to see if it's an `ACTION_SEARCH`. If so, it extracts the search query string and passes it to `performSearch()`.

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 is 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 are written to the Logcat window:



Now that you're getting the search results 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 the same technique used in PlaceBook; to save time, the dependencies are already set up, but there are still a few things that need to be done:

Appcompat app bar

Open the project's `build.gradle` and add the following to the `ext` section:

```
support_lib_version = '28.0.0'
```

Open the module's `build.gradle` 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:$support_lib_version"
implementation "com.android.support:recyclerview-v7:$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"
    app:layout_constraintTop_toTopOf="parent"
    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 Kotlin Extensions:

```
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 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’s **build.gradle** and add the following to the **ext** section:

```
architecture_version = '1.1.1'
```

Open the module's **build.gradle** and add the following to the dependencies section:

```
implementation "android.arch.lifecycle:extensions:$architecture_version"
```

Right-click **com.raywenderlich.podplay**, and create a new package named **viewmodel** to help keep your view models organized. Add new empty Kotlin file inside **viewmodel** and name it **SearchViewModel.kt**.

Open it, and set up the initial search view model class:

```
class SearchViewModel(application: Application) :  
    AndroidViewModel(application) {  
}
```

The **AndroidViewModel** superclass requires the 'application' parameter. 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, **PodcastActivity** — passes this object in before calling any method to fetch the data.

Next, define a data class within the view model that has only the data that's 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 eventually gets 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
            callback(searchViews)
        }
    })
}
```

Going 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 you pass an empty list to the `callback` method.
4. If the results are not `null`, then you map them to `PodcastSummaryViewData` objects. This follows the principle of providing the View with just enough data for presentation.
5. You pass the mapped results to the `callback` method so you can display them.

Next, you need to add the `RecyclerView` to display the search results.

Results RecyclerView

First, define the Layout for a single search result item. Create a new file inside `res/layout` and name it `search_item.xml`. Then, set the contents to the following:

```
<?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"
```

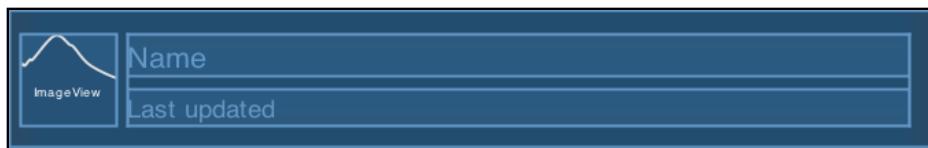
```
    android:layout_height="wrap_content"
    android:paddingTop="10dp"
    android:paddingBottom="10dp"
    android:paddingLeft="5dp"
    android:paddingRight="5dp">
<ImageView
    android:id="@+id/podcastImage"
    android:layout_width="40dp"
    android:layout_height="40dp"
    android:layout_marginEnd="5dp"
    android:adjustViewBounds="true"
    android:scaleType="fitStart"/>

<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginEnd="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, as well as 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:

```
<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 can return up to 50 results.

If you pre-fetch the image for each one and store it locally or in memory, it won't make for an enjoyable user experience; there could potentially be a considerable delay before any results would show up. You could 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 Google recommends is **Glide**.

Glide was developed to make image scrolling as smooth as possible, but you can use it 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 specify a context, the remote image URL and a View to place the image. Glide handles all of the details, including background loading and canceling the image load when the parent View disappears.

To use Glide, add the following to the dependencies section in the module's **build.gradle**:

```
implementation "com.github.bumptech.glide:glide:4.9.0"
```

Create a new package inside **com.raywenderlich.podplay** and name it **adapter**. Add a new Kotlin file to this package and name it **PodcastListAdapter.kt**. Finally, update it with the following contents:

```
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 code was covered in earlier chapters on RecyclerViews, so there's no need to go over the details.

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 load the podcast image into the image view efficiently. 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'll 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`. You can add several other calls 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, only 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 **ViewModelProviders** 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** 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 is 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 need to do in `PodcastActivity.kt` is update `performSearch()` to use the view model you set up:

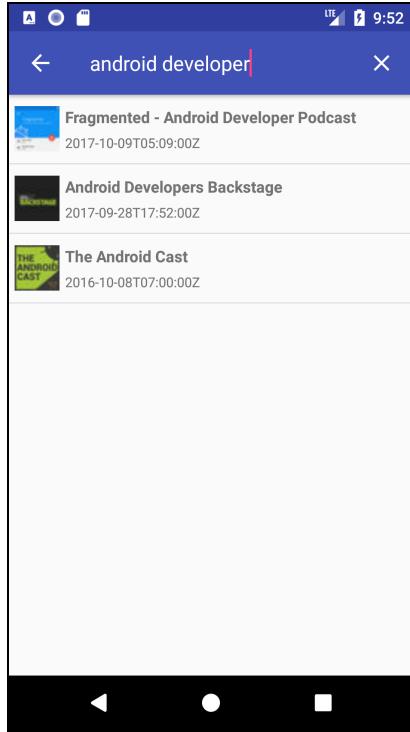
```
private fun performSearch(term: String) {  
    showProgressBar()  
    searchViewModel.searchPodcasts(term, { results ->  
        hideProgressBar()  
        toolbar.title = getString(R.string.search_results)  
        podcastListAdapter.setSearchData(results)  
    })  
}
```

Finally, add the following line to `res/values/strings.xml` 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 notice the movement remains 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 inside **com.raywenderlich.podplay** and name it **util**. Next, add a new Kotlin file and name it **DateUtils.kt** with the following contents:

```
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: Be sure to 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.

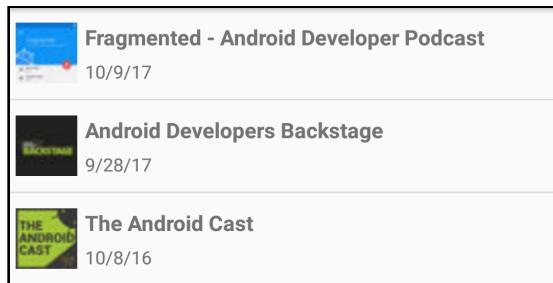
1. First, check that the `jsonDate` string coming in is not `null`. If it is, return a string, which doesn't need to be translated (to avoid calling into Android Resources), indicating that no date was provided.
2. Define a `SimpleDateFormat` to match the date format returned by iTunes.
3. Parse `jsonDate` string and place it 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),
```

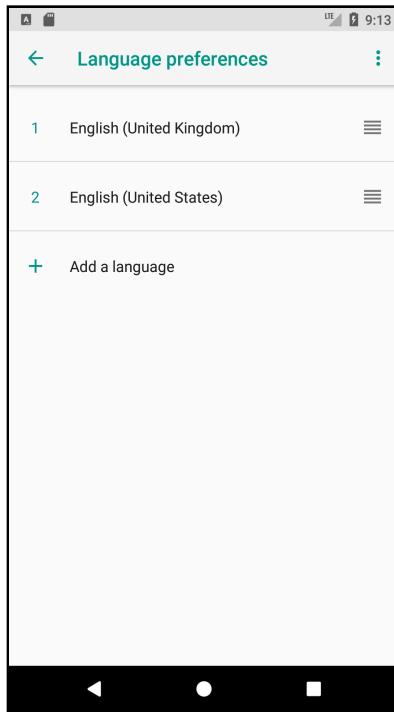
You're calling `jsonDateToShortDate()` to convert the date before it's returned from the `SearchViewModel` — that way the View never has to know that the date has been formatted, but it will still look much nicer to the user.

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.

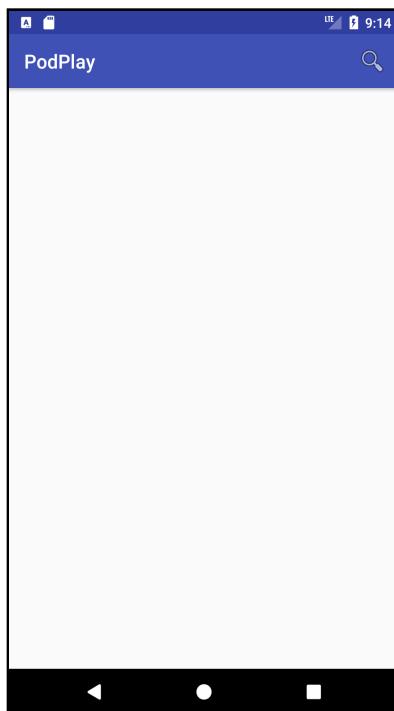


For instance, if you're in the US, the date is formatted similar to the screenshots above, because `en-US` is most likely your default locale. If you're in a country that uses Day/Month/Year formatting, such as the UK, then the date is formatted as 28/9/17 instead of 9/28/17.

Want to double check? Go 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 you'll see this:



Hey, what happened to the search results?

It 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()` pays off. You can grab the saved Intent when the Activity restarts and then redo the search.

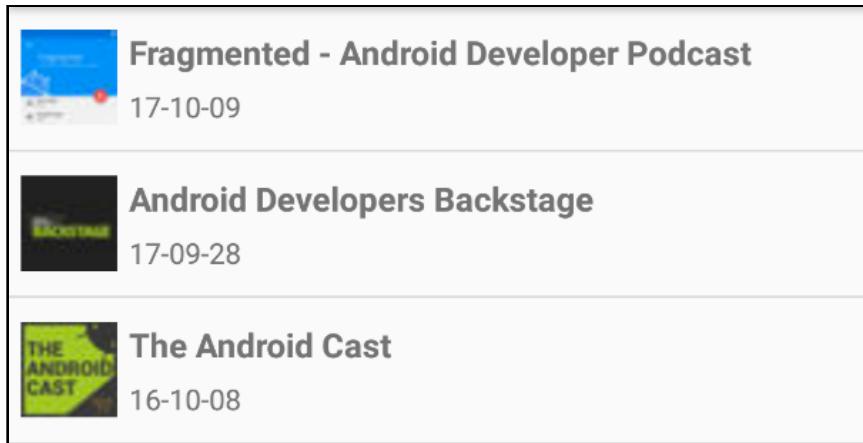
Open `PodcastActivity.kt` and add the following line to the end of `onCreate()`:

```
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, is now 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.

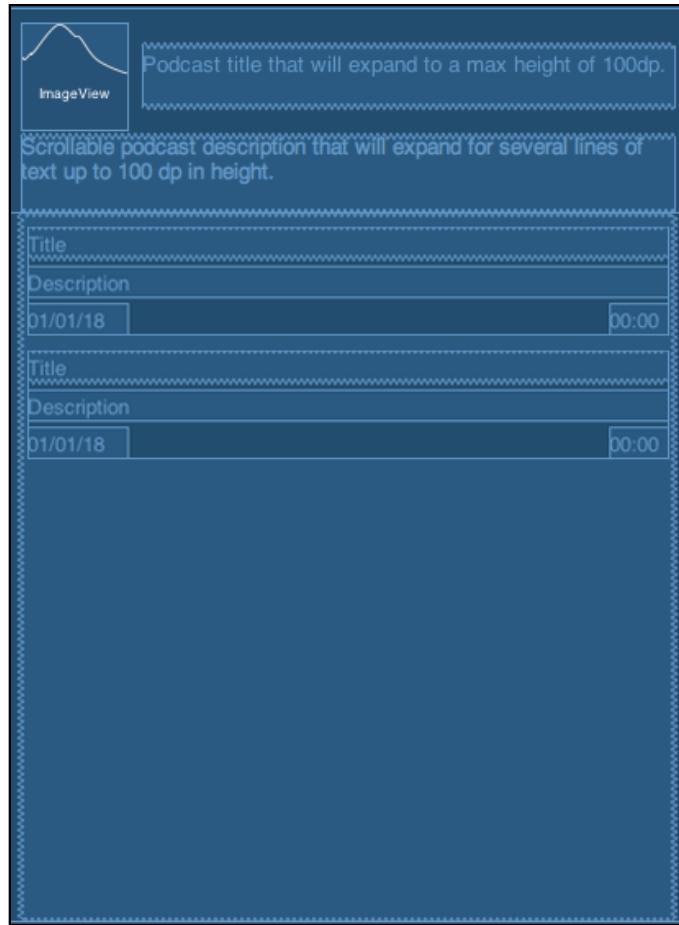
Getting started

If you're following along with your own project, open it and keep using it with this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **PodPlay** project inside the **starter** folder.

The first time you open the project, Android Studio takes a few minutes to set up your environment and update its 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 define 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 evident as you build out the full user interface in later chapters.

Defining the Layouts

Create a new Layout and name it `fragment_podcast_details.xml`. 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 and name it **episode_item.xml**. 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 **activity_podcast.xml** 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 holds the podcast detail Fragment. It's configured to cover the entire Activity View below the app bar. Nothing displays in the container until you load the podcast detail Fragment, which happens after the user taps on a podcast row.

Basic architecture

As in previous chapters, you need to define the basic architecture components consisting of a repository, a service and a view model to display the podcast detail. There's no need for any database layer at this point.

You'll start with a basic implementation to get the navigation working.

Podcast models

To store the podcast data, you need two models: 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 inside **com.raywenderlich.podplay** and name it **model**.

Inside **model**, create a new file and name it **Episode.kt**. Replace the contents with the

following:

```
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. Here's an explanation for each property:

- **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 is either an audio or video file.
- **mimeType**: Determines the type of file located at `mediaUrl`.
- **releaseDate**: Date the episode was released.
- **duration**: Duration of the episode as provided in the RSS feed.

Still inside `model`, create another new file and name it **Podcast.kt**. Replace the contents with the following:

```
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. Here's an explanation of each property:

- **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

You'll use a repo for retrieving the podcast details and returning it to the view model.

Inside **repository**, create a new file and name it **PodcastRepo.kt**. Replace the contents with the following:

```
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; but for now, a simple version of the `Podcast` object is created and passed to the `callback` method.

Podcast view model

Inside **viewmodel**, create a new file and name it **PodcastViewModel.kt**. Replace the contents with the following:

```
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` is set by the caller. The property `activePodcastViewData` holds the most recently loaded podcast view data. `PodcastViewData` contains everything you need to display the details of a podcast.

The repo returns a list of `Episode` models, so you 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 into a list.

You also need a method to convert `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)
        }
    })
}
```

Here's 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. Call `getPodcast()` from the podcast repo with the feed URL.
4. Check the podcast detail object to make sure it's not `null`.
5. Set the podcast title 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. Set the podcast detail image to match the podcast summary image URL if it's not `null`.
7. Convert the `Podcast` object to a `PodcastViewData` object and assign it to `activePodcastViewData`.
8. Call the `callback` method and pass the podcast view data.

Details Fragment

The detail Fragment is responsible for displaying the podcast details; it gets its data from `PodcastViewModel`. This is also where the user can subscribe to a podcast. First, you need to add an action menu with a single `Subscribe` item.

Open `strings.xml` and add the following line:

```
<string name="subscribe">Subscribe</string>
```

Create a menu resource file and name it `menu_details.xml`. Replace the contents with the following:

```
<?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 displays when the details Fragment is active. It contains a single item with the label “Subscribe”.

Inside `ui`, create a new file and name it `PodcastDetailsFragment.kt`. Replace the contents with the following:

```
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 the standard procedure for setting up a Fragment, except for a few important details:

1. The call to `setHasOptionsMenu()` tells Android that this Fragment wants to add items to the options menu. This causes the Fragment to receive a call to `onCreateOptionsMenu()`.
2. `onCreateOptionsMenu()` inflates the `menu_details` options menu so its items are added to the podcast Activity menu.

Next, you need to 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() {
    activity?.let { activity ->
        podcastViewModel = ViewModelProviders.of(activity)
            .get(PodcastViewModel::class.java)
    }
}
```

This retrieves an instance of `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 get the same instance of the `PodcastViewModel` that was created in `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: You can seamlessly share view models with any Fragments managed by the Activity. View models can also survive configuration changes, so you don't need to create them again when 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 set up. 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
    activity?.let { activity ->
        Glide.with(activity).load(ViewData.imageUrl)
            .into(feedImageView)
    }
}
```

Note: If Android Studio complains about not being able to resolve `feedTitleTextView` and `feedDescTextView`, add `import kotlinc.android.synthetic.main.fragment_podcast_details.*` to the top of the file.

This first line checks to make sure there's view data available (and that you have something in `activePodcastViewData`, which you 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 is already 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 `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 `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, read 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 either creates the details Fragment or uses an existing instance if one exists. Here's a closer look at how this works:

1. You use `supportFragmentManager.findFragmentByTag()` to check if the Fragment already exists.

2. If there's no existing fragment, you create a new one using `newInstance()` on the Fragment's companion object.
3. You return the Fragment object.

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
```

You're ready to 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  
}
```

Here's 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 that the search icon is not shown on the details screen.

Note: Adding the Fragment to the back stack is essential for proper app navigation. If you don't add the call to `addToBackStack()`, then pressing the back button while the Fragment is displayed closes 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 gets 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 `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 need to create the `PodcastViewModel` that’s 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` creates a new instance of the `PodcastViewModel` object. If it's just a configuration change, it uses an existing copy of the `PodcastViewModel` object instead.

The `podcastViewModel` object is now ready to use when `onShowDetails()` is called in response to the user tapping on a podcast row. Time to code that!

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")
        }
    })
}
```

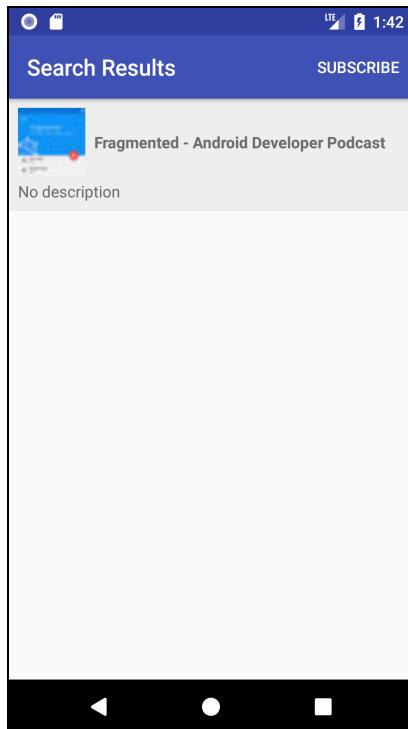
This method is called when the user taps on a podcast. Here's how it works:

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 appears 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
        }
    }
}
```

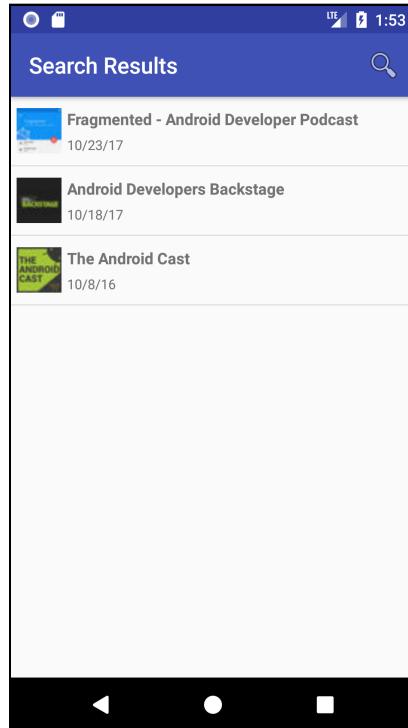
This adds a lambda method that can respond to changes in the Fragment back stack. This is called when items are added or removed from the stack. If the `backStackEntryCount` is `0`, then all Fragments have been removed, and it's safe to make the podcast RecyclerView visible again.

Add the following line to the end of `onCreate()`:

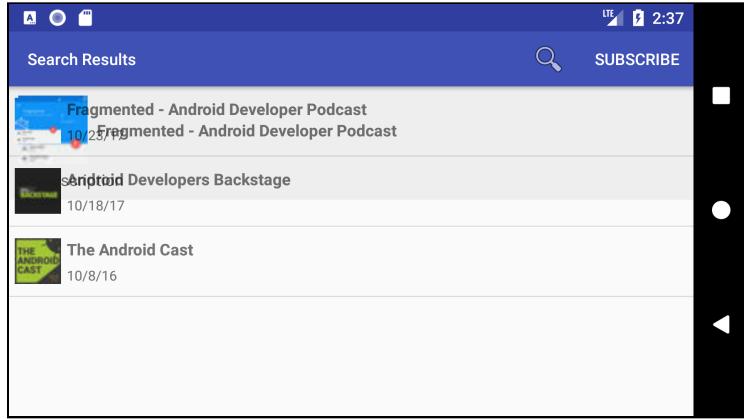
```
addBackStackListener()
```

This adds 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 now looks correct.



Before you call it a day, try to rotate the screen. You'll get an interesting mash-up of the search results and the podcast details. Whoops!



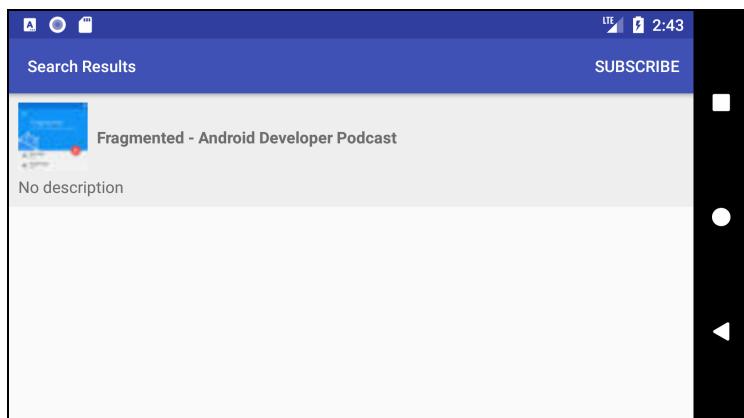
As this test demonstrates, the Android UI is not complete until you've tested it by rotating the screen. Fortunately, this is an easy fix: You need to hide the podcast RecyclerView after a configuration change.

In `onCreateOptionsMenu()`, after the line that calls `searchView.setSearchableInfo()`, add the following:

```
if (supportFragmentManager.backStackEntryCount > 0) {  
    podcastRecyclerView.setVisibility(View.INVISIBLE)  
}
```

Now, when the device rotates, the Activity gets 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 now looks 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're following along with your own project, open it and keep using it with this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **PodPlay** project inside the **starter** folder.

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

Getting started

In previous chapters, you worked with the iTunes Search API, which is excellent 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. However, 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 you can find at <http://www.rssboard.org/rss-specification>.

Let's take a look at a sample RSS file for a fictitious podcast:

```
<?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">15860824851599</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. However, 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 and name it `RssFeedResponse.kt`. Then, add the following:

```
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 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 and name it **RssFeedService.kt**. Then, add the following:

```
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. You use a companion object to provide a singleton instance of the `FeedService`.

Next, you'll start implementing `getFeed()`.

The first task is to download the RSS file, but there's one small issue to address first.

Starting with Android 9 (API Level 28), by default, apps may not use cleartext network traffic. Cleartext traffic results from connections where the URL starts with HTTP, not HTTPS. Since you cannot control the URL of the podcast feed, you'll set a flag that allows the app to use cleartext traffic.

Open `AndroidManifest.xml` and add the following as part of the `application` element header:

```
    android:usesCleartextTraffic="true">
```

Now, you're ready to write some code to fetch the podcast feed.

Add the following 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.

Time to break the code apart:

1. You create a new instance of `OkHttpClient`. You'll use the OkHttp client to fetch the RSS file asynchronously. This ensures 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 specify 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. You pass a `Callback` object to `enqueue()`. When the Request is complete, OkHttp calls either `onFailure()` or `onResponse()` on the callback object.
4. You define `onFailure()` 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. You check the response 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. You check the response body for null.
8. You convert responseBody to a string and print it 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 single 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 a `java.lang.IllegalStateException: closed` exception!

To test `getFeed()`, 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. Look at the Logcat window, and view the output of the RSS XML file.

```
I/System.out: <?xml version="1.0" encoding="UTF-8"?>  
I/System.out: <rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom" xmlns:content="http://purl.org/rss/1.0/modules/  
I/System.out:   <channel>  
I/System.out:     <atom:link rel="self" type="application/atom+xml" href="https://rss.simplecast.com/podcasts/1684/rss" t  
I/System.out:     <title>Fragmented – Android Developer Podcast</title>  
I/System.out:     <generator>https://simplecast.com</generator>  
I/System.out:     <description>The Fragmented Podcast is a podcast for Android Developers hosted by Donn Felker and Kaush  
I/System.out:     <copyright>© 2016 Spec Network, Inc.</copyright>  
I/System.out:     <language>en-us</language>  
I/System.out:     <pubDate>Mon, 06 Nov 2017 05:00:00 +0000</pubDate>  
I/System.out:     <lastBuildDate>Mon, 06 Nov 2017 05:00:54 +0000</lastBuildDate>  
I/System.out:     <link>http://www.fragmentedpodcast.com</link>  
I/System.out:     <image>  
I/System.out:       <url>https://media.simplecast.com/podcast/image/1684/1474255312-artwork.jpg</url>  
I/System.out:       <title>Fragmented – Android Developer Podcast</title>  
I/System.out:       <link>http://www.fragmentedpodcast.com</link>  
I/System.out:     </image>  
I/System.out:     <itunes:new-feed-url>https://rss.simplecast.com/podcasts/1684/rss</itunes:new-feed-url>  
I/System.out:     <itunes:author>Spec</itunes:author>  
I/System.out:     <itunes:image href="https://media.simplecast.com/podcast/image/1684/1474255312-artwork.jpg"/>  
I/System.out:     <itunes:summary>The Fragmented Podcast is a podcast for Android Developers hosted by Donn Felker and Kaush  
I/System.out:     <itunes:subtitle>The Fragmented Podcast is a podcast for Android Developers hosted by Donn Felker and Kaush  
I/System.out:     <itunes:explicit>no</itunes:explicit>  
I/System.out:     <itunes:keywords>android, developer, podcast, java, AndroidDev</itunes:keywords>  
I/System.out:     <itunes:type>episodic</itunes:type>  
I/System.out:     <itunes:owner>  
I/System.out:       <itunes:name>Spec Network, Inc.</itunes:name>  
I/System.out:       <itunes:email>shows@spec.fm</itunes:email>  
I/System.out:     </itunes:owner>  
I/System.out:     <itunes:category text="Technology"/>  
I/System.out:     <itunes:category text="Technology">  
I/System.out:       <itunes:category text="Podcasting"/>
```

XML to DOM

Even though you can use Retrofit to parse XML — and it comes with a built-in XML parser — there are too many edge cases to make Retrofit usable as-is; you need to handle namespaces and ignore duplicate elements properly. 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 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 a 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.

\$[=s=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

It's time 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 **RssFeedService**:

```
private fun domToRssFeedResponse(node: Node,
    rssFeedResponse: RssFeedResponse) {
    // 1
    if (node.nodeType == Node.ELEMENT_NODE) {
        // 2
        val nodeName = node.nodeName
        val parentNode = node.parentNode.nodeName
        // 3
        if (parentNode == "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'll become more clear when you add episode item parsing next.

Here's what's going on with this code:

1. First, you check the `nodeType` 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, you add a new empty `EpisodeResponse` object to the `episodes` list.
5. You assign `nodeList` 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. Notice that the RssFeedResponse top-level information was populated, along with a series of blank episode items.

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
    }
}
}
```

Here's 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. You assign `currentItem` 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, you extract the `url` and `type` 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. Notice 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!

You can now 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 you'll pass into the constructor.

Now, you 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. You assign the list of episodes to `items` 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. You create a new `Podcast` object using the response data and then return it to the caller.

Now you can update `getPodcast()` 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 from the `callBack` method, you'll use coroutines to jump back to the main thread before returning the podcast details from the podcast repo.

First, you need to include the coroutines library.

Open the project **build.gradle** file and add the following to the `ext` element:

```
coroutines_version = '1.1.0'
```

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"
```

In previous uses of coroutines, you only imported `kotlinx-coroutines-core`. To assist in UI programming with coroutines, you're now also importing `kotlinx-coroutines-android`. This will provide the `Dispatchers.Main` context used below.

Now, sync the project.

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)  
    }  
  
    GlobalScope.launch(Dispatchers.Main) {
```

```
        callback(podcast)
    }
})
```

If the `feedResponse` is `null`, you pass `null` to the `callBack` method. If `feedResponse` is valid, then you convert it to a `Podcast` object and pass it to the `callback` method.

Note that the calls to the `callback` method are surrounded with `GlobalScope.launch(Dispatchers.Main)`. This passes the `Dispatchers.main` context to the launch command, forcing the enclosing code to run on the main thread. As mentioned earlier, trying to update any part of the UI from a background thread will produce unexpected results.

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 need to add a list Adapter to populate the `RecyclerView` using `EpisodeViewData` items.

In the **adapter** package, create a new file and name it `EpisodeListAdapter.kt`. Replace the contents with the following:

```
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)
    }

    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  
}
```

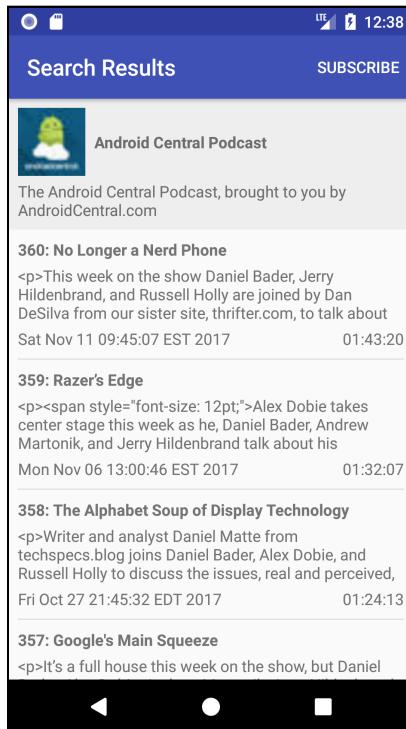
Here's what's going on:

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. You create the EpisodeListAdapter with the list of episodes in activePodcastViewData and assign it to 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 needs some extra processing. You also need to format the dates on the episodes. To fix the HTML formatting, create a utility method that uses a built-in Android 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 and name it **HtmlUtils.kt**. 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 {
            @SuppressLint("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 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 it's still required when running on Android M or lower. `@SuppressLint("DEPRECATION")` is used to allow the code to compile even though it is deprecated.

Next, 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, you need to add a new helper method to convert a `Date` object to a short date formatted string. Open **DateUtils.kt** and add the following method:

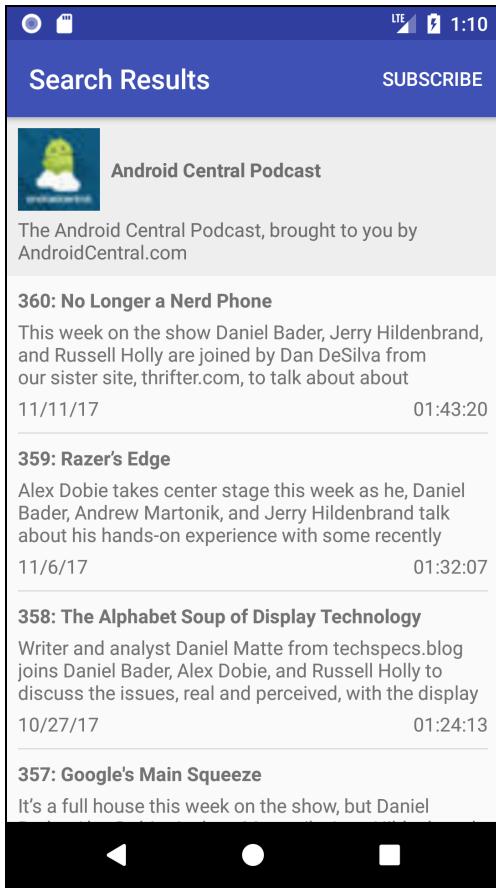
```
fun dateToShortDate(date: Date): String {  
    val outputFormat = DateFormat.getDateInstance(  
        DateFormat.SHORT, Locale.getDefault())  
    return outputFormat.format(date)  
}
```

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 significant 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 throughout these two chapters including:

1. Using Room to store multiple related database tables.
2. Using JobScheduler services to check for new episodes periodically.
3. Using local notifications to alert users when new episodes are available.

Getting started

If you're following along with your own project, open it and keep using it with this chapter. If not, don't worry. Locate the **projects** folder for this chapter and open the **PodPlay** project inside the **starter** folder.

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

Saving podcasts

The first new feature you'll implement is the ability to track podcast subscriptions. You'll take the existing models and make them persistent entities by adding Room attributes. The database will only contain podcasts to which the user subscribes.

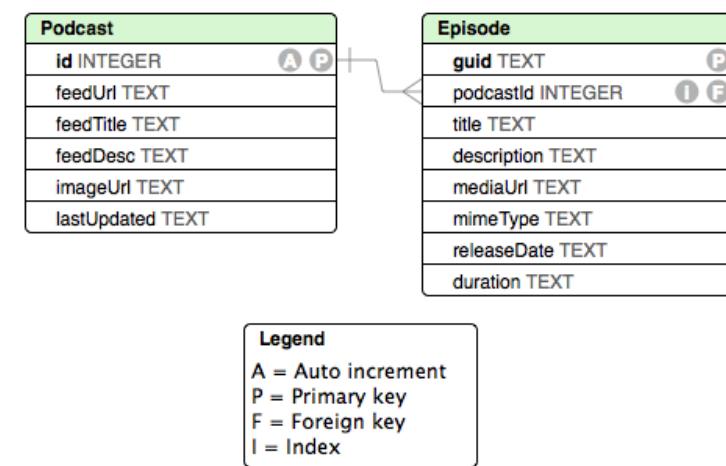
Your first goal is to hook up the subscribe menu item so that it saves the current podcast.

Setting up the database code follows the same general approach used in MapBook:

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 are slightly more difficult this time around because you have two models — podcast and episode — to manage instead of only one. You also have to manage the relationship between these two models. For example, if you delete a podcast from the database, all associated episodes should also be deleted from the database. However, don't fret! This is only slightly more difficult because Room does all of 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 — you're adding a foreign ID (podcastId) pointing the model back to a Podcast model. You'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 = '1.1.0'
```

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.lifecycle:compiler:
$architecture_version"
kapt "android.arch.persistence.room:compiler:$room_version"
```

These are the same libraries you used in PlaceBook. 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 that 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 need to add a primary key to the `Podcast` table.

Add the following as the first property declaration to `Podcast`:

```
@PrimaryKey(autoGenerate = true) var id: Long? = null,
```

This defines an `id` that auto-generates 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 it's time to bring `Episode` 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`, select the following versions respectively:

```
import android.arch.persistence.room.ForeignKeyimport  
android.arch.persistence.room.Index
```

Here, you're adding some 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 you delete a podcast, all related child episodes are 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 define `podcastId` as the index.

There's 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 `Podcast` 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 `guid` with the `@PrimaryKey` annotation as follows:

```
@PrimaryKey var guid: String = "",
```

You need to add the `podcastId` property that defines the foreign key to the `Podcast` entity, so inside `Episode`, add the following property below `guid` and above `title`:

```
var podcastId: Long? = null,
```

Now that you've added 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 need to 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.

Inside `com.raywenderlich.podplay`, create a new package and name it `db`.

Next, create a new file and name it `PodcastDao.kt`, and place it inside `db`. Then, 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 = :podcastId
        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`, select the following versions respectively:

```
import android.arch.persistence.room.Query
import android.arch.persistence.room.OnConflictStrategy.REPLACE
```

Let's break the code down a bit:

1. You define `PodcastDao` interface 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.

In `db`, create a new file and name it **PodPlayDatabase.kt**. Replace the contents with the following:

```
// 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
        }
    }
}
```

Here's a closer look at what's happening:

1. You define `PodPlayDatabase` 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 handles 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() returns a single application-wide instance of the PodPlayDatabase.
6. If an instance of PodPlayDatabase hasn't been created before, it's created now. You use Room.databaseBuilder() to instantiate the PodPlayDatabase object.
7. You return the PodPlayDatabase object to the caller.

Build the project using Command-F9 (Control-F9 on Windows). You'll get the following errors from the compiler:

- Cannot figure out how to save this field into the 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 the models. Why is that? Because Room only knows how to deal with basic and boxed basic types, not complex types. A boxed basic type is one that is 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. It’s time to 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:

```
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`, 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 `PodPlayDatabase`, sandwich a `@TypeConverters` annotation between the `@Database` annotation and the class declaration, so it looks like this:

```
@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 recognizes the two methods for handling `Dates`, and it calls 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 won't attempt to populate it when loading a Podcast from the database.

Build the app again to verify the errors are gone.

That handles 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.

You're ready 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 correctly 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)
```

You create an instance of `PodPlayDatabase` and retrieve the `PodcastDao` object 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) {
    GlobalScope.launch {
        // 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.

Here's a closer look at how this works:

1. First, you insert the `Podcast` into the database. `insertPodcast()` returns the new primary key assigned to the podcast.
2. Using the `for` loop, you walk through each `episode` belonging to the podcast.

3. You assign the episode's podcastId to the id of the inserted Podcast to create a relationship between the two.
4. Finally, you insert the episode into 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 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 gets 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 assigns the `activePodcast` to the podcast loaded by `getPodcast()`. This allows the podcast view model to keep track of the most recently loaded podcast.

You can now 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 `activePodcast` 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  
}
```

Here's a closer look:

1. If `livePodcastData` is `null`, create it.
2. You retrieve the `LiveData` object 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. Convert the list of `LiveData` podcast objects to a list of `LiveData` `PodcastSummaryViewData` objects.
4. Return `livePodcastData` 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` requires 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 `listener` 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 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)
    }
}
```

You call `onOptionsItemSelected()` 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 you call `onSubscribe()` 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 isn't useful if you don't let the user see their subscriptions!

The main podcast Activity already contains a RecyclerView that displays a list of podcasts generated from search results. You can reuse the same RecyclerView 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 are replaced with the search results.

You'll start by updating the podcast Activity to load all of the podcasts and display them in the RecyclerView when the View is first created.

Open **PodcastActivity.kt** and add the following method:

```
private fun showSubscribedPodcasts()
{
    // 1
    val podcasts = podcastViewModel.getPodcasts()?.value
    // 2
    if (podcasts != null) {
        toolbar.title = getString(R.string.subscribed_podcasts)
        podcastListAdapter.setSearchData(podcasts)
    }
}
```

Here's what's going on with this code:

1. You call `getPodcasts()` 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 you update the podcast list Adapter with the `podcasts` object.

Add the following line to **strings.xml** to satisfy the `subscribed_podcasts` resource reference.

```
<string name="subscribed_podcasts">Subscribed</string>
```

Add the following method in **PodcastActivity.kt**:

```
private fun setupPodcastListView() {
    podcastViewModel.getPodcasts()?.observe(this, Observer {
        if (it != null) {
            showSubscribedPodcasts()
        }
    })
}
```

Note: If given import options on Observer, 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 need to call `setupPodcastListView()` when the view is created.

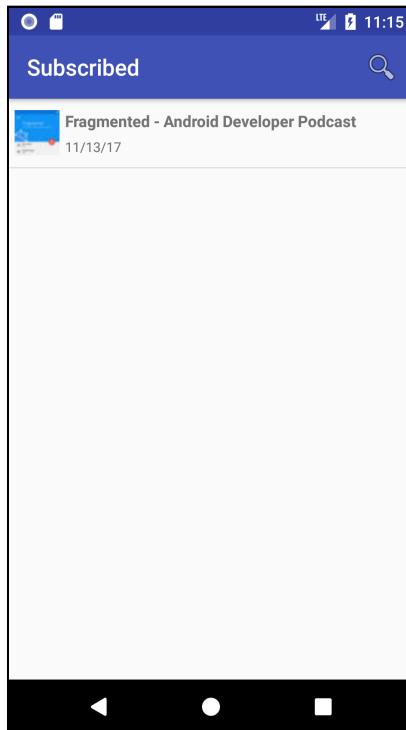
In `onCreate()`, add the following line after the call to `updateControls()`:

```
setupPodcastListView()
```

Build and run the app.

Search and display the details for a podcast. Tap the subscribe button, and the app returns to the search results.

Behind the scenes, the `Observer` you created in `setupPodcastListView()` is 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 reasonably well, but there are a few things that you need to clean 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 perform 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 = :url")
fun loadPodcast(url: String): Podcast?
```

Next, you need to update the repo logic so that it attempts to load from the database first.

Open **PodcastRepo.kt** and add the following to the beginning of `getPodcast()`:

```
GlobalScope.launch {
    val podcast = podcastDao.loadPodcast(feedUrl)
    if (podcast != null) {
        podcast.id?.let {
            podcast.episodes = podcastDao.loadEpisodes(it)
            GlobalScope.launch(Dispatchers.Main) {
                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 that 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()
```

To update the menu item text to display either “Subscribe” or “Unsubscribe” dynamically, you need to save the `MenuItem` in a local property.

Add the following property to the `PodcastDetailsFragment` class:

```
private var menuItem: MenuItem? = null
```

You need a new method 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()`.

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 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 automatically deletes 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) {  
    GlobalScope.launch {  
        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 `activePodcast` 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 rerunning it. If you don't do this, the database won't load the existing episodes correctly.

Build and run the app.

Tap a previously subscribed podcast to display the details screen. The menu action now shows “UNSUBSCRIBE”.

Tap “UNSUBSCRIBE”, and the app returns 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  
        }  
    }  
}
```

You define an `OnActionExpandListener` object with two required overrides and assign it 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, you call `showSubscribedPodcasts()` 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 returns 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

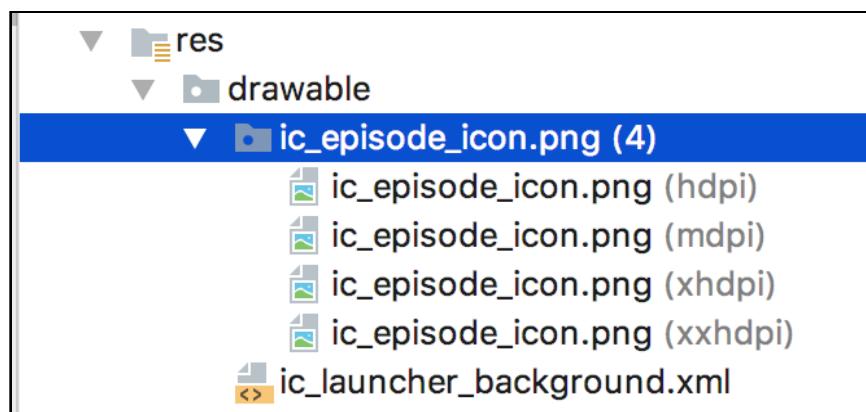
Now that the user can subscribe to podcasts, it's helpful to notify them when new episodes are available. In this chapter, you'll update the app to periodically check for new episodes in the background and post a notification if any are found.

Getting started

If you're following along with your own project, the starter project for this chapter includes an additional icon that you'll need to complete the section. Open your project then copy the following resources from the provided 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 will look like this:



If you don't have your own project, don't worry. Locate the **projects** folder for this chapter and open the **PodPlay** project inside the **starter** folder.

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

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:

Awaken

You can use `AlarmManager` 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 is 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 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 PodPlay.

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.

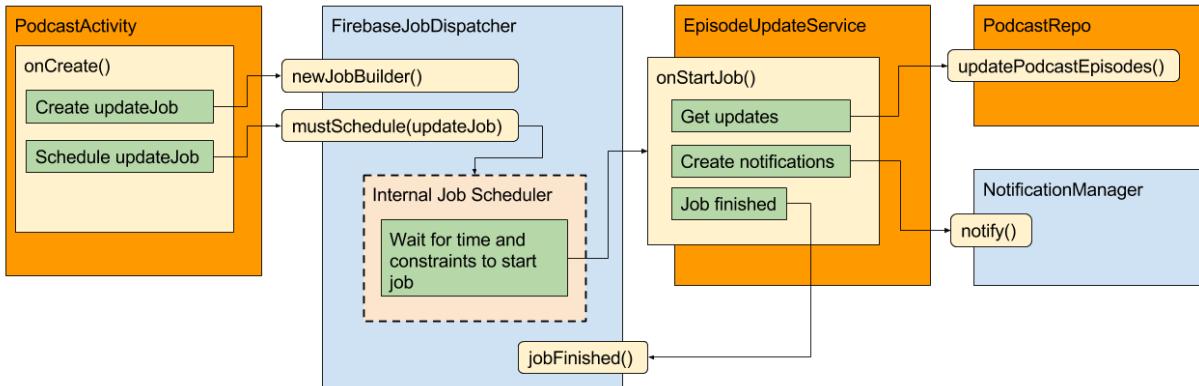
This means you can only use `JobScheduler` if you're targeting API 21 or higher. To support API 19, a nice alternative is the **Firebase JobDispatcher**.

Firebase JobDispatcher

Firebase JobDispatcher is an open source library created by Google that has a similar API to the `JobScheduler` but works with 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 first build out the underlying logic to update podcast episodes.

Here's a diagram showing how it all fits together:



Episode update logic

To keep with the current architecture of using the repo for updating podcast data, you need to 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 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:

```
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. Here's a step-by-step look at how this works:

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 not return a response.

You also need a new method that updates an existing podcast with a new episode.

Add the following method:

```
private fun saveNewEpisodes(podcastId: Long, episodes: List<Episode>) {
    GlobalScope.launch {
        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 PodcastRepo:

```
class PodcastUpdateInfo (val feedUrl: String, val name: String,  
    val newCount: Int)
```

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 run until the loop iterates over all podcasts and returns 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're 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 were 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 the `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.

You must add the Firebase JobDispatcher library to the project first.

Open the module **build.gradle** file and add the following line to the dependencies section:

```
implementation "com.firebaseio:firebase-jobdispatcher:0.8.5"
```

Sync the project.

In the **service** package, create a new Kotlin file and name it **EpisodeUpdateService.kt**. Replace the contents with the following:

```
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.firebaseio.jobdispatcher.JobService` and `com.firebaseio.jobdispatcher.JobParameters` instead of the built-in `android.app.job.JobService` and `android.app.job.JobParameters` imports.

Kotlin's auto-complete 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 the 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 `JobService`:

- **onStartJob()**: This is where you perform the episode updating logic. This method returns `true` if you're processing the job on a background thread, which you'll be doing. If your job does something simple and returns without starting a background thread, then you'll return `false`.

The job dispatcher calls this method when it's time for you to perform your work. A `WakeLock` is kept on your app 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, you must call the `jobFinished()` method on your `JobService` 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 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 releases 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 calls 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, you must register the job service 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.firebaseio.jobdispatcher.ACTION_EXECUTE"/>
    </intent-filter>
</service>
```

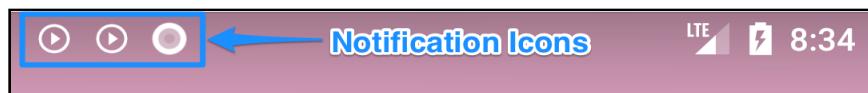
Now you can start adding some supporting methods to the job service.

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 `NotificationManager` to trigger notifications based on a `Notification` object that's created with `NotificationCompat.Builder`.

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 `EpisodeUpdateService`:

```
companion object {
    val EPISODE_CHANNEL_ID = "podplay_episodes_channel"
}
```

This defines a channel ID that identifies this channel to the notification system. This can be any string that is unique to your app.

Add the following method that creates 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)
    }
}
```

Here's the breakdown:

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 doesn't exist, then a new `NotificationChannel` object is created with the name "Episodes". The notification manager is instructed to create the channel.

It's time to 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, `$d` indicates that this first parameter is a digit. Similarly, `%2$s` indicates that the second parameter is a string.

In `EpisodeUpdateService.kt`, add a new constant to the companion object:

```
val EXTRA_FEED_URL = "PodcastFeedUrl"
```

Then, add the following method:

```
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`, 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 uses 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 lets 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:

```
// 1
val db = PodPlayDatabase.getInstance(this)
val repo = PodcastRepo(FeedService.instance, db.podcastDao())
// 2
GlobalScope.launch {
    // 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
```

Here's what's happening:

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 need a new constant to define the job tag.

Open `ui\PodcastActivity.kt` and add the following line to the companion object:

```
private val TAG_EPISODE_UPDATE_JOB = "com.raywenderlich.podplay.episodes"
```

Add the following method:

```
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` causes 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 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 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 runs, and the notifications work. If the user taps the notification, it activates 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 active 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 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 results in the initial subscription missing an episode and causes 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.

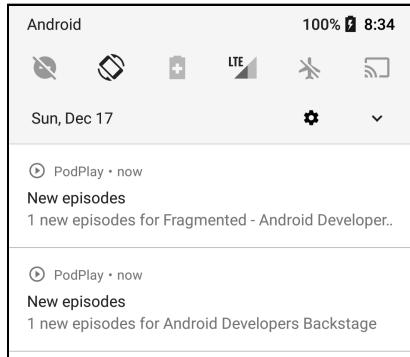
You may also want to reduce the execution window times on the job to have the job run without waiting an hour.

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 launches 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 completed the main podcast management part of the app. In the next chapter, you'll finally make PodPlay live up to its namesake by implementing the media playback interface.

Chapter 26: Podcast Playback

By Tom Blankenship

At this point, you've built a decent podcast management app, but 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 integrate it 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're following along with your own project, the starter project for this chapter includes an additional icon that you'll need to complete the section. Open your project then copy the following resources from the provided starter project into yours:

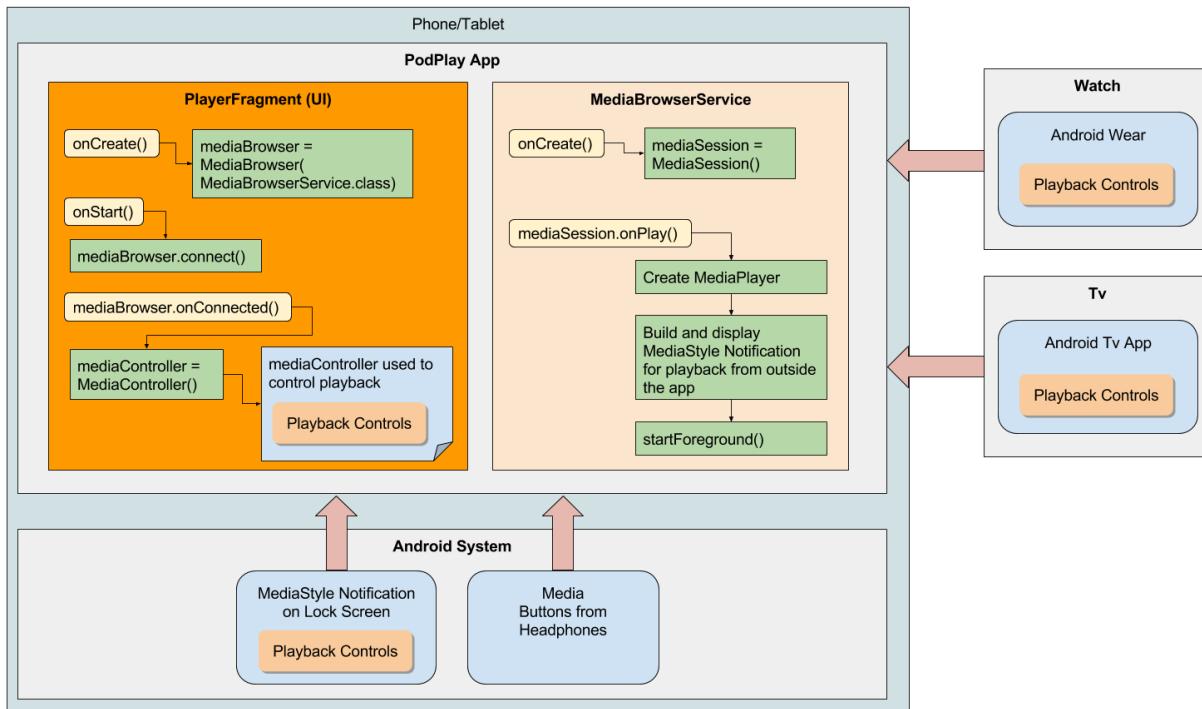
- src/main/res/drawable/ic_pause_white.png
- src/main/res/drawable/ic_play_arrow_white.png
- src/main/res/drawable/ic_episode_icon.png

Also, copy all of the files from the drawable folders, including folders with the **-hdpi**, **-mdpi**, **-xhdpi**, **-xxhdpi** and **-xxxhdpi** extensions.

If you don't have your own project, don't worry. Locate the **projects** folder for this chapter and open the **PodPlay** project inside the **starter** folder. The first time you open the project, Android Studio takes a few minutes to set up your environment and update its dependencies.

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 might 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 **MediaPlayer**. 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 of using a MediaSession is that systems other than your app 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, which you can use 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 of PodPlay. There are many ways a user may want to control audio from outside an app, 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 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 **MediaBrowser**. This class 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

MediaBrowserService is where all of the hard work of managing the podcast playback happens. You'll start with a basic implementation that's just enough to get a podcast playing and then expand the service later.

In the app's **build.gradle**, add the following dependency:

```
implementation "com.android.support:support-media-compat:  
$support_lib_version"
```

Click **make project**.

Inside **service**, create a new file and name it **PodplayMediaService.kt**. Replace its contents with the following:

```
import android.os.Bundle  
import android.support.v4.media.MediaBrowserCompat  
import android.support.v4.media.MediaBrowserServiceCompat  
  
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()**. You'll come back to these methods later in the chapter.

Similar to other services, **PodplayMediaService** needs an entry in the manifest.

Open **AndroidManifest.xml** and add the following below the main `<application>` section:

```
<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 MediaBrowserService is MediaSession. As PodPlay and other apps 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'll 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 need to define a `MediaSessionCompat.Callback` to handle media events.

Inside `service`, create a new file and name it `PodplayMediaCallback.kt`. 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 the skeleton code for the `Callback`; it doesn't do anything yet. Although you can handle other events, these are sufficient for PodPlay.

You'll come back to this later and fill in the details of each callback method. In the meantime, 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`, you'll 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 typically 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

Before adding the `MediaBrowser` object, you need to define the callback classes.

First, create the `MediaController.Callback` class. This class will receive messages when the playback state changes and is where you'd typically update your player UI to reflect the current state.

Open `PodcastDetailsFragment.kt` and add the following inner class:

```
inner class MediaControllerCallback: MediaControllerCompat.Callback() {
    override fun onMetadataChanged(metadata: MediaMetadataCompat?) {
        super.onMetadataChanged(metadata)
        println("metadata changed to ${metadata?.getString(MediaMetadataCompat.METADATA_KEY_MEDIA_URI)}")
    }

    override fun onPlaybackStateChanged(state: PlaybackStateCompat?) {
        super.onPlaybackStateChanged(state)
        println("state changed to $state")
    }
}
```

```
    }
```

You haven't implemented a playback UI yet, so the callback methods only print information for now.

Next, 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:

```
private lateinit var mediaBrowser: MediaBrowserCompat
private var mediaControllerCallback: MediaControllerCallback? = null
```

Add the following method:

```
private fun registerMediaController(token: MediaSessionCompat.Token) {
    // 1
    val fragmentActivity = activity as FragmentActivity
    // 2
    val mediaController = MediaControllerCompat(fragmentActivity, token)
    // 3
    MediaControllerCompat.setMediaController(fragmentActivity,
        mediaController)
    // 4
    mediaControllerCallback = MediaControllerCallback()
    mediaController.registerCallback(mediaControllerCallback!!)
}
```

Here's what's happening:

1. You assign a local `fragmentActivity` to `activity` since `activity` is a property that can change to `null` between calls.
2. 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` is part of the Android media session package, and it used to communicate with an active media session.

3. Assign the `MediaController` to the Activity so that you can retrieve it later with `getMediaController()`.

4. Create a new instance of `MediaControllerCallback` and set it as the callback object for the media controller.

Add the following inner class:

```
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` eventually calls `onConnected()` upon successful connection to the `MediaBrowserService`, or it calls `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.

Initialize 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() {  
    val fragmentActivity = activity as FragmentActivity  
    mediaBrowser = MediaBrowserCompat(fragmentActivity,  
        ComponentName(fragmentActivity, 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) {  
        val fragmentActivity = activity as FragmentActivity  
        if (MediaControllerCompat.getMediaController(fragmentActivity) ==  
null) {  
            registerMediaController(mediaBrowser.sessionToken)  
        }  
    } else {  
        mediaBrowser.connect()  
    }  
}
```

First, check to see if the media browser is already connected. This happens when a configuration change occurs, such as a screen rotation. If it's 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 complete.

Unregister the controller

The media controller callbacks should be **unregistered** when the Activity or Fragment is **stopped**.

Add the following method:

```
override fun onStop() {
    super.onStop()
    val fragmentActivity = activity as FragmentActivity
    if (MediaControllerCompat.getMediaController(fragmentActivity) != null)
    {
        mediaControllerCallback?.let {
            MediaControllerCompat.getMediaController(fragmentActivity)
                .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. Look at Logcat, and notice that things did not go as planned.

There are error messages from the `MediaBrowserService` and the `MediaBrowser`. Also, `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 calls 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)
```

Next, you need to 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. 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, and 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 get 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) : RecyclerAdapter<EpisodeListAdapter.ViewHolder>() {
```

This adds the `episodeListAdapterListener` argument to the constructor.

Update the `ViewHolder` definition to the following:

```
class ViewHolder(
    v: View, private
    val episodeListAdapterListener: EpisodeListAdapterListener) : RecyclerAdapter.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 `ViewHolder`:

```
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` now calls `onSelectedEpisode()` when the user taps an episode.

From here, you can make `PodcastDetailsFragment` implement the `episodeListAdapterListener` interface. First, you need to define a method to start the playback from an `EpisodeViewData` item.

Open `PodcastDetailsFragment.kt` and add the following method:

```
private fun startPlaying(
    episodeViewData: PodcastViewModel.EpisodeViewData) {
    val fragmentActivity = activity as FragmentActivity
    val controller =
        MediaControllerCompat.getMediaController(fragmentActivity)
    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, you need to implement the `episodeListAdapterListener` interface in `PodcastDetailsFragment`.

Update the `PodcastDetailsFragment` class definition as follows:

```
class PodcastDetailsFragment : Fragment(), EpisodeListAdapterListener {
```

Add the following method to implement the `onSelectedEpisode` logic:

```
override fun onSelectedEpisode(episodeViewData: EpisodeViewData) {
    // 1
    val fragmentActivity = activity as FragmentActivity
    // 2
    val controller =
        MediaControllerCompat.getMediaController(fragmentActivity)
    // 3
    if (controller.playbackState != null) {
        if (controller.playbackState.state ==
            PlaybackStateCompat.STATE_PLAYING) {
            // 4
            controller.transportControls.pause()
        } else {
            // 5
            startPlaying(episodeViewData)
        }
    } else {
        // 6
        startPlaying(episodeViewData)
    }
}
```

This is called when the user taps an episode. The current episode either plays or pauses depending on the current playback state.

Let's go over things in detail:

1. You assign a local `fragmentActivity` to `activity` since `activity` is a property that can change to `null` between calls.
2. You get the media controller that was previously assigned to the Activity.
3. If the playback state is not `null`, then you check the state.
4. If the playback state is “playing”, then you pause the episode using the transport controls.
5. If the playback state is “paused”, then you call `startPlaying()` to play the episode.
6. 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, it's time to 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 — you'll add more 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, you set the media session playback state to `STATE_PLAYING`.

Add the following line to the end of `onPause()`:

```
setState(PlaybackStateCompat.STATE_PAUSED)
```

When receiving the pause command, you 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. However, 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 to play back the media, including third-party media players.

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 need to 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` keeps track of the currently playing media item, and `newMedia` indicates if it's a new item. `mediaExtras` keeps track of the media information passed into `onPlayFromUri()`.

Next, 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`.

Audio Focus

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.

Android changed the way the audio focus is controlled starting with Android 8.0 (API Level 26). The new method is not compatible with older versions of Android, so you need to write slightly different code based on the version that the user is running.

First, create a method that requests audio focus.

Add the following property to the `PodplayMediaCallback` class:

```
private var focusRequest: AudioFocusRequest? = null
```

This is used in the code below to store an audio focus request when running Android 8.0 and above.

Next, add the following method:

```
private fun ensureAudioFocus(): Boolean {
    // 1
    val audioManager = this.context.getSystemService(
        Context.AUDIO_SERVICE) as AudioManager

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        // 2
        val focusRequest =
            AudioFocusRequest.Builder(AudioManager.AUDIOFOCUS_GAIN).run {
                setAudioAttributes(AudioAttributes.Builder().run {
                    setUsage(AudioAttributes.USAGE_MEDIA)
                    setContentType(AudioAttributes.CONTENT_TYPE_MUSIC)
                    build()
                })
                build()
            }
        // 3
        this.focusRequest = focusRequest
        // 4
        val result = audioManager.requestAudioFocus(focusRequest)
        // 5
        return result == AudioManager.AUDIOFOCUS_REQUEST_GRANTED
    } else {
        // 6
        val result = audioManager.requestAudioFocus(null,
            AudioManager.STREAM_MUSIC,
            AudioManager.AUDIOFOCUS_GAIN)
        // 7
        return result == AudioManager.AUDIOFOCUS_REQUEST_GRANTED
    }
}
```

Note: Android Studio will complain that the second `requestAudioFocus` call is deprecated. While it is deprecated in newer versions of Android, you must use it in the older version of Android since PodPlay supports version 4.4 and newer.

Here's the break down:

1. The `AudioManager` system service object is obtained.
2. If the version of Android is 8 (Android O) or newer, then an `AudioFocusRequest` object is generated using the `AudioFocusRequest` builder and stored in a local variable. The builder requires a single **focusGain** parameter, which is set to `AUDIOFOCUS_GAIN`. This tells Android that you want to gain audio focus and are about to start playing audio. A set of audio attributes are defined on the focus request to indicate that you are using media (`USAGE_MEDIA`) and the content type is music (`CONTENT_TYPE_MUSIC`). Other types of usage and content types can be set for different scenarios.
3. The class property `focusRequest` is assigned to the local `focusRequest` variable.
4. The call is made to `requestAudioFocus()` passing in the `focusRequest`.
5. `True` is returned if the focus request was granted; otherwise `False` is returned.
6. If the version of Android is less than 8, then a call is made to `requestAudioFocus()` passing in the following parameter types:

OnAudioFocusChangeListener: This is an optional callback allowing you to respond to audio focus changes. You won't handle focus changes in PodPlay, so the value is set to `null`.

streamType: This is the type of audio stream, and is similar to the content type used in Android 8 above.

durationHint: This is equivalent to the **focusGain** parameter in Android 8, and is set to `AUDIOFOCUS_GAIN`.

7. `true` is returned if the focus request was granted; otherwise `false` is returned.

Now you can use this new method to make sure you have audio focus before playback is started.

Update `onPlay()` after the call to `super` to surround the code with a call to `ensureAudioFocus()`, like so:

```
if (ensureAudioFocus()) {  
    mediaSession.isActive = true  
    setState()  
}
```

You 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  
  
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {  
        focusRequest?.let {  
            audioManager.abandonAudioFocusRequest(it)  
        }  
    } else {  
        audioManager.abandonAudioFocus(null)  
    }  
}
```

Note: Android Studio will complain that the `abandonAudioFocus` call is deprecated. While it is deprecated in newer versions of Android, you must use it to support older versions, including version 4.4, which PodPlay supports.

You'll call this method any time you pause or stop audio playback.

Just like the request to gain audio focus, this call changed starting with Android 8. If using Android 8 or newer, you call `abandonAudioFocusRequest()` and pass it the `focusRequest` that was obtained when gaining focus. If using a version before Android 8, you call `abandonAudioFocus()`.

Note: The above code is the minimum required to let Android know when you need audio focus so it can properly inform other apps. You're encouraged to review the full details of audio focus at <https://bit.ly/2ryV5dZ>. You can read about the different options for building audio requests, and how to implement an audio focus listener to handle focus changes in PodPlay.

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's called here instead.

Create a method to prepare the media for the `MediaPlayer`.

Add the following method to `PodplayMediaCallback`:

```
private fun prepareMedia() {
    if (newMedia) {
        newMedia = false
        mediaPlayer?.let { mediaPlayer ->
            mediaUri?.let { mediaUri ->
                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, you can update this to grab the position from the player.

Add the following after the `var position: Long = -1` line in `setState()`:

```
mediaPlayer?.let {
    position = it.currentPosition.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 you instruct it to play the media. You also set the media session state to `STATE_PLAYING`.

Add the following method to pause playback of the audio media.

```
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 you instruct it to pause the media. You also set the media session state to `STATE_PAUSED`.

Finally, you need to handle the case where playback is stopped.

Add the following method to `PodplayMediaCallback`:

```
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 need to call them at the appropriate times.

Add the following lines before the call to `onPlay()` in `onPlayFromUri()`:

```
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 the 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 pauses. Tap the same episode and playback begins 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'll also get shut down if you close PodPlay.

Foreground service

To keep the audio playing, you need to set `PodplayMediaService` as a foreground service. Any foreground⁹ service requires that it display a visible notification to the user. This is 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.

You'll assign two possible actions to the notification: a play action for when the media is **not** currently playing, and a pause action for when the media **is** currently playing. Whenever the media playback state changes, the notification gets 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: Choose `android.support.v4.app.NotificationCompat` for the `NotificationCompat` import.

You create pause and play actions and return them 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 need a method to determine if the MediaPlayer is currently playing media.

Add the following method to PodplayMediaService:

```
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:

```
private fun getNotificationIntent(): PendingIntent {
    val openActivityIntent = Intent(this, PodcastActivity::class.java)
    openActivityIntent.flags = 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:

```
private const val PLAYER_CHANNEL_ID = "podplay_player_channel"
```

Add the following method:

```
@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.

You're ready to 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()
}
```

Here are the details:

1. The method accepts a `MediaDescriptionCompat` object and a `bitmap`. These contain all of the details required to construct the notification.
2. The main notification intent is created. This is set as the content Intent on the notification and 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 tie this all together and create a method to display the notification.

First, you 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:

```
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
    GlobalScope.launch {
        // 5
        val iconUrl = URL(mediaDescription.iconUri.toString())
        // 6
        val bitmap = BitmapFactory.decodeStream(iconUrl.openStream())
        // 7
        val notification = createNotification(mediaDescription, bitmap)
        // 8
        ContextCompat.startForegroundService(
            this@PodplayMediaService,
            Intent(this@PodplayMediaService,
                PodplayMediaService::class.java))
        // 9
        startForeground(PodplayMediaService.NOTIFICATION_ID, notification)
    }
}
```

Note: Make sure to choose `java.net.URL` as the `URL` 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. A coroutine is launched in the background so the album artwork can be loaded from the network.
5. A `URL` object is created based on the album artwork icon internet location. This allows you to load the image over the network.

6. A stream is opened on the `iconUrl` and passed to the `BitmapFactory.decodeStream().decodeStream()` loads the image from the internet and it's stored in the `bitmap` object.
7. After the image is loaded, you create the notification using the description of the podcast episode and the album art bitmap.
8. `startForegroundService()` starts the service in foreground mode.
9. `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 `PodplayMediaCallback` 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 `PodplayMediaCallback` will call in response to key playback events.

First, you need a `listener` property that the media browser service can set.

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()
```

You're ready to 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)  
}
```

Here's what each one does:

- **onStateChanged()**: Displays the notification when the state changes between play and pause.
- **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 keeps running indefinitely.

- **onPausePlaying()**: Removes the service from the foreground but passes in `false`, so the notification is not removed.

Finally, you need to 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 is 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 `PodplayMediaService`:

```
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>
```

Since API 28, you must now add a permission to run your app as a foreground service. If you fail to do this, your app will crash. Add the following additional permission to the manifest:

```
<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />
```

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,
```

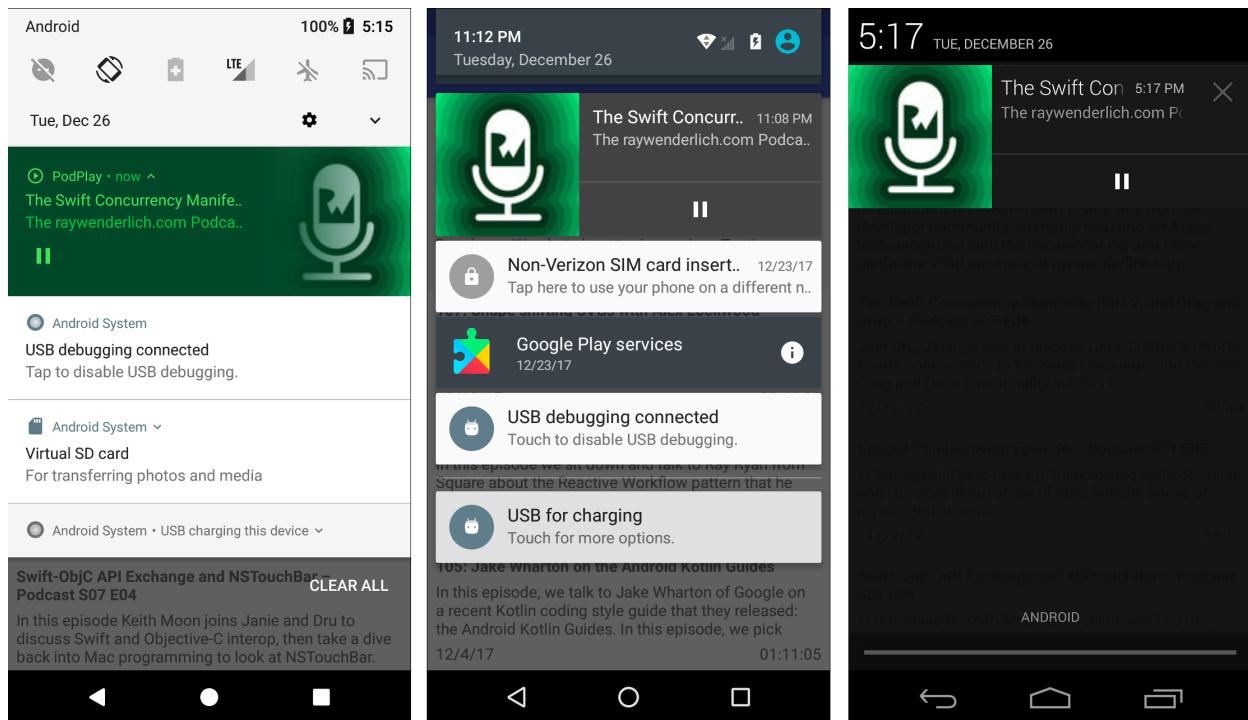
Open `SearchViewModel.kt` and replace `artworkUrl30` with `artworkUrl100` in `itunesPodcastToPodcastSummaryView()`:

```
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 displays in the status bar. Pull down the notification to reveal the expanded view. Tap on the pause button to pause the playback.



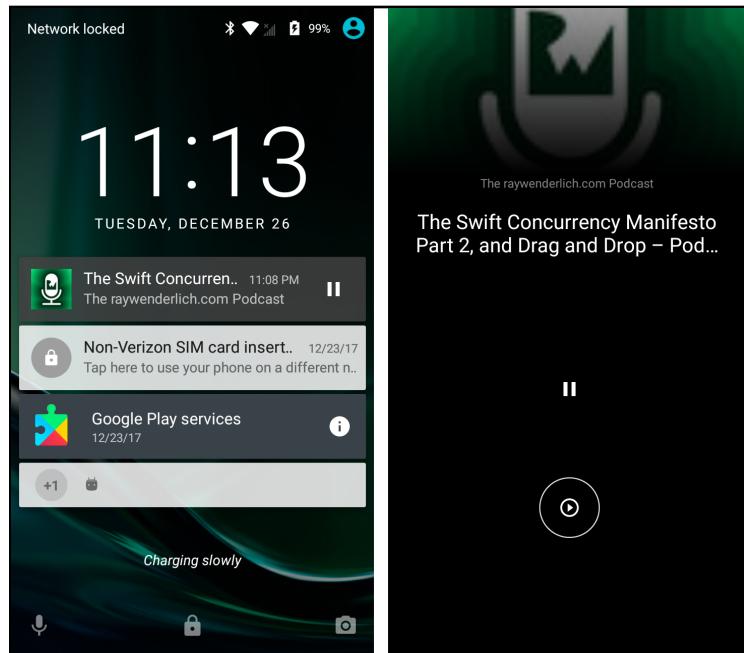
Depending on the version of Android you're running, the notification will display with a different style. Notice on Android Oreo that the notification takes on a tint color based on the album artwork.



From left to right, Android Oreo (8), Android Marshmallow (6), Android Lollipop (5)

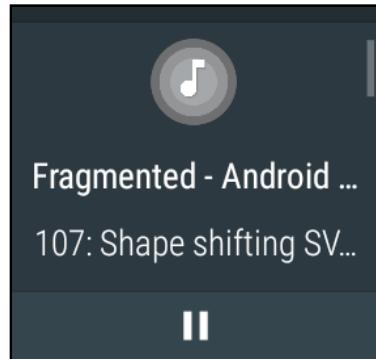
Press the play button to restart the audio playback, then exit PodPlay. The podcast keeps playing, and you can still control it from the notification view.

Turn off the phone and display the lock screen. The notification shows 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 correctly 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 PodPlay 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 this section, you'll finish up PodPlay by adding a full playback interface and support for videos.

If you're following along with your own project, the starter project for this chapter includes an additional icon that you'll need to complete the section. Open your project then copy the following resources from the provided starter project into yours. Be sure to copy the **.png** files from all of the **dpi** folders. This includes the following resources:

- res/drawable-?dpi/ic_forward_30_white.png
- res/drawable-?dpi/ic_replay_10_white.png
- res/drawable/ic_play_pause_toggle.xml

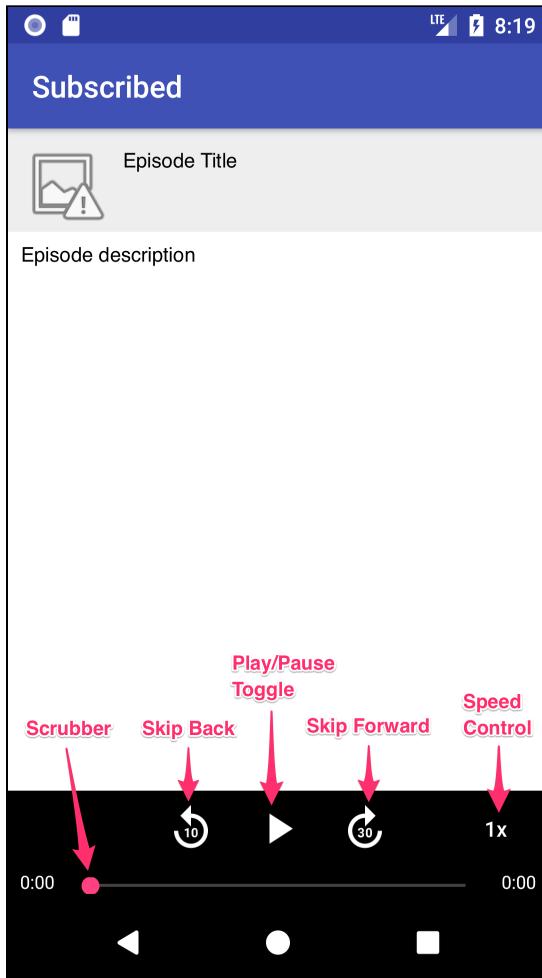
If you don't have your own project, don't worry. Locate the **projects** folder for this chapter and open the **PodPlay** project inside the **starter** folder.

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

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 provides an overview of the episode and playback controls. The design looks 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; and because episode descriptions can be long, the `TextView` is scrollable so that 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.

First up, creating the basic layout.

Episode player layout

Inside **res/layout**, create a new file and name it **fragment_episode_player.xml**.

Replace its 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"
    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="0dp"
        android:layout_height="0dp"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        android:visibility="invisible"/>

    <android.support.constraint.ConstraintLayout
        android:id="@+id/headerView"
        android:layout_width="0dp"
        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="0dp"
    android:layout_height="76dp"
    android:background="@android:color/background_dark"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    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.

Find the `headerView ConstraintLayout` section by looking for `android:id="@+id/headerView"`. Add the following before the `</android.support.constraint.ConstraintLayout>` line, after the `headerView` section:

```
<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.

It's time to take care of the primary player transport controls. Find the `playerControls` `ConstraintLayout` section by looking for `android:id="@+id/playerControls"`. Add the following before the `</android.support.constraint.ConstraintLayout>` line, after the `playerControls` section:

```
<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.

Still inside the `playerControls ConstraintLayout` section, add the following text directly after the code you just added:

```
<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

You're ready to build out the episode player Fragment. This Fragment will display the episode layout and handle all of the playback logic. You'll move the media related code from the `PodcastDetailsFragment` class into this new episode player fragment.

Inside ui, create a file named **EpisodePlayerFragment.kt** and Replace its contents with:

```
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: 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 needs 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 **OnPodcastDetailsListener** interface.

```
fun onShowEpisodePlayer(episodeViewData: EpisodeViewData)
```

Replace all of the 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.

In **PodcastActivity.kt**, add the following code to the companion object:

```
private const val TAG_PLAYER_FRAGMENT = "PlayerFragment"
```

This tag keeps track of the episode player Fragment in the support Fragment Manager.

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()** method 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` to keep track of the currently active episode. This makes 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
```

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 RecyclerView. It then hides the search menu item.

Now that all of the supporting methods are in place, you're ready to implement `onShowEpisodePlayer()`.

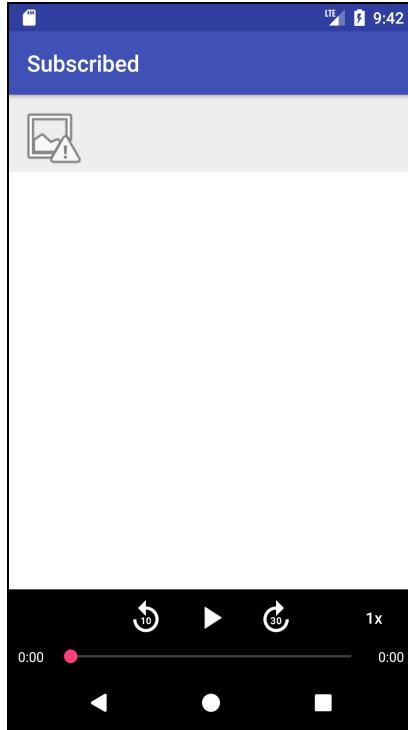
Add the following to `onShowEpisodePlayer()`:

```
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's blank since you haven't populated any of the views yet. Press the back button, and it navigates 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() {
    val fragmentActivity = activity as FragmentActivity
    podcastViewModel = ViewModelProviders.of(fragmentActivity)
        .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.

Next, 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
    val fragmentActivity = activity as FragmentActivity
    Glide.with(fragmentActivity)
        .load(podcastViewModel.activePodcastViewData?.imageUrl)
        .into(episodeImageView)
}
```

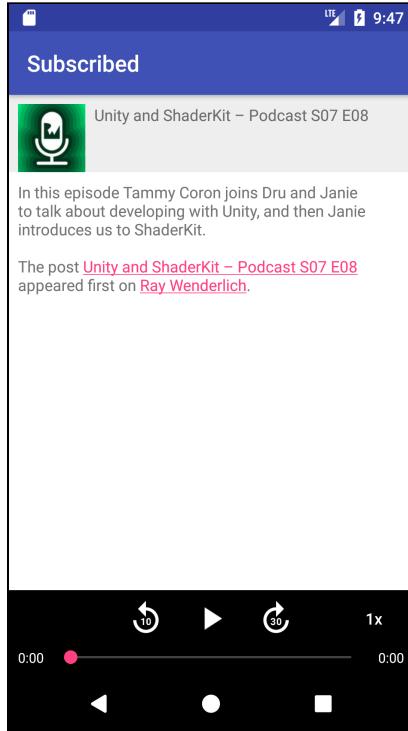
Let's take this one item at a time:

1. Set the episode title text view to the episode title.
2. Just like the podcast description that's shown on the podcast details View, the episode description can have HTML formatting that causes 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. It also sets `movementMethod` to `ScrollingMovementMethod` to allow the description to scroll.
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.

During the previous chapter, you added some media playback code to the **PodcastDetailsFragment** class. This was sufficient to test that podcast playback was working, but now **EpisodePlayerFragment** will handle all playback. You'll start by moving some media playback code from **PodcastDetailsFragment** to the new **EpisodePlayerFragment**.

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
```

Note: If Android Studio changes `MediaControllerCallback` to `PodcastDetailsFragment.MediaControllerCallback`, change it back to `MediaControllerCallback`; it will show a compile error until you get to step 5.

2. Move `startPlaying()` from **PodcastDetailsFragment.kt** to **EpisodePlayerFragment.kt**.
3. Move the code below `super.onStart()` from `onStart()` in **PodcastDetailsFragment.kt** to the bottom of `onStart()` in **EpisodePlayerFragment.kt**.
4. Move the code below `super.onStop()` from `onStop()` in **PodcastDetailsFragment.kt** to the bottom of `onStop()` in **EpisodePlayerFragment.kt**.
5. Move the `MediaBrowserCallBacks` and `MediaControllerCallback` inner classes from **PodcastDetailsFragment.kt** to **EpisodePlayerFragment.kt**.
6. Move `initMediaBrowser()` from **PodcastDetailsFragment.kt** to **EpisodePlayerFragment.kt**.
7. Move `registerMediaController()` from **PodcastDetailsFragment.kt** to **EpisodePlayerFragment.kt**.
8. Move the call to `initMediaBrowser()` from `onCreate()` in **PodcastDetailsFragment.kt** to the bottom of `onCreate()` in **EpisodePlayerFragment.kt**.

Note: If Android Studio has again changed `MediaControllerCallback` to `PodcastDetailsFragment.MediaControllerCallback` anywhere in **EpisodePlayerFragment.kt**, change them back to `MediaControllerCallback`

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 fragmentActivity = activity as FragmentActivity  
    val controller =  
        MediaControllerCompat.getMediaController(fragmentActivity)  
    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 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:

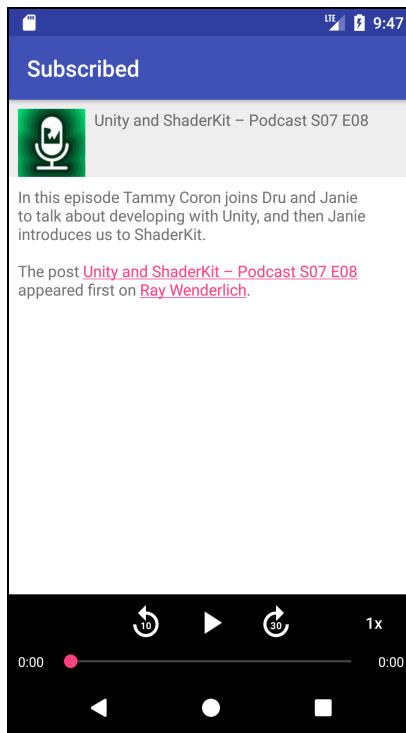
```
val state = state ?: return  
handleStateChange(state.getState())
```

Finally, add the call to `setupControls()` before the call to `updateControls()` in `onActivityCreated()`:

```
setupControls()
```

Build and run the app.

Load a podcast episode and test out the new play button functionality.



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 the media browser service that you want to change the speed? The answer is by using a custom command.

You need to 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()`, 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 { mediaPlayer ->
```

```
// 5
try {
    mediaPlayer.playbackParams =
mediaPlayer.playbackParams.setSpeed(speed)
}
catch (e: Exception) {
// 6
    mediaPlayer.reset()
    mediaUri?.let { mediaUri ->
        mediaPlayer.setDataSource(context, mediaUri)
    }
    mediaPlayer.prepare()
// 7
    mediaPlayer.playbackParams =
mediaPlayer.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 playback 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 sets 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:

```
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:

```
override fun onCommand(command: String?, extras: Bundle?,
    cb: ResultReceiver?) {
    super.onCommand(command, extras, cb)
    when (command) {
        CMD_CHANGESPEED -> extras?.let { changeSpeed(it) }
    }
}
```

Note: 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 needs to send the custom command when the user changes the speed.

First, you 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 fragmentActivity = activity as FragmentActivity
    val controller =
        MediaControllerCompat.getMediaController(fragmentActivity)
    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.

Seeking

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 `PodplayMediaCallback`.

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 fragmentActivity = activity as FragmentActivity
    val controller =
        MediaControllerCompat.getMediaController(fragmentActivity)
    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 invokes `onSeekTo()` you defined in the media browser service.

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 need to take the current playback position, add a plus or minus offset to get a new position, and then set the new position.

Start by adding 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. You may have noticed that the scrubber at the bottom does not move to reflect the current playback position. You'll fix that now and implement the 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 only 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 stores 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: 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.

You need to call this new method when the metadata is 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.max = 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 places 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 `EpisodePlayerFragment`:

```
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 fragmentActivity = activity as FragmentActivity
            val controller =
                MediaControllerCompat.getMediaController(fragmentActivity)
            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 invalid, 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 `EpisodePlayerFragment`:

```
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
    if (timeRemaining < 0) {
```

```
        return;
    }
// 4
progressAnimator = ValueAnimator.ofInt(
    progress, episodeDuration.toInt())
progressAnimator?.let { animator ->
    // 5
    animator.duration = timeRemaining.toLong()
    // 6
    animator.interpolator = LinearInterpolator()
    // 7
    animator.addUpdateListener {
        if (draggingScrubber) {
            // 8
            animator.cancel()
        } else {
            // 9
            seekBar.progress = animator.animatedValue as Int
        }
    }
    // 10
    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. If `timeRemaining` is negative then the function is abandoned. This will prevent any unintended side effects when switching between podcasts.
4. Create a new `ValueAnimator` with the starting and ending value of the animation and assign it to the `progressAnimator` property.
5. The animation duration is set to the time remaining. This stops the animation when it reaches the end of the episode.
6. 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.
7. Set an update listener on the animator. This listener is called by the animator on each step of the animation.
8. 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.

9. If the user is not dragging the scrubber, then update the scrubber indicator to the current value from the animator.

10. 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():9`

```
    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.

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.

Now update the call to `handleStateChange()` in the `onPlaybackStateChanged()` method of the `MediaControllerCallback` class to the following:

```
    handleStateChange(state.state, state.position, state.playbackSpeed)
```

This passes in the additional parameters added to `handleStateChange()`.

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 fragmentActivity = activity as FragmentActivity
    val controller =
        MediaControllerCompat.getMediaController(fragmentActivity)
    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()
```

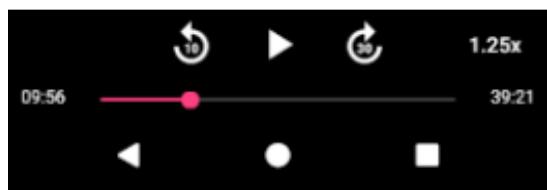
Add the call to `onStart()` before the `else` statement.

```
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 you'll control it 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
```

The updated **EpisodeViewData** class should match the following:

```
data class EpisodeViewData (
    var guid: String? = "",
    var title: String? = "",
    var description: String? = "",
    var mediaUrl: String? = "",
    var releaseDate: Date? = null,
    var duration: String? = "",
    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, **isVideo** on the **EpisodeViewData** is set to **true**.

Now you need to update **EpisodePlayerFragment.kt** to handle video playback.

To start video playback, you need to perform a few tasks:

1. Create a media session and a 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 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 a 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 that media buttons are ignored if the app is not in the foreground.

Media player

You also need a `MediaPlayer` object just like you did with the `MediaBrowserService`.

Add the following property to `EpisodePlayerFragment`:

```
private var mediaPlayer: MediaPlayer? = null
```

You 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 loads 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 fragmentActivity = activity as FragmentActivity  
                val episodeMediaCallback = PodplayMediaCallback(fragmentActivity,  
                    mediaSession!!, it)  
                mediaSession!!.setCallback(episodeMediaCallback)  
                // 6  
                setSurfaceSize()  
                // 7  
                if (playOnPrepare) {  
                    togglePlayPause()  
                }  
            }  
            // 8  
            it.prepareAsync()  
        }  
    } else {  
        // 9  
        setSurfaceSize()  
    }  
}
```

Here'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 happens if there's 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.

Next, you'll add some conditional code that skips the `MediaBrowser` creation and usage if it's 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()`, except for `super.onStart()`, with a check for `isVideo`:

```
if (!isVideo) {  
    if (mediaBrowser.isConnected) {  
        val fragmentActivity = activity as FragmentActivity  
        if (MediaControllerCompat.getMediaController(fragmentActivity) ==  
            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 allows 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 player controls background color to a 50% transparency level.

Call `setupVideoUI()` when the video is playing by adding the following as the first line inside the "if (`isPlaying`) {" section of `handleStateChange()`:

```
if (isVideo) {  
    setupVideoUI()  
}
```

You need to manually stop the playback when the fragment is exited, so add the following to the end of `onStop()`:

```
if (!fragmentActivity.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 `mediaPlayer` is not `null`, then the controls are updated from the media controller state.

There's one minor change required in **PodplayMediaCallback.kt** to make sure the media player is not prepared a second time. You need this 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 is used to indicate if the media player needs to be prepared.

In `initializeMediaPlayer()`, add the following line to the end of the method:

```
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.

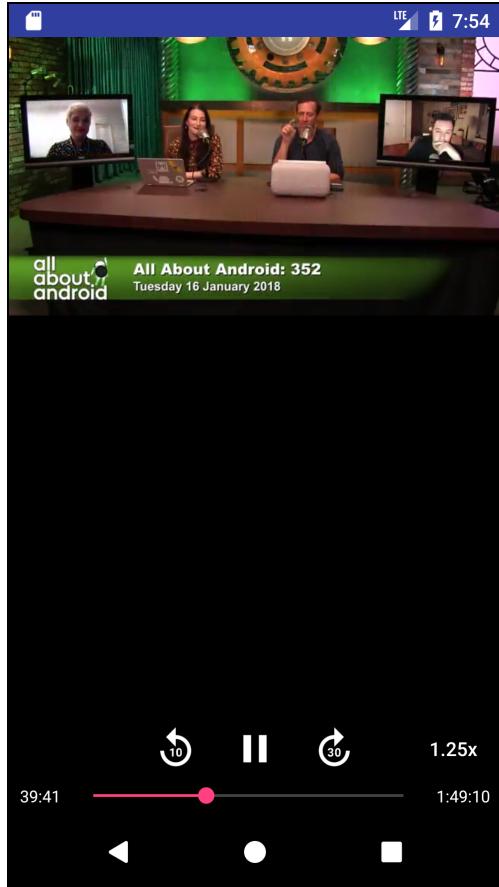
That's all the changes required in the shared `PodplayMediaCallback` object to support video playback. All of the existing controls, including skip and speed, will work without any changes.

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 shows 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 fully functional podcast player worthy of praise and bragging rights! Pat yourself on 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, you'll discover 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, all Android device would run a single version of Android and app development would be easy. As it turns out, the world isn't perfect.

In May 2017 (the last announcement we could find) Google announced there were two *billion* monthly active Android devices around the world, all running various versions of Android. That's an impressive statistic for Google, and most certainly there are many more nearly two years later, but it's also terrifying for developers who wants their app to work on as many devices as possible.

This chapter, however, might help to put you at ease. Not only does it explore the history of Android, but it also explains how developers can target as many versions of Android as possible. Within its pages, you'll learn:

- 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 this book uses the Support Libraries as a way to be backward compatible.

Android: An open operating system

To understand where the fragmentation problem originates, it's important to understand how Android came to be the most popular operating system on the planet.

Google originally acquired Android by buying a company named **Android Inc** in 2005. Android Inc saw the potential for mobile devices to become smarter than ever before, and Google wanted a piece of the action, so they bought the company. 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 phone manufacturers such as Samsung, LG and HTC. What Google offered to phone manufacturers was a stable operating system, and 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 phone manufacturers, it was a way of keeping up with the competition. The approach toward 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 toward 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, devices eventually needed 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 the 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 with particular devices.

If you look at the leading Android devices of today, you'll 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'll 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, led to tech journalists declaring 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 rearchitect the Android OS via a project named **Project Treble** (<https://source.android.com/devices/architecture/treble>), aiming to abstract the core of Android and provide interfaces for manufacturers to use in their own Android implementations.

The idea is a stable Android core that allows manufacturers to customize Android in a quicker, cheaper and safer way, allowing users to receive Android updates much quicker.

The Pixel devices from Google run unmodified versions of Android, often called the “vanilla” version. This means these devices can be 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's 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 backward 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. In fact, you've been using the Support Libraries all this time as you've worked through this book.

Look through the **ListMaker** app you created in Section II, and you'll see that the Support Libraries are used throughout the app. You can see the first sign in **build.gradle** 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-jdk7:$kotlin_version"
    implementation 'com.android.support:appcompat-v7:28.0.0'
    implementation 'com.android.support.constraint:constraint-layout:1.1.3'
    implementation 'com.android.support:design:28.0.0'
    implementation 'com.android.support:support-v4:28.0.0'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'com.android.support.test:runner:1.0.2'
    androidTestImplementation
    'com.android.support.test.espresso:espresso-core:3.0.2'
    implementation 'com.android.support:recyclerview-v7:28.0.0'
    implementation 'com.android.support:cardview-v7:28.0.0'
}
```

The dependencies prefixed with `com.android.support:` are all part of the Support Library collection, created explicitly for backward compatibility of newer features to older versions of Android.

It's thanks to the Support Libraries that **Constraint Layouts** are compatible back to Android Gingerbread. Gingerbread was released in December 2010, yet Constraint Layouts were introduced in February 2017. That's an incredible amount of support for old software.

It may be 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 macOS, hover the mouse cursor over the **AppCompatActivity** subclass at the top and left-click it:

```
package android.support.v7.app;

import ...

public class AppCompatActivity extends FragmentActivity implements AppCompatActivityCallback, S
    private AppCompatActivityDelegate mDelegate;
    private int mThemeId = 0;
    private Resources mResources;

    public AppCompatActivity() {
}

protected void onCreate(@Nullable Bundle savedInstanceState) {
    AppCompatActivityDelegate delegate = this.getDelegate();
    delegate.installViewFactory();
    delegate.onCreate(savedInstanceState);
    if (delegate.applyDayNight() && this.mThemeId != 0) {
        if (VERSION.SDK_INT >= 23) {
            this.onApplyThemeResource(this.getTheme(), this.mThemeId, first: false);
        } else {
            this.setTheme(this.mThemeId);
        }
    }
    super.onCreate(savedInstanceState);
}
```

Note: For Windows / Linux users, hold Ctrl and left-click.

Android Studio jumps to **AppCompatActivity.java**. **AppCompatActivity** is part of the `com.android.support:appcompat-v7:28.0.0` library. You can tell by the 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 backward compatible feature that **AppCompatActivity** offers is the ease of supplying a **ToolBar** to the top of the Activity.

Next, open **MainActivity.kt**. Then hold the **Command** button, hover your mouse over `setSupportActionBar(toolbar)` in `onCreate()` and left-click to jump to the implementation.

```
public void setSupportActionBar(@Nullable Toolbar toolbar) {  
    this.getDelegate().setSupportActionBar(toolbar);  
}
```

Android Studio takes you back into **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 handy. 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.

It's time to 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.

```
public Fragment() {  
}  
  
public static Fragment instantiate(Context context, String fname) {  
    return instantiate(context, fname, (Bundle)null);  
}
```

Android Studio opens **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;
```

That's right, 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. However, thanks to the Support Library implementation, Fragments can be used back to Android Donut, which was released *two years before* Fragments were introduced.

Open **ListSelectionFragment.kt** and **Command-left-click** over the **RecyclerView** defined at the top of the class.

```
package android.support.v7.widget;  
import ...  
public class RecyclerView extends ViewGroup implements ScrollView, NestedScrollingChild2 {
```

Android Studio shows 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;
```

This is another Support Library component that you've been using.

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 into the Support Library as they had recognized how integral the libraries had become. This decision meant **Recyclers** were not released in a particular version of Android. As part of the Support Libraries, they became a crucial element of the UI that are backward-compatible to Android Eclair, which was 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 their apps can reach as many users and devices as possible. While not every feature can be backported, the most important ones that provide consistency for the user experience are there for you to use.

That said, 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; without them, development across multiple Android 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 features the Support Libraries provide at <https://developer.android.com/topic/libraries/support-library/features.html>.
- Finally, you can find a list of 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 your app; some will uninstall it a few minutes later; and a small few will genuinely find your app useful and use it regularly, perhaps even leaving reviews. This last group contains the users to which you owe your attention and commitment.

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 for 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 is 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 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 deploy your app. 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 few dashboards at (<https://developer.android.com/about/dashboards/index.html>) that detail key metrics:

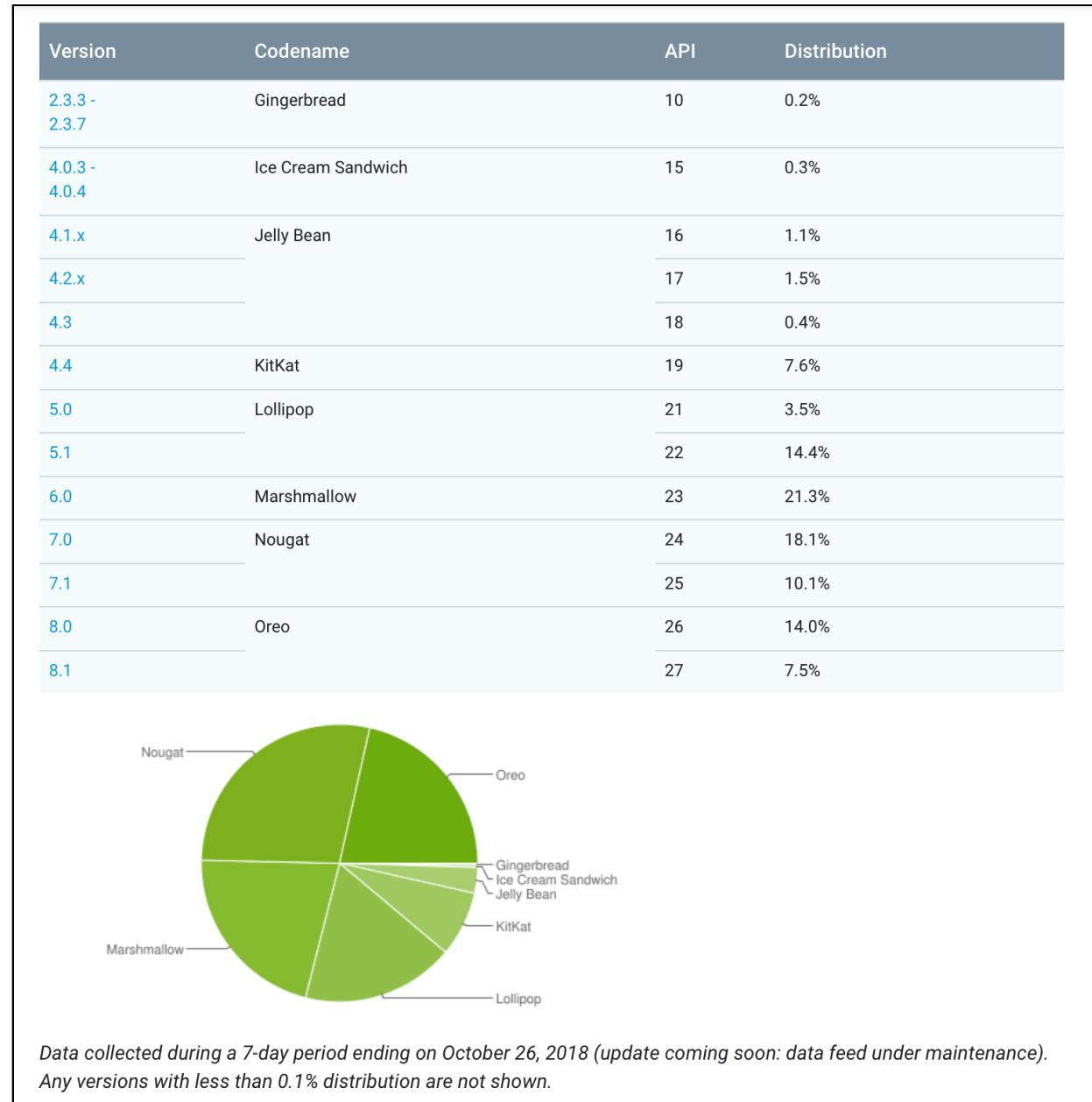
- Android versions.
- Screen size and density.
- OpenGL versions.

Google generates this information from devices that visited the Google Play Store within the last seven days, so you can rely on the dashboards to provide an accurate portrait of Android within the Google ecosystem.

Android versions

Choosing the right platform to target can lead to building simpler apps if you’re not preoccupied trying to backport features or trying to fallback gracefully for Android versions that don’t support your app’s features.

Looking at this dashboard can help you decide to leave versions in the dust. You’ll explore this idea later on in this section.



Screen size and density

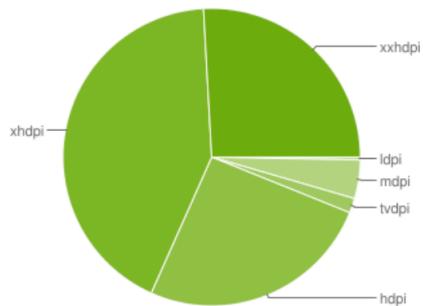
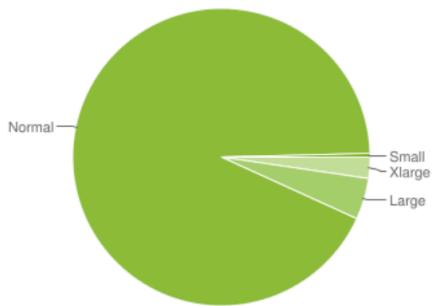
Keeping an eye on the most common screen sizes and densities can help you decide to focus on the sizes users are using most. Keeping up with the latest trends here can help you shed unneeded assets and keep your APK size slim.

Screen sizes and densities

This section provides data about the relative number of devices that have a particular screen configuration, defined by a combination of screen size and density. To simplify the way that you design your user interfaces for different screen configurations, Android divides the range of actual screen sizes and densities into several buckets as expressed by the table below.

For information about how you can support multiple screen configurations in your application, read [Supporting Multiple Screens](#).

	ldpi	mdpi	tvdpi	hdpi	xhdpi	xxhdpi	Total
Small	0.3%					0.1%	0.4%
Normal		0.7%	0.3%	24.7%	41.9%	25.2%	92.8%
Large		2.0%	1.3%	0.4%	0.3%	0.5%	4.5%
Xlarge		1.5%		0.5%	0.3%		2.3%
Total	0.3%	4.2%	1.6%	25.6%	42.5%	25.8%	



Data collected during a 7-day period ending on October 26, 2018 (update coming soon: data feed under maintenance). Any screen configurations with less than 0.1% distribution are not shown.

OpenGL versions

OpenGL is a library used in games or graphically intensive apps to render 2D or 3D graphics. It's incredibly popular due to its portability across platforms.

Depending on the Android device, it will contain a specific version of the OpenGL library. The more recent versions of OpenGL contain more efficient and newer ways of rendering graphics, although OpenGL is also backward compatible.

It's useful to know about the distribution of OpenGL across devices if you're a developer trying to port across a PC game, for instance.

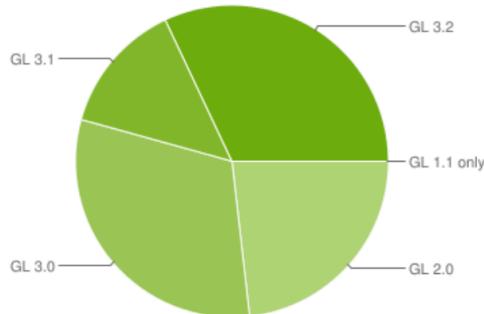
It gives you an indication as to whether your OpenGL code will be compatible or not.

OpenGL ES version

This section provides data about the relative number of devices that support a particular version of OpenGL ES. Note that support for one particular version of OpenGL ES also implies support for any lower version (for example, support for version 2.0 also implies support for 1.1).

To declare which version of OpenGL ES your application requires, you should use the `android:glEsVersion` attribute of the `<uses-feature>` element. You can also use the `<supports-gl-texture>` element to declare the GL compression formats that your application uses.

OpenGL ES version	Distribution
2.0	21.1%
3.0	29.8%
3.1	13.6%
3.2	35.5%



Data collected during a 7-day period ending on October 26, 2018 (update coming soon: data feed under maintenance).

How you use the data that Google provides to focus your development efforts on depends entirely on your personal goals for your app.

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 sense 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 get notified of upcoming Android updates is to 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 fix 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 a user than updating their device 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/2018/07/final-preview-update-official-android-p.html>), or 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 provide Google engineers with feedback on issues 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 be the end of the world if you don't. The engineers at Google have done excellent work in making various libraries on Android backward-compatible, which 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 in the future lets you adjust your development ahead of time — or even make the call to not update your app.

Working with older versions of Android

Although there is a lot of support in Android for backward 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 for your app to run. If the API you’re targeting doesn’t exist on an 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 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 to 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 vast numbers of 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 reacts on older devices, and whether the app still functions as you intended on older versions of Android.

The backport approach

The third option is 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 signal 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 & Publishing**](#)



Chapter 30: Preparing for Release

By Fuad Kamal

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 focuses 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.

You're ready to 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 the 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, **PodPlay** 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. It also 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. 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

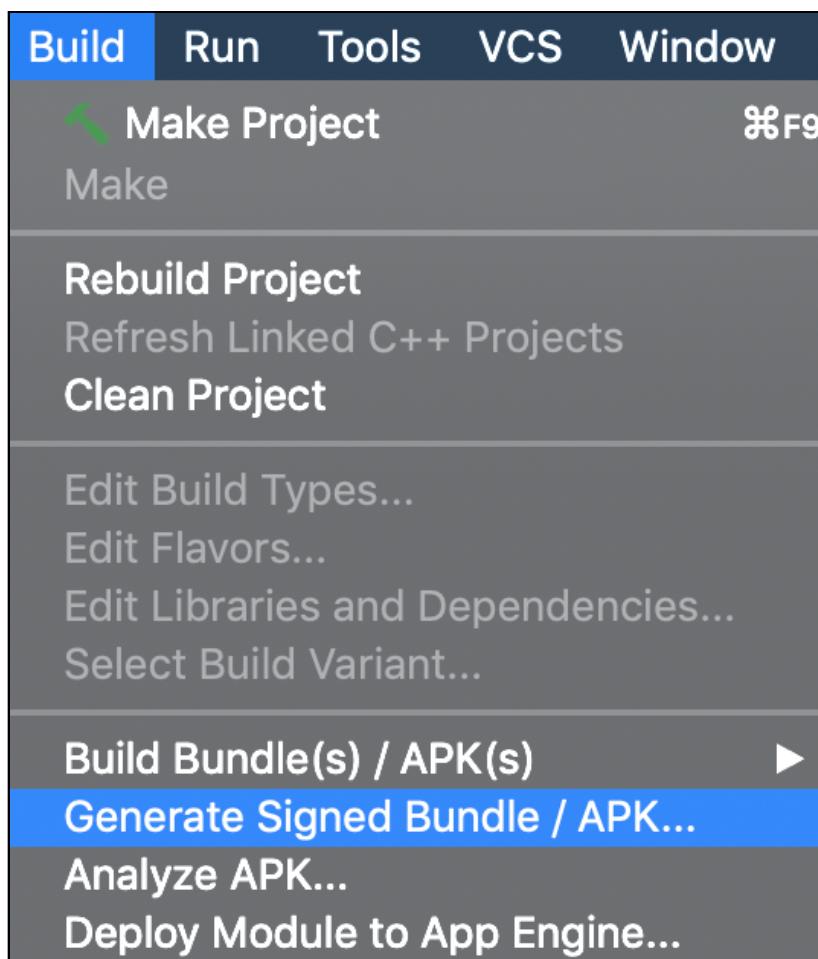
Your first step in building a release version is to generate a signing key, which you'll use to sign the app. This key is stored in a keystore file, and any future versions of the same app must be signed with the same key.

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!

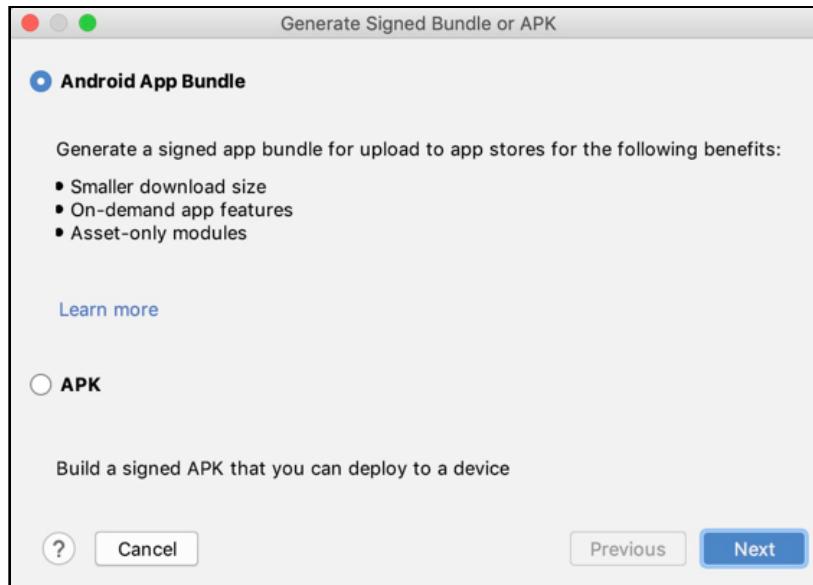
Note: Google has a Google Play App Signing feature. This service lets Google manage your signing key, giving you some options if you lose your key or it gets compromised. When using this method, you'll sign the app with an **Upload Key**, and then Google will resign the app with your actual app signing key. This is 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:

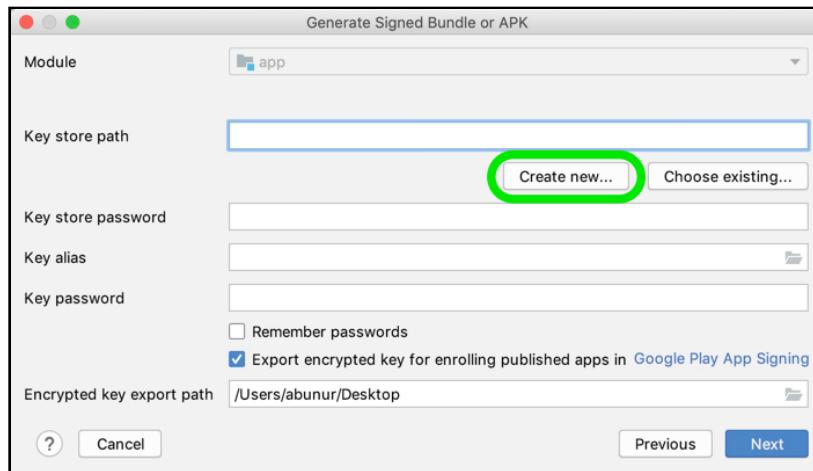
1. Click **Build > Generate signed Bundle / APK...** from the menu.



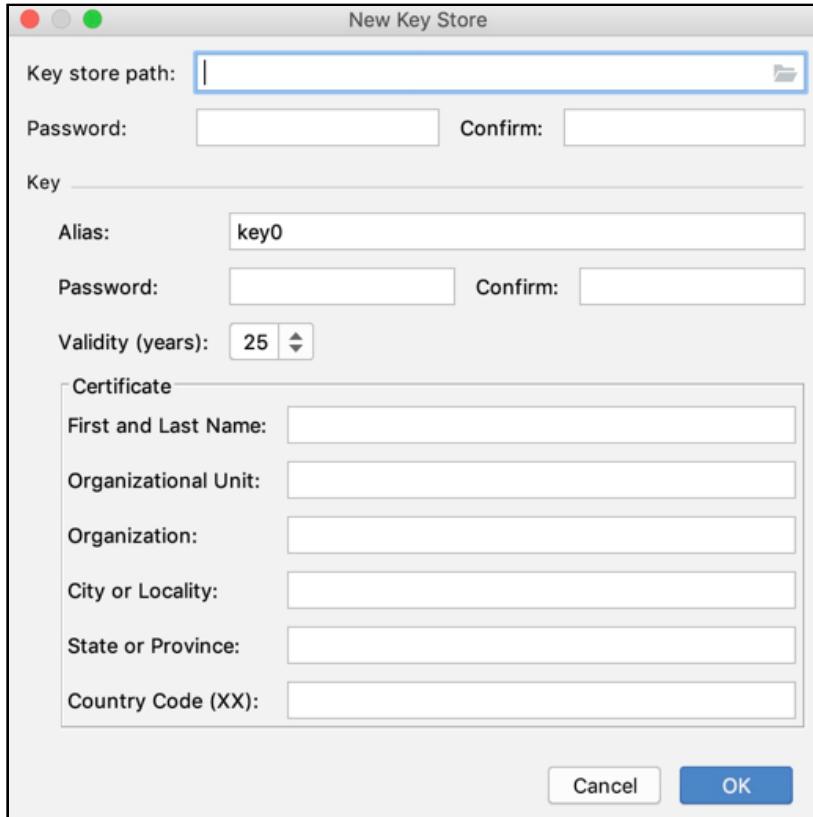
2. An Android App Bundle is a format that includes all of your app's compiled code and resources, but defers APK generation and signing to Google Play. Google Play then uses your app bundle to generate and serve optimized APKs for each user's device configuration, so they download only the code and resources they need to run your app. Select **Android App Bundle**, 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**:

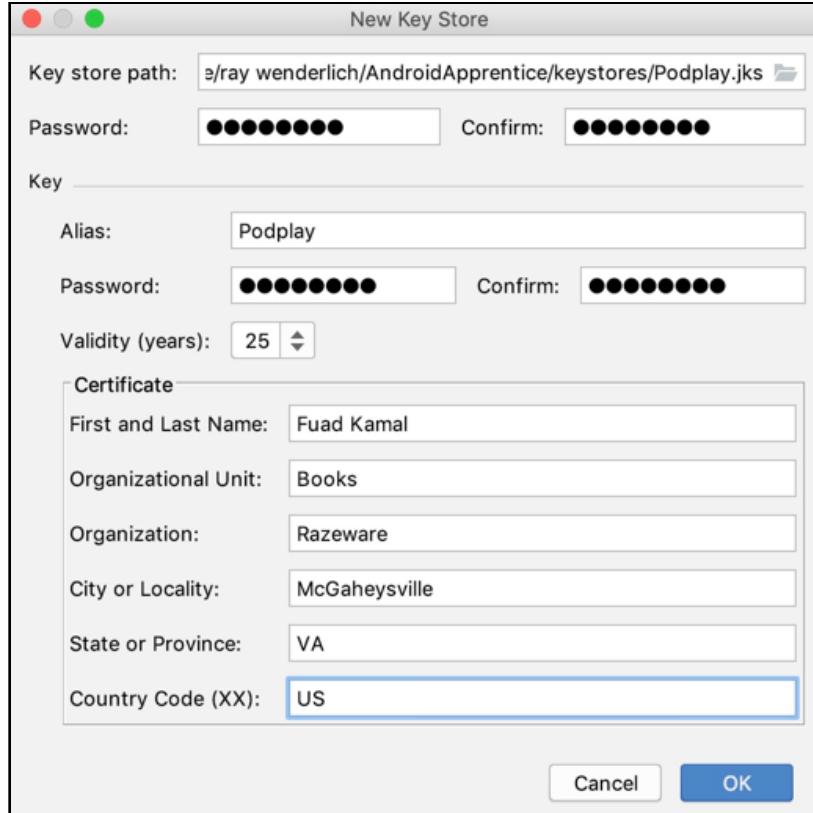
Alias: Enter a name for the key. Usually the name of your app.

Password: Enter a password to protect the keystore file.

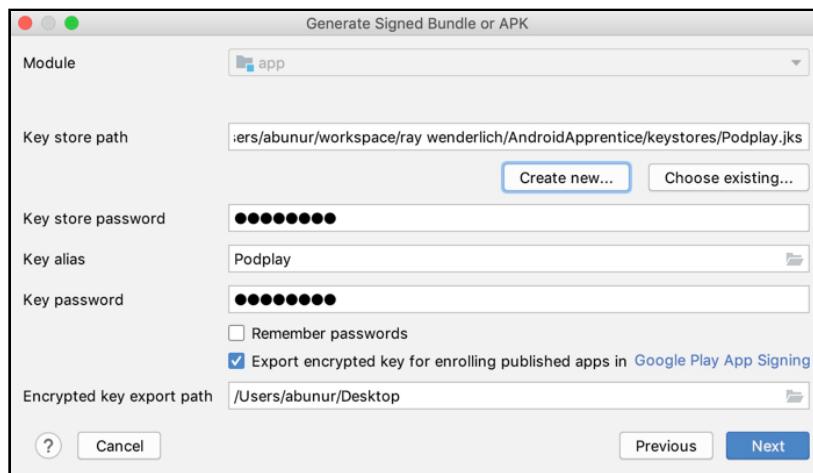
Confirm: Repeat your password.

Validity (years): Leave this at 25 years. The key expires after this time.

Certificate: Enter your personal information in these fields. The user won't see your data, but it's part of the signing certificate in the APK file.



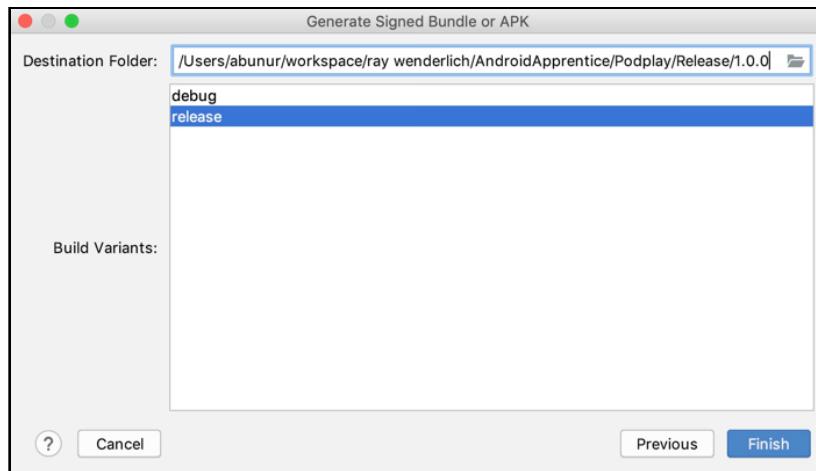
8. Click **OK**, and the original dialog, with the values already populated, appears.



9. If you don't want to enter passwords each time you build a release version, check **Remember passwords**.
10. To take advantage of Google Play App Signing, check **Export encrypted key for enrolling published apps in Google Play App Signing** and choose a destination folder to save the encrypted key. This key is encrypted for transfer to Google Play.
11. Click **Next**.

12. Fill in the **Destination Folder**. Normally, this a folder outside of your main project folder.

Under **Build Variants**, ensure **release** is selected.



13. Click **Finish**.

Android Studio builds and signs the release Android App Bundle file and places it in the destination folder. A popup displays in the bottom right corner of Android Studio when the build is complete.

The final output file is given the name **app.abb**.

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, including distributing malicious apps under your identity.

Check file size

Check the size of the app bundle file. If it's over 500MB, you won't be able to publish it as-is to the Play Store. You can get around this limitation by using dynamic feature modules. This is not an issue for most applications, but if you find yourself with a large bundle file, you can find details about using app bundles and dynamic feature modules files here: <https://developer.android.com/guide/app-bundle/>

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.

Test your Android App Bundle using **bundletool** to generate APKs from your app bundle and deploy them to a connected device. You can find details about downloading and using bundletool here: <https://developer.android.com/studio/command-line/bundletool>

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:

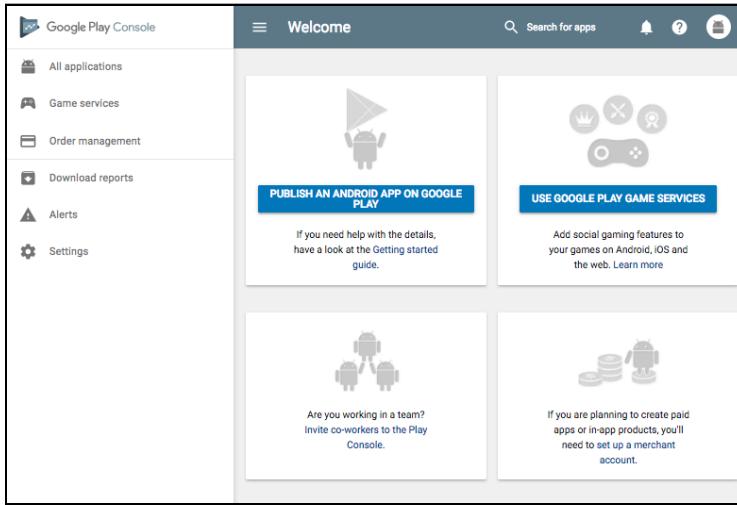
<https://play.google.com/apps/publish/>

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's shown in the Play Store below the name of your app.

The main console

Once you're finished with signup, you'll get to the main console.



In the menu on the left, you have several options:

- **All Applications** is where you add new applications or manage existing ones.
- **Game Services** 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 including giving refunds.
- **Download Reports** provides a variety of reports, including crashes, reviews, statistics, user acquisition and financial records.
- **Alerts** is where you can see any alerts generated by the Play Store for your apps.

And finally **Settings** 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 email lists:** 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 **PUBLISH AN ANDROID APP ON GOOGLE PLAY** on the main console screen.

Note: At this point, you're just preparing the store listing and creating a draft version of the app; nothing gets published until you use the Publish step.

First, fill in the title of your app and click **CREATE**.

Create application

Default language *
English (United States) - en-US

Title *
PodPlay
7/50

CANCEL CREATE

This creates the app and presents you with several pages of information related to it.

The first page you'll see is the Store listing. Here's a partial view of this page:

All applications

- App releases
- Android Instant Apps
- Artifact library
- Device catalog
- App signing
- Store listing**
- Content rating
- Pricing & distribution
- In-app products
- Translation service
- Services & APIs
- Optimization tips

Updated location for the Submit update and Timed publishing buttons, and your app's status

The following changes will come into effect.

- Status of your app will be visible in the app search field on the top-right of your app's Play Console pages.
- Save draft will move to a static location at the bottom of the Store listing and Pricing & distribution pages, so that you save your draft app at any point irrespective of where you are on these pages.

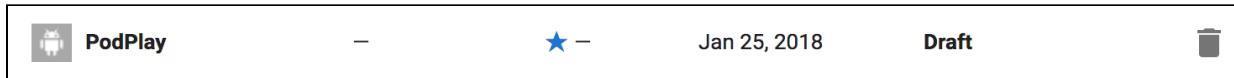
Product details

ENGLISH (UNITED STATES) - en-US Manage translations ▾

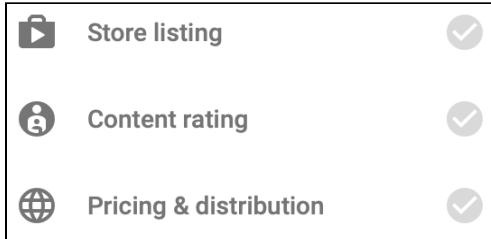
Title *
English (United States) - en-US PodPlay
7/50

Short description *
English (United States) - en-US
0/80

Go back to the home console screen, and you'll see the new app you just added, with a status of **Draft**.



Click on the app name to go into the Dashboard view for the app. Look at the left side of the screen. The items with grayed out checkmarks 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 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.

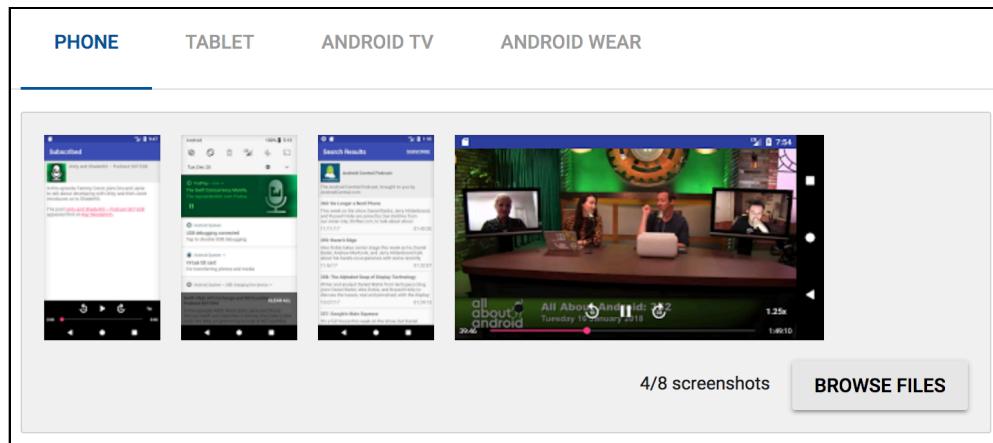
Store listing

Click **Store listing** and fill out the following required items:

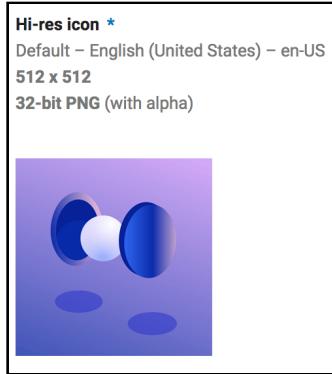
- **Short description:** Up to 80 characters. Mention the most important feature of your app and explain why a user would want to install it. Think of this as the app promotional text.
- **Full Description:** Up to 4000 characters. Provide the full benefits and features of your app. Use keywords in the 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.

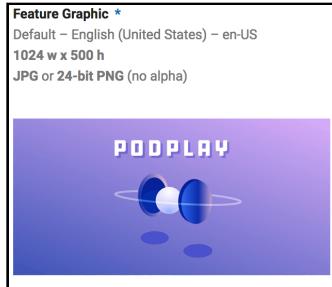
- **Screenshots:** Drag your screenshots to the appropriate device tabs.



- **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 PodPlay.
- **Content Rating:** You may see a message, “You need to fill a rating questionnaire and apply a content rating”. This appears if you haven’t yet uploaded an APK and filled out the content rating questionnaire. You’ll do this in the next chapter.
- **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**.

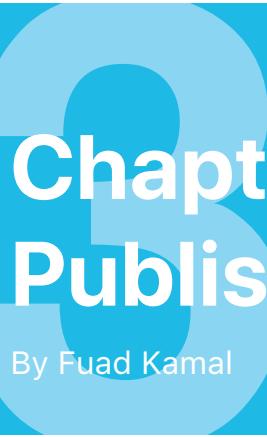
The **Determining content rating** and **Pricing and distribution** sections require you to first upload an APK to Google Play before you can fill this out. These sections are covered in the next chapter.

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 of 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 Fuad Kamal

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.

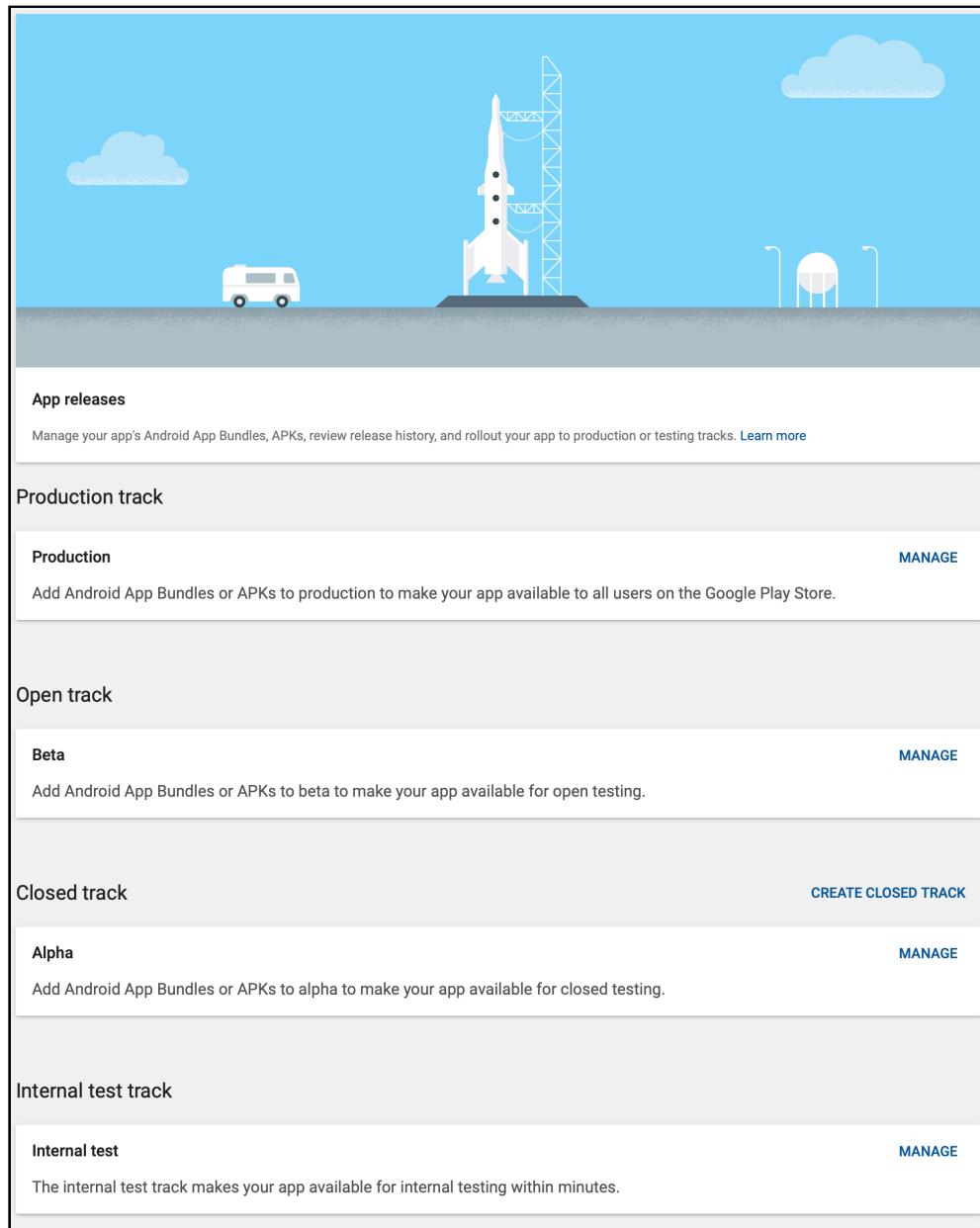
Time to dive into the details of each release type as well as **open** vs. **closed** testing.

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 on the name of your app in the list of apps to go to the dashboard view for that app. On the left side of the page, under **Release Management**, click **App releases**. The list of **Production**, **Beta** and **Alpha** release types are shown.



2. Under Alpha, click **MANAGE**.

This is where you can create a new Alpha release, upload the APK and manage the testers.

Create release

You can prepare, review, and then publish the version of your app you want to make available to users of the Play Store.

CREATE RELEASE

Manage testers Closed Alpha Testing ▲

You must upload an APK before you can configure testers.

Alpha country availability

Unavailable countries	Available countries
0	144
<small>+ rest of world synced with production</small>	

Manage country availability ▼

3. Click **CREATE RELEASE**.

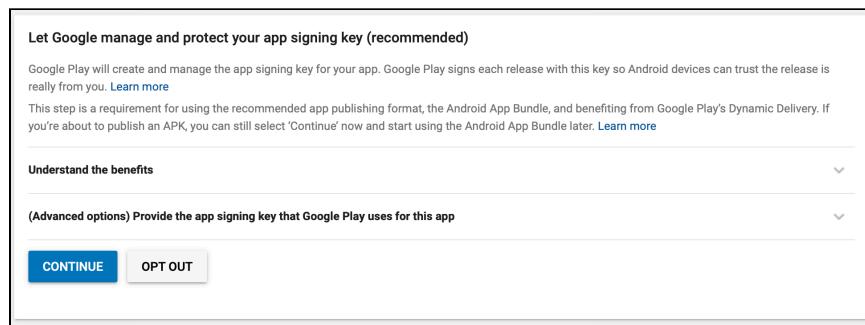
At this point, if you don't already have it set up, you'll be given an option to start using Google Play App Signing. For more information about app signing, refer to the previous chapter, "Preparing for Release".

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.

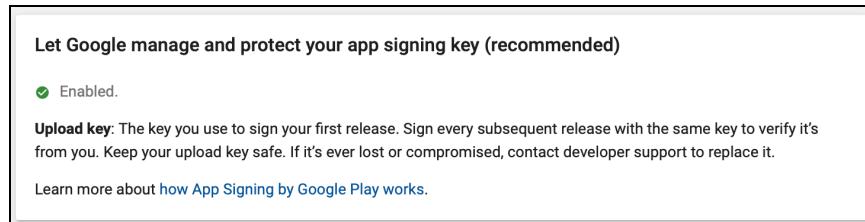
Using Google Play App Signing during this phase makes it much easier than turning it on later. Google automatically generates an app signing key and stores it; all you have to do is upload the App Bundle you already signed. The key you generated earlier becomes your upload key. If you decide to enable 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.

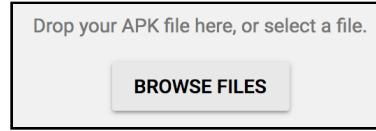


4. Click **CONTINUE** if you'd like to enroll in Google Play App Signing, and **ACCEPT** the terms of service.

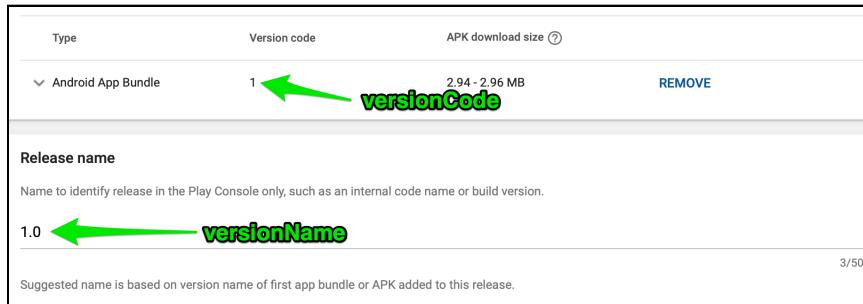
Google generates an app signing key and shows that Google Play App Signing is enabled.



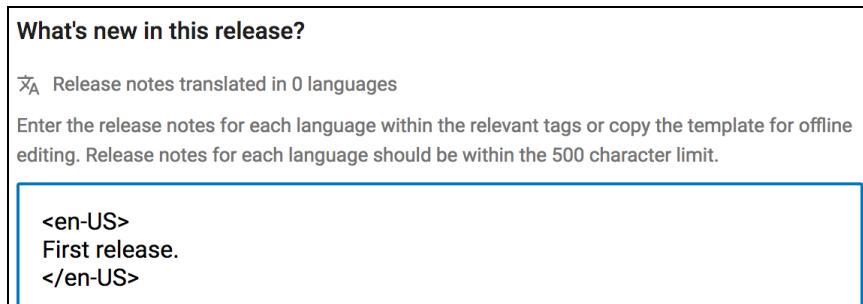
5. Click **BROWSE FILES** and select your signed release App Bundle or APK file.



6. This shows the **versionCode** for the app, taken from the setting in the app gradle file, and the type of file. End users won't see this code. In this example, you uploaded an App Bundle, so that is shown under the type. 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.

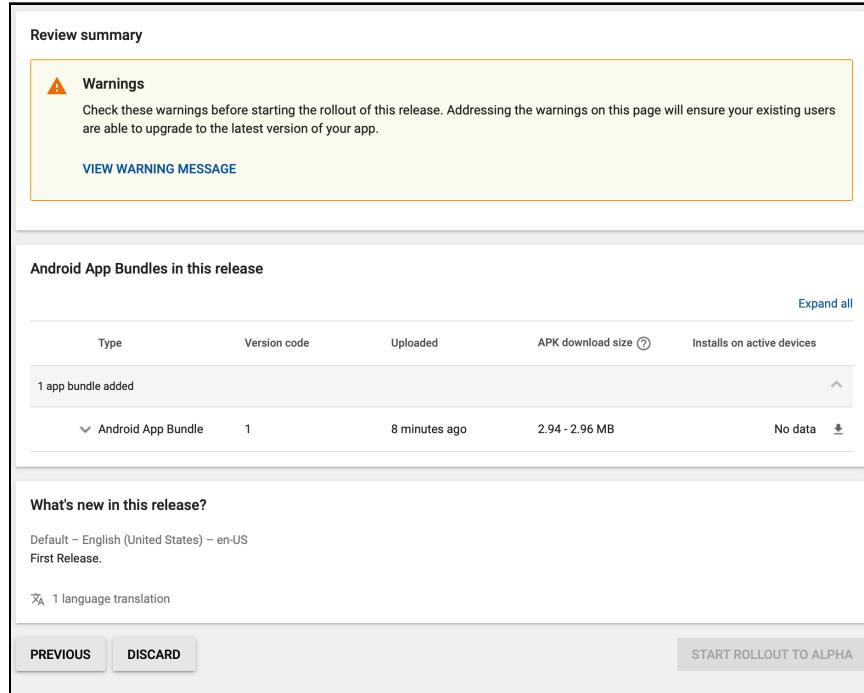


7. Enter the release notes for this version. Make sure to place the notes within the language tags as shown in the template.

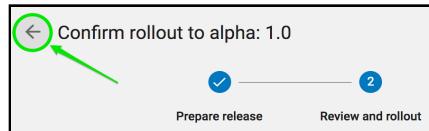


8. Click **SAVE**. Google validates what you've entered and then enables the **REVIEW** button.

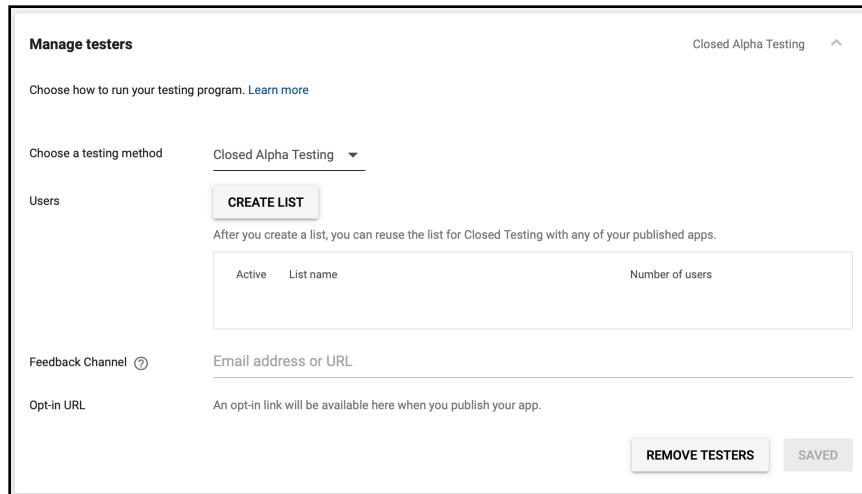
9. Click **REVIEW**. This shows a summary of the release and a warning that you need to add users before you can roll it out.



10. Click the back arrow to go back to the main Alpha release screen.



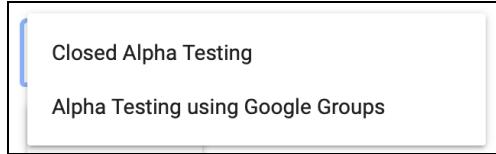
You'll see the **Manage testers** section at the top.



11. Select either **Closed Alpha Testing** or **Alpha Testing using Google Groups** testing method.

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 **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 address.

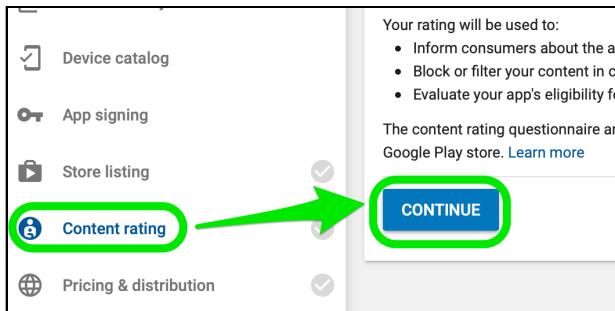
12. Fill in the required fields and click **SAVE**.

If everything checks out, the **App releases** row on the left shows a green check.



Determining the 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.

Email address *	rwdroidapprentice@gmail.com
Confirm email address *	rwdroidapprentice@gmail.com

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.

Please complete the questionnaire so that we can calculate your app rating.

 SOCIAL NETWORKING, FORUMS, BLOGS, AND UGC SHARING
App is a social networking app. Edit Category

APP TYPE

Is the primary focus of the app to connect people for the purposes of dating or sexual relationships or endeavors? * [Learn more](#)

Yes No

Does the app permit the public sharing of nudity? *

Yes No

Does the app permit the public sharing of real-world, graphic violence outside of a newsworthy context? *

Yes No

Does the app share the user's current physical location with other users? * [Learn more](#)

Yes No

Does the app allow users to purchase digital goods? * [Learn more](#)

Yes No

CALCULATE RATING **SAVE QUESTIONNAIRE**

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.

Rating System	Rating Category	Descriptors
Classificação Indicativa (ClassInd) Brazil	 Rated 12+	Inappropriate Language
Entertainment Software Rating Board (ESRB) North America	 Teen	
Pan-European Game Information (PEGI) Europe	 Parental guidance	Parental Guidance Recommended
Unterhaltungssoftware Selbstkontrolle (USK) Germany	 USK Ages 12+	USK Ages 12+
IARC Generic Rest of world	 Rated for 12+	Parental Guidance Recommended
Google Play Russia	 Rated for 12+	Parental Guidance Recommended
Google Play South Korea	 Rated for 12+	Parental Guidance Recommended

Click **APPLY RATING** to apply the rating to the store listing.

The **Content Rating** section on the left shows a green checkmark.

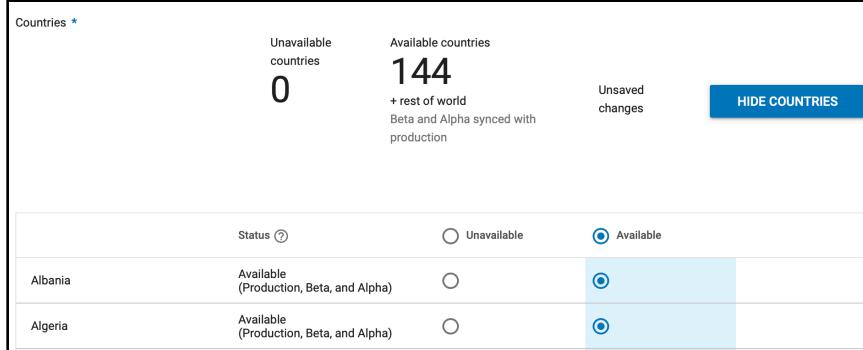
Pricing and distribution

You also need to provide pricing information and specify where your app will be distributed.

Click the **Pricing & Distribution** link.

You need to decide if you want your app to be Free or Paid. Once you mark it as “Free” and publish, you won’t be able to change it to a paid app later. 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 144 countries after making them all available.



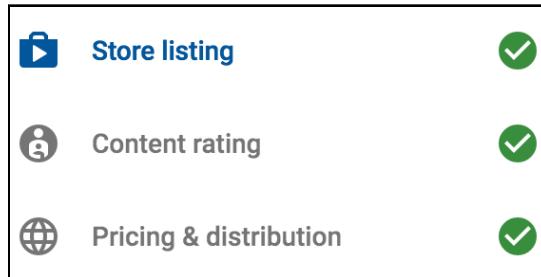
There are several items required on this page:

- **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 shows a green checkmark.

If you followed along and completed the Store listing section in the previous chapter, now there will only be green checkmarks and no grayed out ones on the menu items on the left.

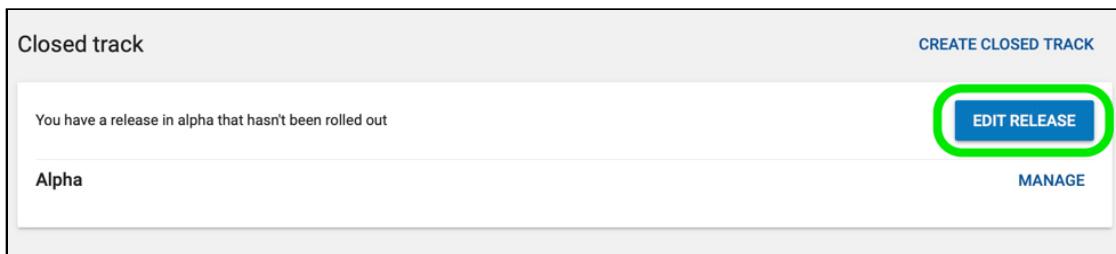


The top of the page indicates that your app is “Ready to publish”.



You're ready to roll out the Alpha release to your testers.

1. Click the **App releases** link on the left.
2. Click the **EDIT RELEASE** button under the Alpha release section.



1. Click the **REVIEW** button at the bottom of the page. This displays a summary of the release and waits for you to confirm the rollout.

← Confirm rollout to alpha: 1.0

1 Prepare release 2 Review and rollout

Review summary

This release is ready to be rolled out.

Android App Bundles in this release

Type Version code Uploaded APK download size ⓘ Installs on active devices

Type	Version code	Uploaded	APK download size ⓘ	Installs on active devices
1 app bundle added				
Android App Bundle	1	Feb 19, 7:35 PM	2.94 - 2.96 MB	No data

Expand all

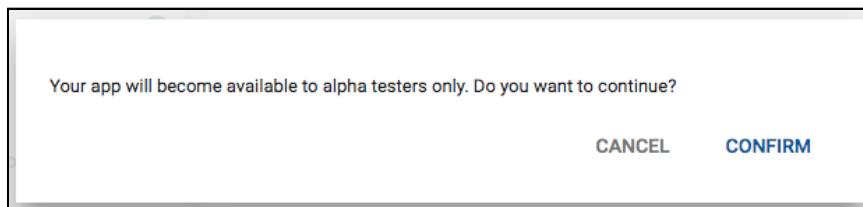
What's new in this release?

Default – English (United States) – en-US
First Release.

1 language translation

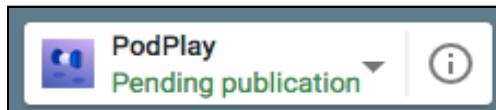
PREVIOUS DISCARD START ROLLOUT TO ALPHA

1. Click the **START ROLLOUT TO ALPHA** button at the bottom of the page if everything looks good.
2. Click **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.

Device model	Avg. CPU (Percent) ⓘ	Avg. network sent (Bytes/Sec) ⓘ	Avg. network received (Bytes/Sec) ⓘ	Avg. memory (Bytes) ⓘ	Startup time (ms) ⓘ
Galaxy S9 ⓘ	-	-	-	-	-
Xperia XZ1 Compact ⓘ	-	-	-	-	-
Pixel ⓘ	-	-	-	-	-
Pixel 2 ⓘ	-	-	-	-	-
Mate 9 ⓘ	-	-	-	-	-
Moto Z ⓘ	-	-	-	-	-
K3 2017 ⓘ	-	-	-	-	-
Pixel 2 ⓘ	-	-	-	-	-
P8 Lite ⓘ	-	-	-	-	-
Nokia 1 ⓘ	-	-	-	-	-
Moto G4 Play ⓘ	-	-	-	-	-

Pre-launch Performance 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 production.

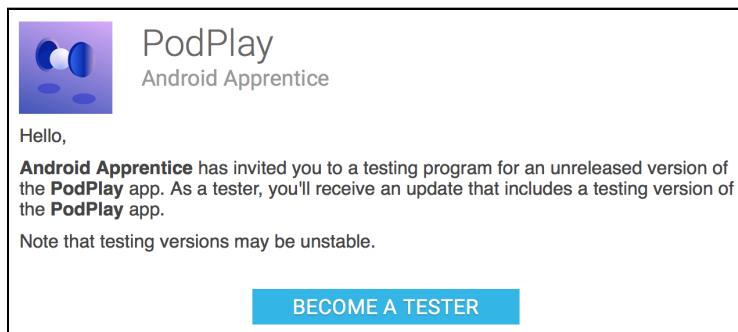
The screenshot shows a release summary for version 1.0. At the top, there's a "Release: 1.0" section with a play icon, a "Edit" button, and a note "Today, 3:42 PM: Full rollout." To the right is a large blue "RELEASE TO PRODUCTION" button. Below this, the "Rollout history" section shows "Today, 3:42 PM: Full rollout." Under "What's new in this release?", it says "Default – English (United States) – en-US" and "First Release." There's a note about "1 language translation" with a link to "Edit release notes". The "Android App Bundles and APKs" section has a table:

Type	Version code	Uploaded	APK download size	Installs on active devices
1 app bundle added	1	Feb 19, 7:35 PM	2.94 - 2.96 MB	No data

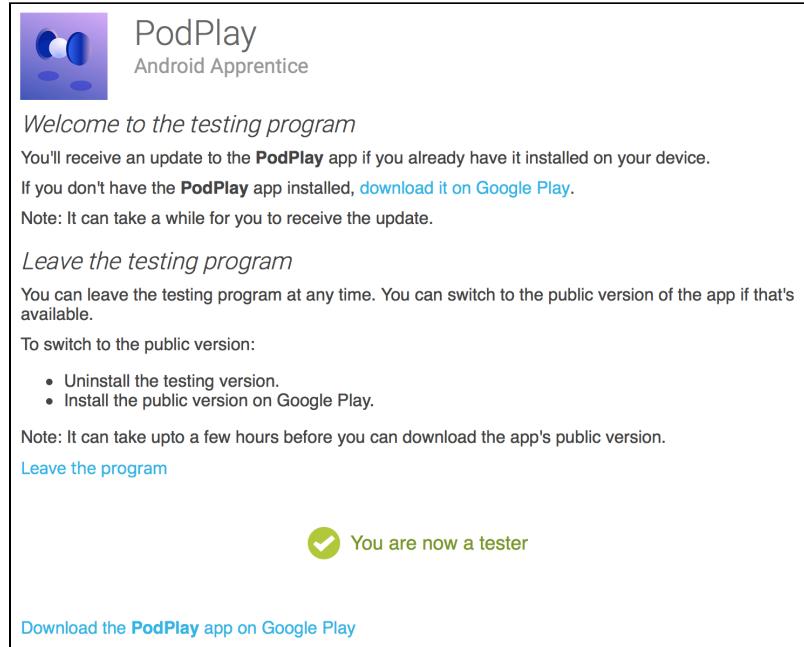
Expand the **Manage testers** section to view the new **Opt-in URL**. You can send this URL to your testers. Google won't send emails to your testers for you; you must notify them when the release is ready and include the Opt-in URL.

The screenshot shows the "Opt-in URL" section. It displays the URL <https://play.google.com/apps/testing/com.raywenderlich.podplay> and a "Share this opt-in link with your testers." button.

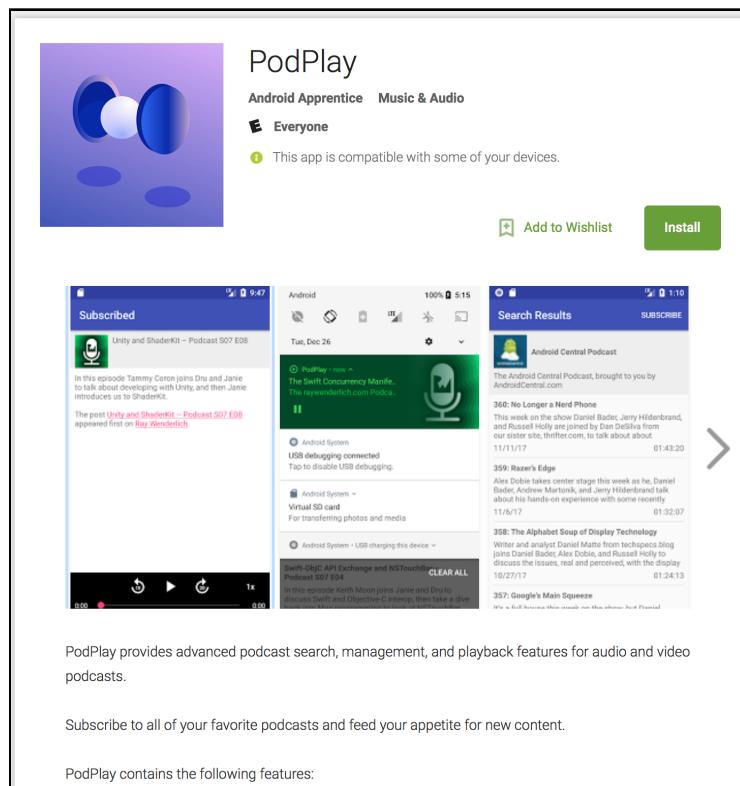
When a tester brings up the Opt-in URL, they'll see a message like the following:



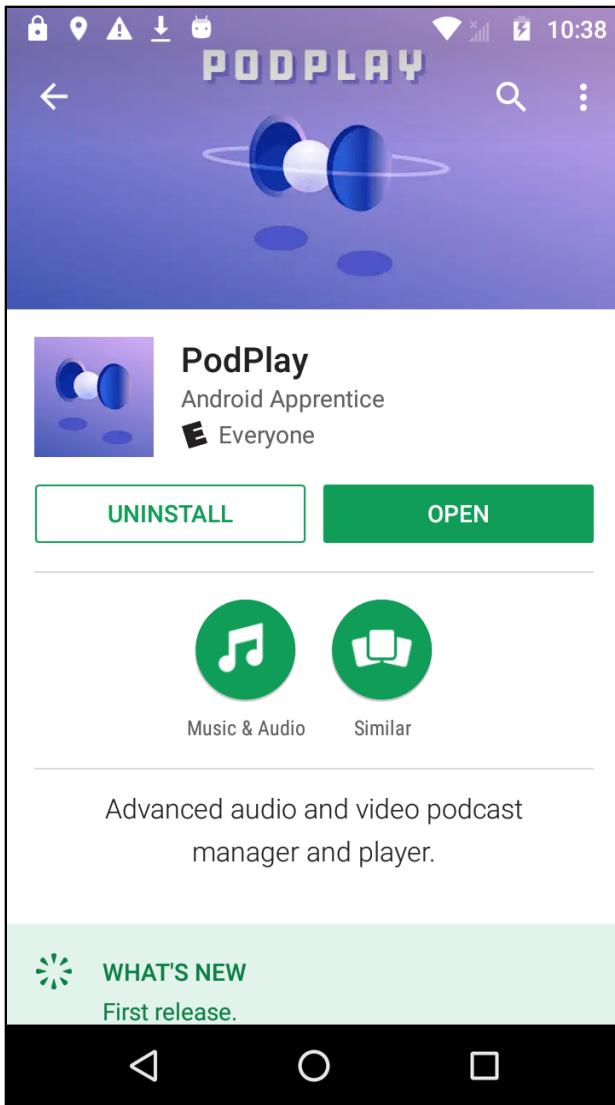
Once the user clicks **BECOME A TESTER**, they'll get a confirmation screen with a link to the Google Play store listing.



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 the 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 create multiple "closed tracks" beyond the Alpha track. For example, if you wanted to roll out a test feature to a select group of testers, separate from the Alpha group, you would click the "Create Closed Track" link and

go through the same process as with the Alpha track. Beta test groups are always an “Open Track”.

Go to the **App releases ▶ Beta** and click **Manage**.

This takes you to the release screen for Beta where you can create a release, manage testers, add release notes and select countries similar to the Alpha process you went through earlier. However note that now under “Manage Testers” there is only one option for “Choose a testing method”, which is open beta testing.

The screenshot shows the Google Play Console interface for managing a beta release. At the top, there's a decorative header with three icons: a paper airplane, a biplane, and a rocket ship. Below this, the main content area has the following sections:

- Create release**: A section for preparing and publishing the app version. It includes a "CREATE RELEASE" button.
- Manage testers**: A section for configuring the testing program. It shows the "Open Beta Testing" method selected. Other options include "Internal Testing" and "External Testing".
 - Choose a testing method**: Set to "Open Beta Testing".
 - Maximum number of testers**: A text input field with a note: "If you set a maximum number of testers, the number needs to be at least 1000."
 - Feedback Channel**: An input field for "Email address or URL".
 - Opt-in URL**: A text input field containing "https://play.google.com/apps/testing/com.anaara.podplay" with a copy icon.
 - Buttons**: "REMOVE TESTERS" and "SAVED".
- Beta country availability**: A summary of available countries.

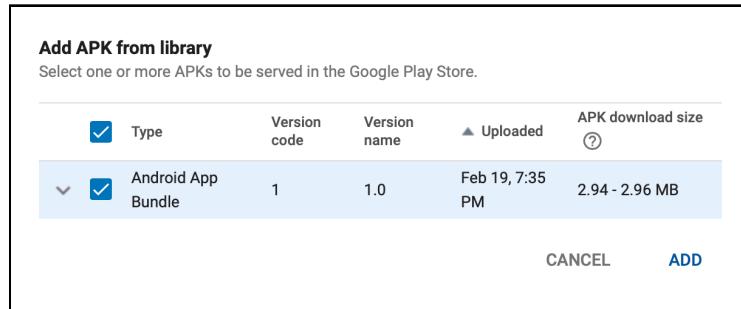
Unavailable countries	Available countries
0	144
+ rest of world	
synced with production	
- Manage country availability**: A link to manage country-specific settings.

Once you have set the testing method to open beta testing, click the **Create Release** button to go to the beta release preparation page. Enter a name for the beta under **Release Name**.

The screenshot shows the 'Prepare release' screen in the Google Play Console. At the top, there are two tabs: 'Prepare release' (which is selected) and 'Review and rollout'. Below the tabs, there's a section titled 'Let Google manage and protect your app signing key (recommended)' with a checked 'Enabled' option. It includes a note about the upload key and a link to learn more about App Signing. The next section is 'Android App Bundles and APKs to add', which contains a placeholder for dropping files or selecting them from a library. The 'ADD FROM LIBRARY' button is visible. Below this is a 'Release name' field containing 'first beta trial', with a character count of 16/50. A note says the suggested name is based on the version name of the first app bundle or APK added. The final section is 'What's new in this release?' with a note about release notes for multiple languages. It shows a code block for the en-US language: '<en-US>\n Enter or paste your release notes for en-US here\n</en-US>'. There are 'COPY FROM PREVIOUS RELEASE' and 'DISCARD' buttons at the bottom right, along with 'SAVE' and 'REVIEW' buttons.

If you had a new App Bundle or APK you wanted to use for this beta, you could upload it in the **Android App Bundles and APKs to add** section using the **browse files** button. However, if you wanted to use the same build you are already using, for example from an Alpha release, you can instead choose an existing build using the **ADD FROM LIBRARY** button.

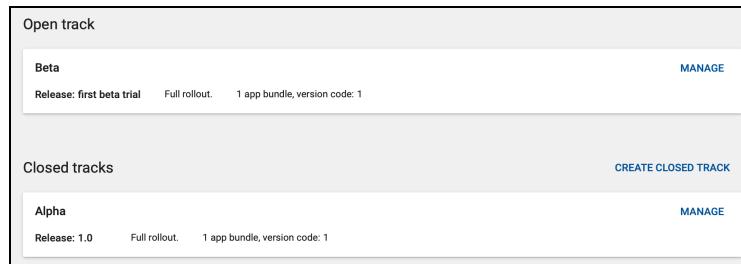
Click this button now and choose the build you previously used to set up the Alpha release.



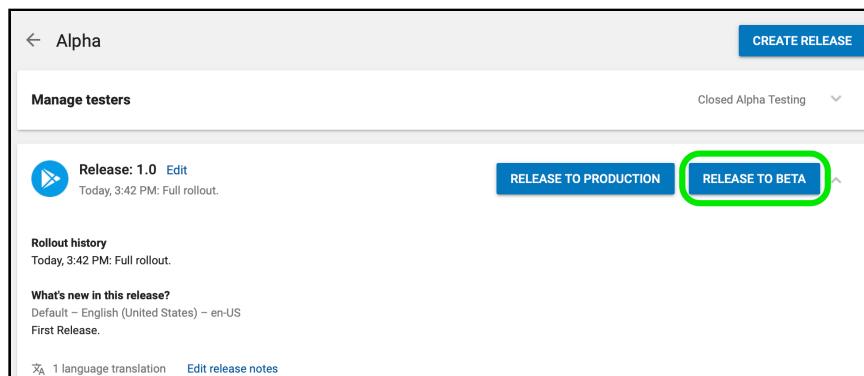
Click the **Add** button to add the build to the beta release. Update the release notes and click the **Save** button. Click the **REVIEW** button.

Click **START ROLLOUT**.

After the rollout to Beta is complete, the **App release** overview page shows the new Beta release.



If you go to manage the existing Alpha release now, you will see there is a new button there now, **Release to Beta**, which lets you directly promote an existing Alpha release to the Beta track.



Production release

Once the Alpha and Beta testing are 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 releases** ▶ **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 you're 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 of 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.

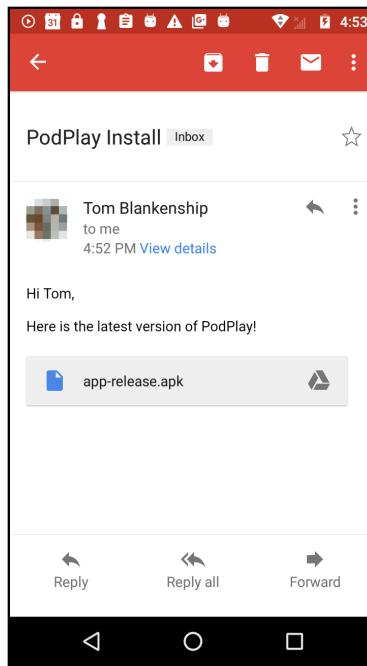
Users will need to configure their device to allow "unknown sources" before the APK can be installed. It's a good idea to include instructions in the email when sending out the APK.

If a user is running Android 8.0 or newer, they should look for the **Install unknown apps** section in the device settings.

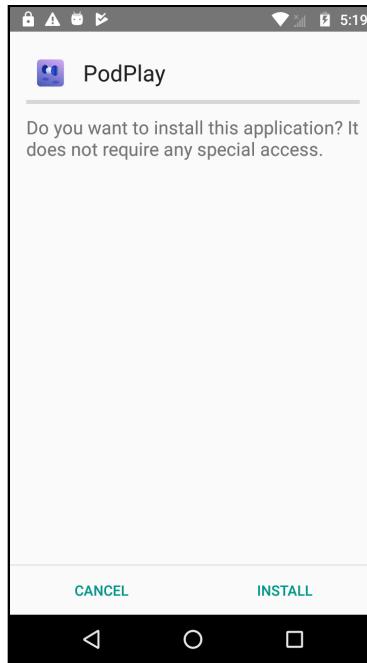
If a user is running a version before Android 8.0, they should enable **Unknown sources**

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 is no registration fee for developers on the Amazon Appstore, and it also offers some unique monetization models.

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, Fuad, Namrata, Kevin, Vijay, Ellen, Tammy, Eric and Manda

The *Android Apprentice* team

Learn Android programming with Kotlin!

Learning Android programming can be challenging. Sure, there is plenty of documentation, but the tools and libraries available today for Android are easily overwhelming for newcomers to Android and Kotlin.

Android Apprentice takes a different approach. From building a simple first app, all the way to a fully-featured podcast player app, this book walks you step-by-step, building on basic concepts to advanced techniques so you can build amazing apps worthy of the Google Play Store!

Who This Book Is For

This book is for anyone interested in writing mobile apps for Android. Though no previous mobile experience is necessary, this book is also a great resource for iPhone developers transitioning from iOS.

Topics Covered in Android Apprentice:

- ▶ **Getting Started:** Learn how to set up Android Studio and the Android Emulator.
- ▶ **Layouts:** Create layouts that can be used for both Activities and Fragments
- ▶ **Debugging:** No one's perfect! Learn how to dig down and troubleshoot bugs in your apps.
- ▶ **Communication:** Design separate Activities and communicate and send data between them using Intents.
- ▶ **Scrolling Layouts:** Learn how to use Recycler Views to make efficient, reusable views that scroll fluidly at a touch.
- ▶ **Google Places:** Integrate location APIs to bring the magic of maps into your Android apps.
- ▶ **Networking:** Learn how to access resources on the internet and handle networked responses.
- ▶ **Material Design:** Make sure your apps conform to modern best practices by using Google's standards of Material Design
- ▶ **And much, much more!**

One thing you can count on: after reading this book, you'll be prepared to write feature-rich apps from scratch and go all the way to submitting them to the Google Play Store!

About the Tutorial Team

The Tutorial Team is a group of app developers and authors who write tutorials at the popular website raywenderlich.com. We take pride in making sure each tutorial we write holds to the highest standards of quality. We want our tutorials to be well written, easy to follow, and fun.

If you've enjoyed the tutorials we've written in the past, you're in for a treat. The tutorials we've written for this book are some of our best yet — and this book contains detailed technical knowledge you simply won't be able to find anywhere else.