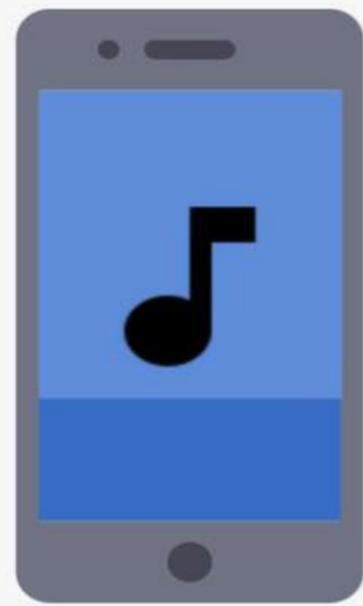


# Learn Android Programming

How to build seven Android apps using Kotlin



Adam Hawke

# Learn Android Programming

How to build seven Android apps using Kotlin

# Learn Android Programming

How to build seven Android apps using Kotlin

Copyright © 2022 Adam Hawke

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher or author, except in the case of brief sufficiently cited quotations embedded in critical articles or reviews. Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Coders' Guidebook, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book. Coders' Guidebook and the author have endeavoured to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, neither the author nor Coders' Guidebook can guarantee the accuracy of this information.

[www.codersguidebook.com](http://www.codersguidebook.com)

[Acknowledgements](#)

[Preface](#)

[About the author](#)

## **[Getting started](#)**

[An introduction to Kotlin and Android programming](#)

[Obtaining the example code for the projects in this book](#)

[Installing Android Studio](#)

## **[Testing and debugging an application](#)**

[Introduction](#)

[Testing an application on a virtual device](#)

[Testing an application on a real device](#)

[Using the Logcat to track system messages](#)

## **[How to create a Notes application](#)**

[Introduction](#)

[Getting started](#)

[Defining the string resources used in the app](#)

[Creating the Note data class](#)

[The layout for writing new notes](#)

[The layout for displaying notes](#)

[Enabling View Binding](#)

[The NewNote dialog window](#)

[The ShowNote dialog window](#)

[The add new note floating action button](#)

[Displaying note previews in a RecyclerView widget](#)

[Initialising the RecyclerView widget, adapter and dialog windows](#)

[Using JSON to save the user's notes](#)

[Creating the Settings activity and preferences file](#)

[Designing the themes](#)

[Summary](#)

## **[How to create a Camera application](#)**

[Introduction](#)

[Getting started](#)

[Configuring the Gradle Scripts](#)

[Defining the string resources used in the app](#)

[Requesting permission to access the device's camera and storage](#)

[Setting up the Camera fragment and layout](#)

[Connecting to a camera and displaying a live feed](#)

[Capturing photos](#)

[Creating the Photo data class](#)

[Loading images from the user's device](#)

[Setting up the Gallery fragment and layout](#)

[Displaying image previews](#)

[Deleting an image from the MediaStore](#)

[Setting up the Photo Filter fragment](#)

[Navigating to the photo filter fragment](#)

[Applying filters to a photo](#)

[Saving the image once a filter has been applied](#)

[The BottomNavigationView widget](#)

[Summary](#)

## **[How to create a Weather application](#)**

[Introduction](#)

[Getting started](#)

[Configuring the Gradle scripts and Manifest file](#)

[Defining the string resources used in the app](#)

[Setting up the weather API](#)

[Designing the activity main layout](#)

[Finding the user's location](#)

[Requesting data from the OpenWeather API](#)

[Displaying weather data to the user](#)

[Loading weather data for a specific city](#)

[Enabling swipe-to-refresh](#)

[Summary](#)

## **How to create a Treasure Map application**

- [Introduction](#)
- [Getting started](#)
- [Defining the string resources used in the app](#)
- [Setting up the Google Maps SDK](#)
- [Designing the activity maps layout](#)
- [Requesting user permissions](#)
- [Finding the user's location](#)
- [Create a Geofence around the treasure location](#)
- [How to add a visible circle around the Geofence area](#)
- [Detect Geofence alerts](#)
- [Respond to Geofence area enter transitions](#)
- [Plotting markers on the map](#)
- [Initiating the treasure hunt](#)
- [Monitoring the device's location](#)
- [Monitoring the device's orientation](#)
- [Summary](#)

## **How to create a phone call and SMS communications app**

- [Introduction](#)
- [Getting started](#)
- [Configuring the manifest file](#)
- [Defining the string resources used in the app](#)
- [Defining the drawable resources used in the project](#)
- [Styling the phone's dial pad buttons](#)
- [Designing the phone dial pad layout](#)
- [Creating the phone dial pad fragment](#)
- [Initiating phone calls](#)
- [Designing the phone call log layout](#)
- [Creating the call log fragment and call log adapter](#)
- [Retrieving the device's call history](#)
- [Displaying the call log](#)
- [Handling user interactions with call log events](#)
- [Designing the SMS fragment layout](#)
- [Creating the SMS fragment and SMS adapter](#)
- [Retrieving the device's recent SMS messages](#)
- [Displaying the SMS messages](#)
- [Handling user interactions with SMS messages](#)
- [Viewing full SMS messages](#)
- [Sending an SMS message](#)
- [Detecting newly received SMS messages](#)
- [The BottomNavigationView widget](#)
- [Summary](#)

## **How to create a mobile Store application**

- [Introduction](#)
- [Getting started](#)
- [Configuring the Gradle scripts](#)
- [Configuring the Manifest file](#)
- [Defining the string resources used in the app](#)
- [Setting up the Products fragment and layout](#)
- [Storing product information](#)
- [Displaying products in the Products fragment](#)
- [Using the currency exchange rate API](#)
- [Changing the active currency](#)
- [Updating the prices of products](#)
- [Setting up the Checkout fragment and layout](#)
- [Preparing the Checkout adapter](#)
- [Displaying the user's shopping basket](#)
- [Calculating the total price of the user's order](#)
- [Setting up the Braintree payments API](#)
- [Linking your Braintree account with your PayPal business account](#)
- [Configuring your Braintree account to accept payment in multiple currencies](#)
- [Generating a client token](#)
- [Initiating a PayPal transaction using Braintree](#)

[Processing transactions](#)  
[Clearing the shopping basket following successful payment](#)  
[The BottomNavigationView widget](#)  
[Going live](#)  
[Summary](#)

## **[How to create a Music application](#)**

[Introduction](#)  
[Getting started](#)  
[Configuring the Gradle scripts](#)  
[Configuring the Manifest file](#)  
[Defining the string resources used in the app](#)  
[Customising the application's themes](#)  
[Creating the Song data class](#)  
[Configuring the Room SQLite database](#)  
[The music data access object \(DAO\)](#)  
[The database repository](#)  
[The music and playback view models](#)  
[The app toolbar menu](#)  
[Defining the drawable resources used in the project](#)  
[The media browser service - part 1](#)  
[The media browser service - part 2](#)  
[Interacting with the media browser service](#)  
[Initiating playback](#)  
[Preparing the fragment packages](#)  
[Designing the activity\\_main layout](#)  
[The main navigation graph](#)  
[The navigation drawer](#)  
[Setting up the Library fragment](#)  
[Setting up the Songs fragment and layout](#)  
[Displaying songs in the RecyclerView](#)  
[Handling changes in the user's music library](#)  
[Handling user interactions with songs](#)  
[Setting up the EditSongInfo fragment and layout](#)  
[Saving changes to a song's metadata](#)  
[Setting up the PlayQueue fragment and layout](#)  
[Displaying the play queue](#)  
[Reordering items in the play queue](#)  
[Interacting with the play queue](#)  
[Setting up the Search fragment and layout](#)  
[Displaying the search results](#)  
[Handling user interactions with the Search fragment](#)  
[The playback controls navigation graph](#)  
[Setting up the PlaybackControls fragment and layout](#)  
[Responding to the playback controls](#)  
[Setting up the Currently Playing fragment and layout](#)  
[Handling user interactions with the CurrentlyPlaying fragment](#)  
[Building the music library](#)  
[Maintaining the music library](#)  
[Summary](#)

## **[Releasing an application on the Google Play store](#)**

[Introduction](#)  
[Preparing your application for publication](#)  
[Creating an Android Application Bundle](#)  
[Creating a Google Play store listing](#)  
[Privacy policy](#)  
[Ads](#)  
[App access](#)  
[Content ratings](#)  
[Target audience and content](#)  
[News apps](#)  
[COVID-19 contact tracing and status apps](#)  
[Data safety](#)  
[Publish your Android App Bundle](#)

## [Summary](#)

## Acknowledgements

I would like to gratefully thank the following creators who made icons that are used throughout this book and in the Android apps. All icons were sourced from [www.flaticon.com](http://www.flaticon.com) and are free for commercial use with attribution:

- The smartphone icon used on the book cover was made by Freepik
- The phone dial pad icons used in the Communication application were made by Pixel Buddha.
- The carrot, broccoli and strawberry icons used for demonstration products in the Store application were made by Icongeek26.
- The play, pause, skip forward and skip backward playback control icons used in the Music application were made by Elias Bikbulatov.

## Preface

This book is written for Android developers of all abilities, including those who have never coded before. Throughout the book, the reader will learn how to build seven Android applications from different categories. The book also covers how to test Android applications on real and virtual devices and publish Android applications to the Google Play store.

The Android apps covered in this book are ordered by complexity. In building the apps and engaging with the content, the reader should build a thorough understanding of Kotlin-based Android programming and modern Android development practices. By the end of the book, the reader will possess both the knowledge and confidence to develop Android applications of their own.

## About the author

Hi there! My name is Adam Hawke, the author of the book you are about to read. I hope you will find this book resourceful and the springboard for your Android development journey. Whether you are looking to pursue Android development professionally or simply curious to learn a new skill, this is the book for you. Within these pages, you will find a wealth of knowledge on cutting-edge Android development and Kotlin programming techniques that you can apply to any future project you undertake.

A little bit about myself: I am a software developer with experience working in Biomedical Sciences research and the Energy industry. Currently, I am based in the UK, helping the government to build the digital infrastructure required to trade with the rest of the world. In my spare time, I manage a coding tutorials website called [codersguidebook.com](http://codersguidebook.com). My mission is to equip everyone with the knowledge and skills required to achieve their programming goals, without all the jargon and confusing instructions found elsewhere.

Android programming is my favourite area to work in. I have developed and released many applications, from a Spotify-based app that generates playlists based on the user's mood to a productivity app that formulates actionable plans for achieving goals. The possibilities are vast, and the Android development community is one of the most active and innovative in the world. Some techniques covered in this book were released just weeks before publication! Furthermore, I purposefully designed projects that cover a range of categories, so you will develop a comprehensive skill set that you can draw upon whatever direction you decide to take in your Android development journey.

# Getting started

## An introduction to Kotlin and Android programming

Before we jump into the projects, it is worth covering a few key concepts that will recur throughout the book. A good foundation of knowledge is critical to your success as an Android developer, so here's what you should know before getting started. First, the applications covered in this book are powered by a programming language called Kotlin. Kotlin was developed by a company called JetBrains. The first stable release became available in 2016. While Kotlin was released relatively recently, it is heavily based on an older programming language called Java. Kotlin and Java are interoperable, which means you can use both languages in the same project.

So why use Kotlin if it is so similar to Java? In short, Kotlin is concise, expressive and safe. Kotlin creates powerful applications that perform well and are easy to maintain. The benefits of Kotlin are such that in 2019 Google announced that Kotlin is the preferred programming language for Android applications. We will discuss Kotlin in greater detail throughout the book. For now, it is sufficient to understand the following key concepts:

- Kotlin is an object-oriented programming language. In programming terms, an object is an instance of a class, and a class contains data and code. The class acts as a blueprint for the object, and a single class can generate unlimited objects. For example, imagine you had a recipe book. If the recipe book was the application, then each recipe would be an object of a class called Recipe.

```
class Recipe {  
      
}
```

- Objects contain data. Each field of data is called a parameter, and a parameter can be stored as a variable. Variables can be referenced elsewhere in the class, or even shared between different objects. For example, the ingredients of a recipe could be considered parameters.
- Variables can take various forms. If the value of a variable is fixed then it is initialised using the `val` keyword, whereas if the value can be changed then it is initialised using the `var` keyword. Also, variable values can be null. Null variables are deemed as valueless; however, if you initialise a null variable with the `var` keyword then you can assign it a value later. It is convention to write variable names using camel case. Camel case means the variable name is written without spaces, with the first word lowercase and the first letter of subsequent words capitalised. For example, if the variable name was 'My first variable' then you would write it as `myFirstVariable`

```
var nullVariable = null
```

- Often, the value assigned to a variable falls under one of the following primitive data types:
  - Char - A single character.  

```
val grade = 'D'
```
  - String - Text such as a word or sentence.  

```
val carrot = "carrot"
```
  - Integer - A whole number. Can be negative, positive or 0.  

```
var ovenTemperature = 180
```
  - Float - A number containing decimal places.  

```
val litresOfWater = 0.5
```
  - Boolean - A true/false value.  

```
var veganFriendly = true
```

In addition to primitive data types, you can also assign objects or collections such as lists to variables.

- Kotlin classes can store the instructions for completing a given task inside a method. Methods can return values, and those values can be stored in variables. Also, methods can feature arguments, which are items of data that must be supplied to the method when it is called. For example, the below code defines a method called `temperatureDifference`, which calculates the difference between two temperatures. The method requires two arguments to be supplied in Integer format: the current temperature and the target temperature. The method calculates and returns the difference between the temperatures. In this case, the result is stored in a variable called `tempDifference`.

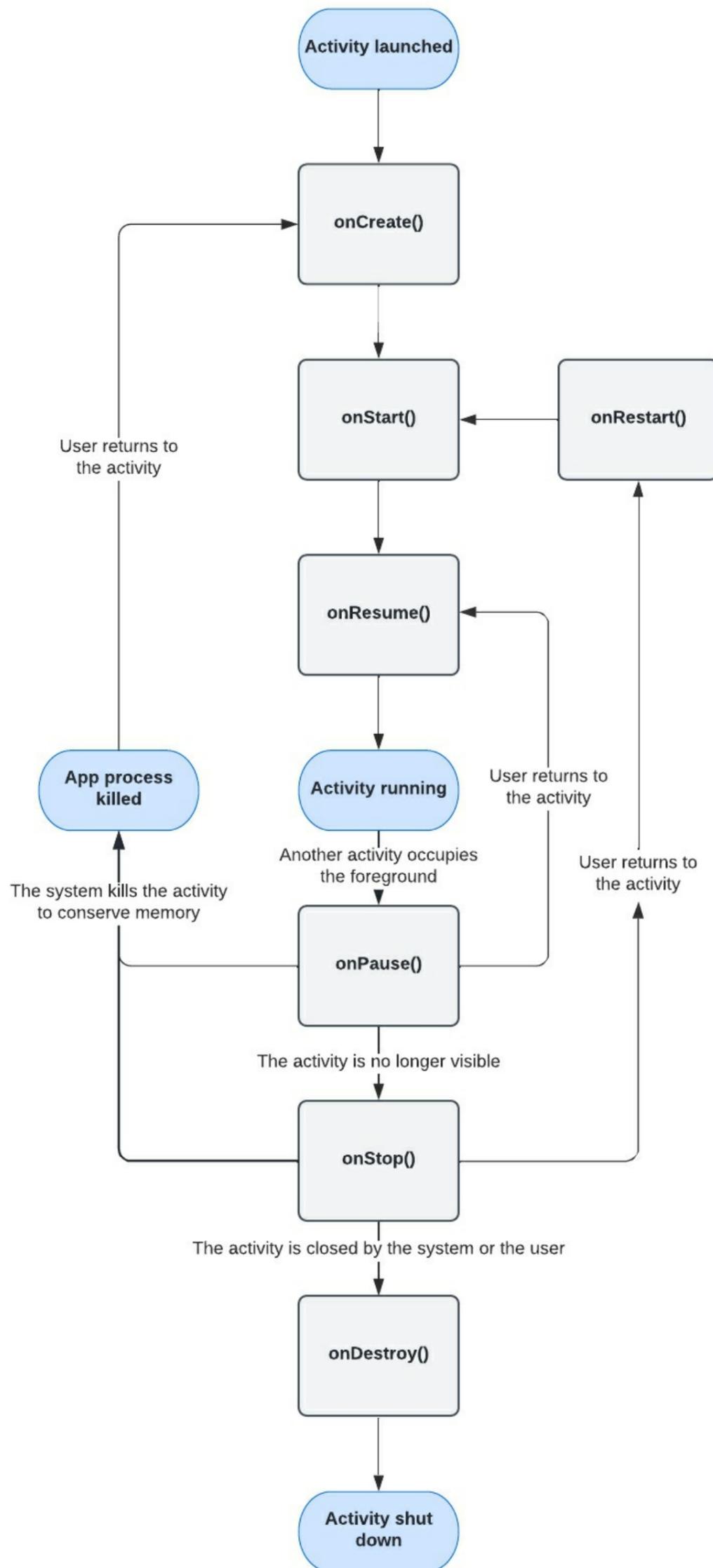
```
val tempDifference = temperatureDifference(80, 200)
```

```
fun temperatureDifference(currentTemperature: Int, targetTemperature: Int): Int {  
    val difference = targetTemperature - currentTemperature  
    return difference  
}
```

Kotlin is a vast subject area. It can be confusing to grasp certain concepts, especially when you are new to programming. The above bullet points provide a brief overview of the core Kotlin principles required to develop Android applications. It's OK if not everything makes sense immediately, we will revisit these concepts regularly throughout the book and gradually explore more advanced techniques as we progress.

Another topic that you should be aware of is the lifecycle of Android activities and fragments. An activity is a module that serves a role within the application, while a fragment is a section of an activity that features a unique user interface. An activity can use zero or multiple fragments. Also, activities are reusable and can be shared between applications. For example, an email application may contain an activity that allows the user to draft and send an email message. If you were designing an application that required the ability to send an email, then you could incorporate the activity from the email application rather than creating the feature from scratch.

At runtime, an activity will cycle between several states in a sequence called the Android activity lifecycle, as summarised in the below diagram. The first stage of the lifecycle is called `onCreate`. The `onCreate` stage is where user interface components are initialised and any data operations required to set up the activity are performed. Next, the activity progresses to the `onStart` stage, where the user interface becomes visible. The `onStart` stage is followed by the `onResume` stage once the activity is ready to handle user interactions. The activity will then occupy the system foreground for as long as it is visible to the user. The foreground activity is prioritised when the device allocates system memory and computational processing power.



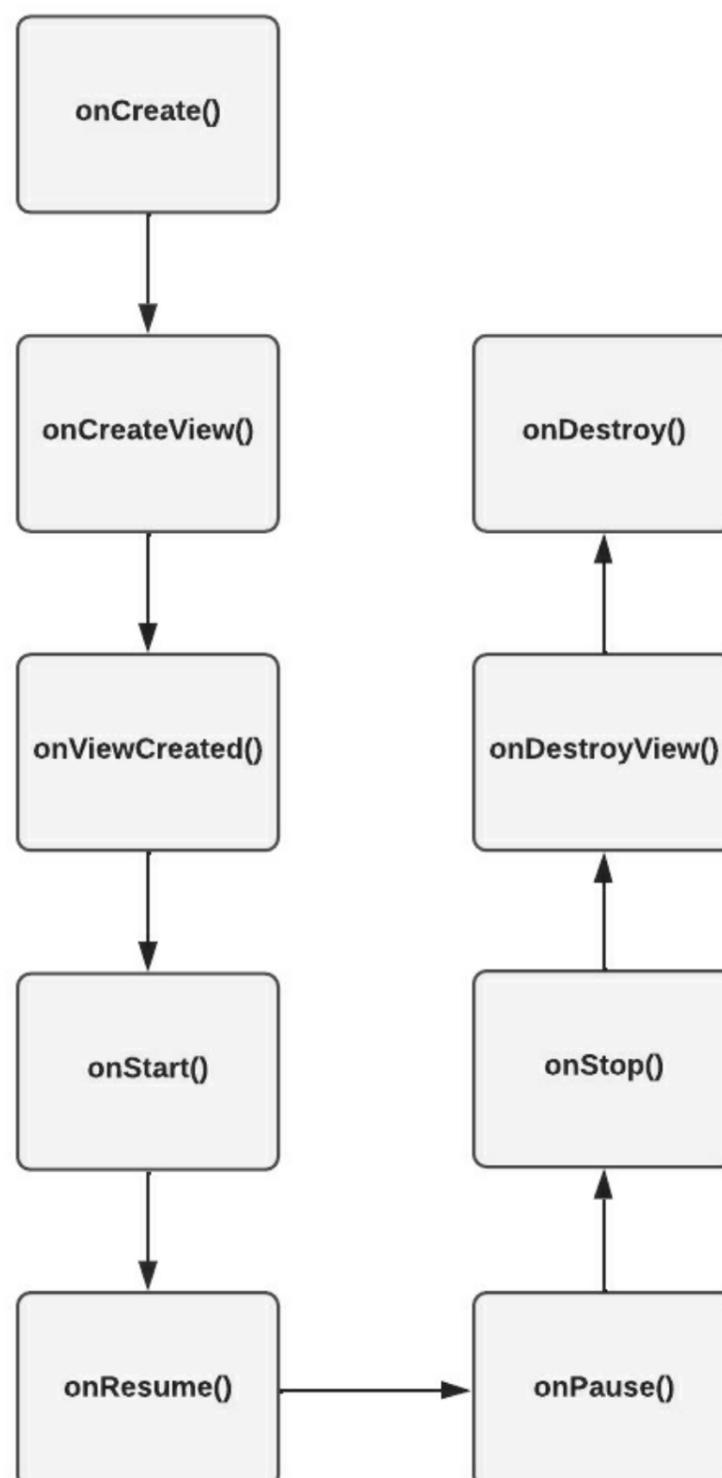
If an activity loses its foreground status, like when the user closes the application or another activity occupies the foreground, the losing activity will enter the `onPause` stage of its lifecycle. The `onPause` stage will transition to the `onStop` stage when the activity is no longer visible. At either the `onPause` stage or the `onStop` stage, the activity can reenter the foreground state if the user returns to the activity. If the user returns to the activity when it is in the `onStop` stage, then the activity must progress through a stage called `onRestart` before the activity can become visible

again.

An important consideration for activities that enter the onPause and onStop stages is that those activities may be killed if there is insufficient memory to keep them running in the background. If the user returns to a killed activity, then the activity must start over from the onCreate stage. Finally, if the activity is closed and the user does not return, then the activity will progress to the onDestroy stage. The onDestroy stage handles any procedures that should be carried out before the activity is destroyed.

As mentioned previously, an activity can comprise multiple fragments. Each fragment contains a lifecycle of its own, albeit directly interrelated with the activity lifecycle. For example, if the fragment's parent activity closes then the fragment will shut down also. The fragment lifecycle shares many stages with the activity lifecycle; however, the fragment lifecycle does not include the onRestart stage. Also, the fragment lifecycle contains several extra stages relating to the fragment's user interface view, as summarised below:

- **onCreateView()** - The fragment is loading. In this stage, you should specify the layout resource file that will be used for the user interface.
- **onViewCreated()** - The user interface layout is now available. In this stage, you should initialise the layout's components and handle user interactions.
- **onDestroyView()** - The fragment is shutting down. In this stage, the user interface layout is detached from the fragment and destroyed.



In the projects covered in this book, we will often instruct activities and fragments to perform actions at various stages in their respective lifecycle. For this reason, it is important to understand the general flow of events and how they relate to one another.

## Obtaining the example code for the projects in this book

To get the most out of this book, you might like to download the example code for the projects that we will be creating. You can download all of the code in a single ZIP folder by heading to the following URL:  
<https://codersguidebook.com/books/seven-android-apps>

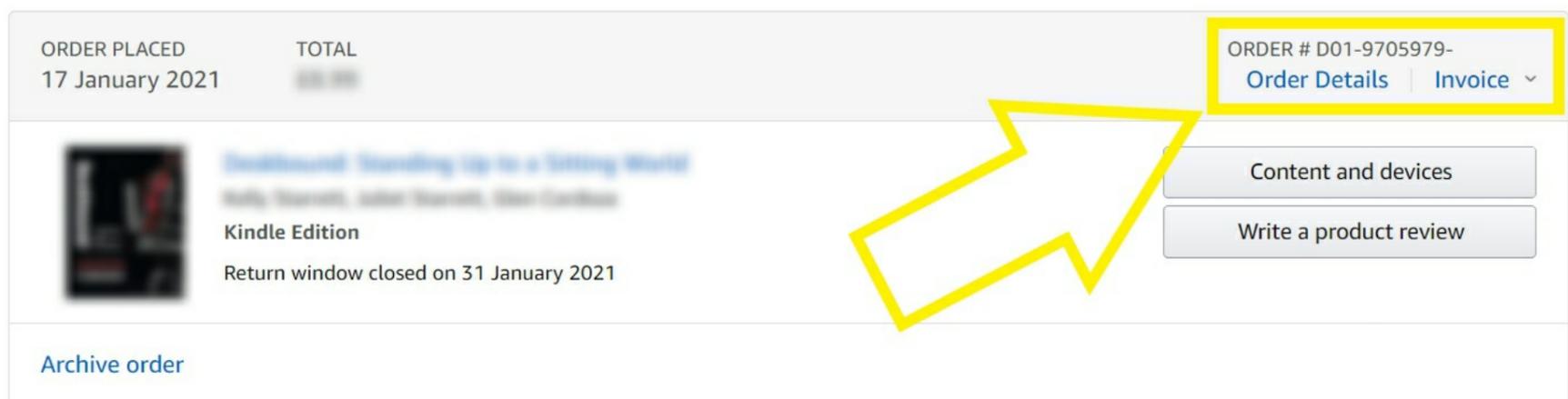
On the webpage, you should find a form to enter your Amazon order number.

### Learn Android Programming: How to build seven Android apps using Kotlin

Thank you for purchasing our book 'Learn Android Programming: How to build seven Android apps using Kotlin'. To download the example code for the Android Studio projects covered in the book, simply enter your Amazon order number in the form below. If you have not yet purchased the book then you can grab a copy [here](#).



You can find your order number by logging into your Amazon account, navigating to the Your Orders section and finding your order of 'Learn Android Programming: How to build seven Android apps using Kotlin'. The order number should be in the top right corner of the order summary. Do not include the hashtag when entering the order number into the download box.



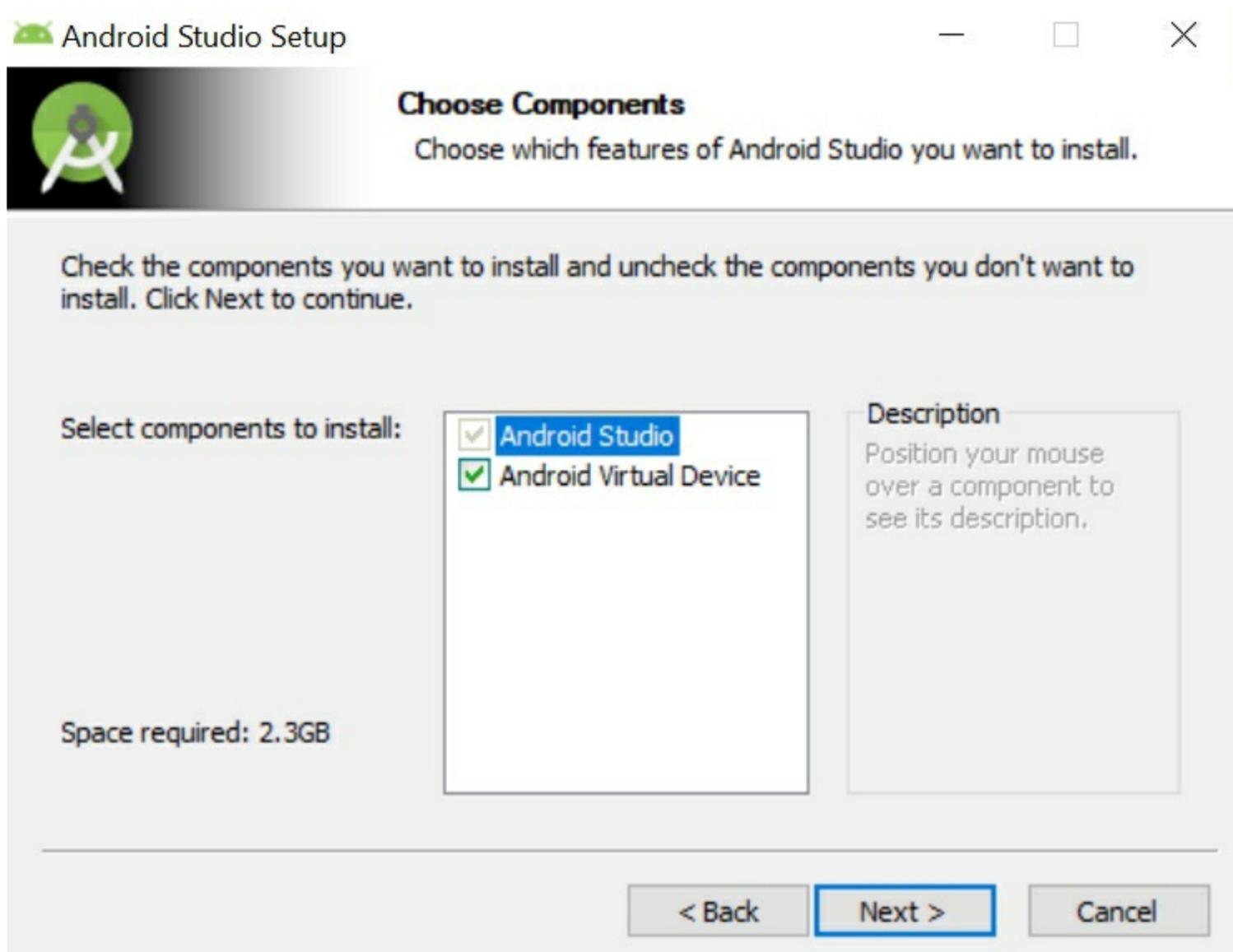
Once you have entered your order number into the form, click the download example code button to download a ZIP folder containing the project code to your computer.

If you have any questions or problems downloading the code then feel free to contact us at [hello@codersguidebook.com](mailto:hello@codersguidebook.com)

## Installing Android Studio

To develop Android applications you will need to install a free software programme called Android Studio. Android Studio is the official development environment for building and testing Android apps and is available for Windows, macOS and Linux operating systems. You can download the latest version of Android Studio from Android's website: <https://developer.android.com/studio>. The apps in this book were built using Android Studio Arctic Fox, which is the most up-to-date release at the time of writing. A newer version of Android Studio may be available when you read this. It is fine to use a newer version of Android Studio; however, your user interface may look a little different to the screenshots included in this book.

During the installation process, it is usually fine to select the default/standard options; however, you may want to install the Android Virtual Device when prompted. The Android Virtual Device is a virtual phone and tablet emulator that allows you to test your apps using Android Studio, which is handy if you do not have many physical devices present. In the next chapter, we will explore how to configure a virtual device.



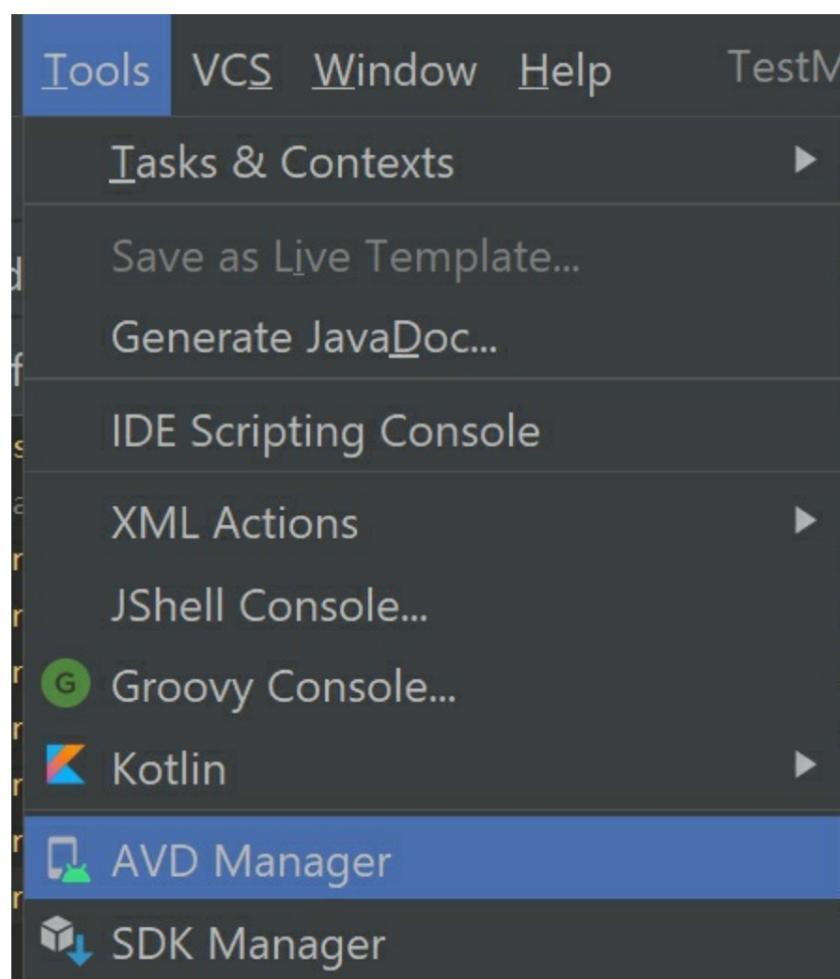
# Testing and debugging an application

## Introduction

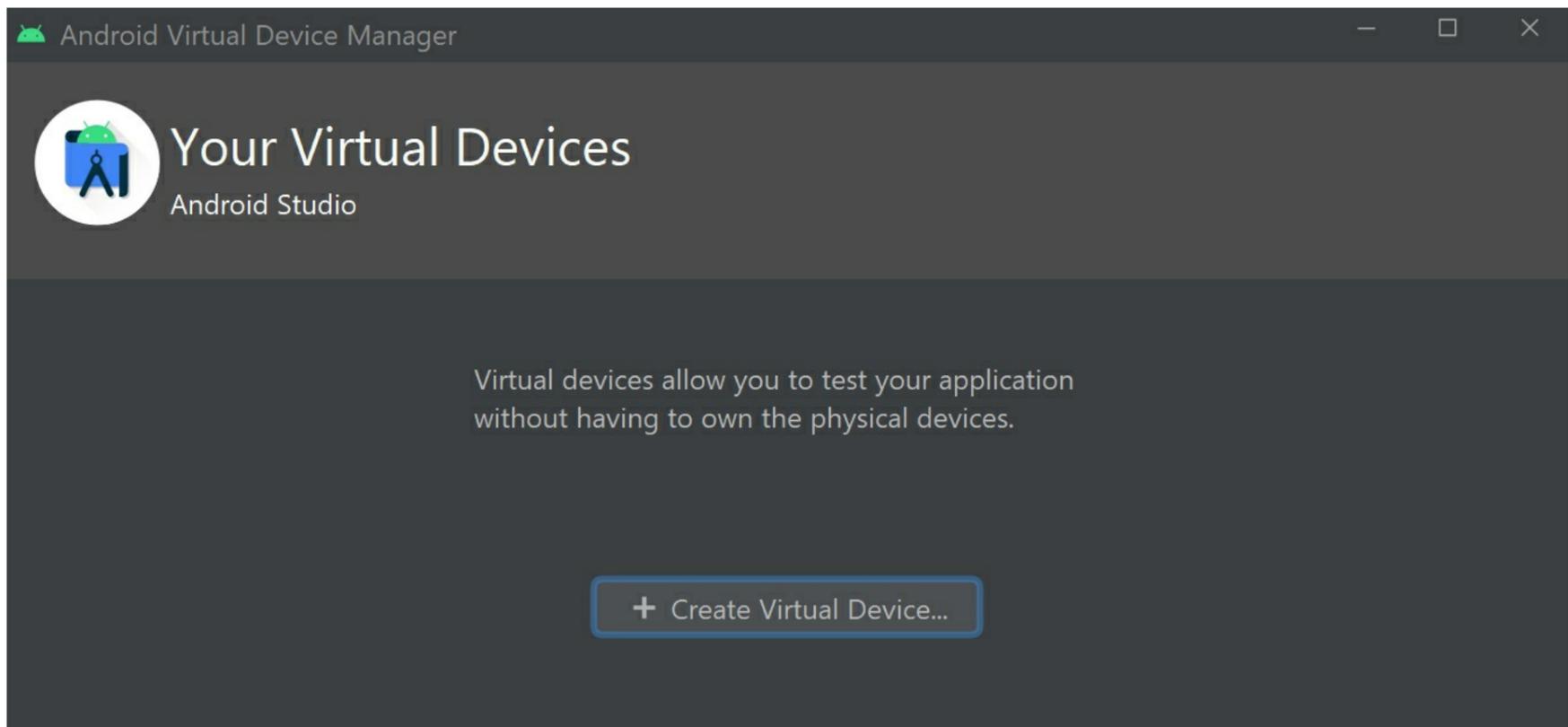
As you progress on your Android development journey, you will likely want to try the applications you create. For this purpose, you can install your applications on real and virtual devices using Android Studio. In this chapter, we will cover both approaches and discuss Android Studio's virtual emulator, which allows you to test applications on various virtual phones and tablets. As you test your applications, you will undoubtedly encounter bugs. Bugs are defects that can cause your application to crash or work in unexpected ways. To facilitate the debugging process, this section will also explore an Android Studio tool called Logcat. The Logcat contains a log of system messages. We will discuss how to use the Logcat to monitor application processes and fix bugs.

## Testing an application on a virtual device

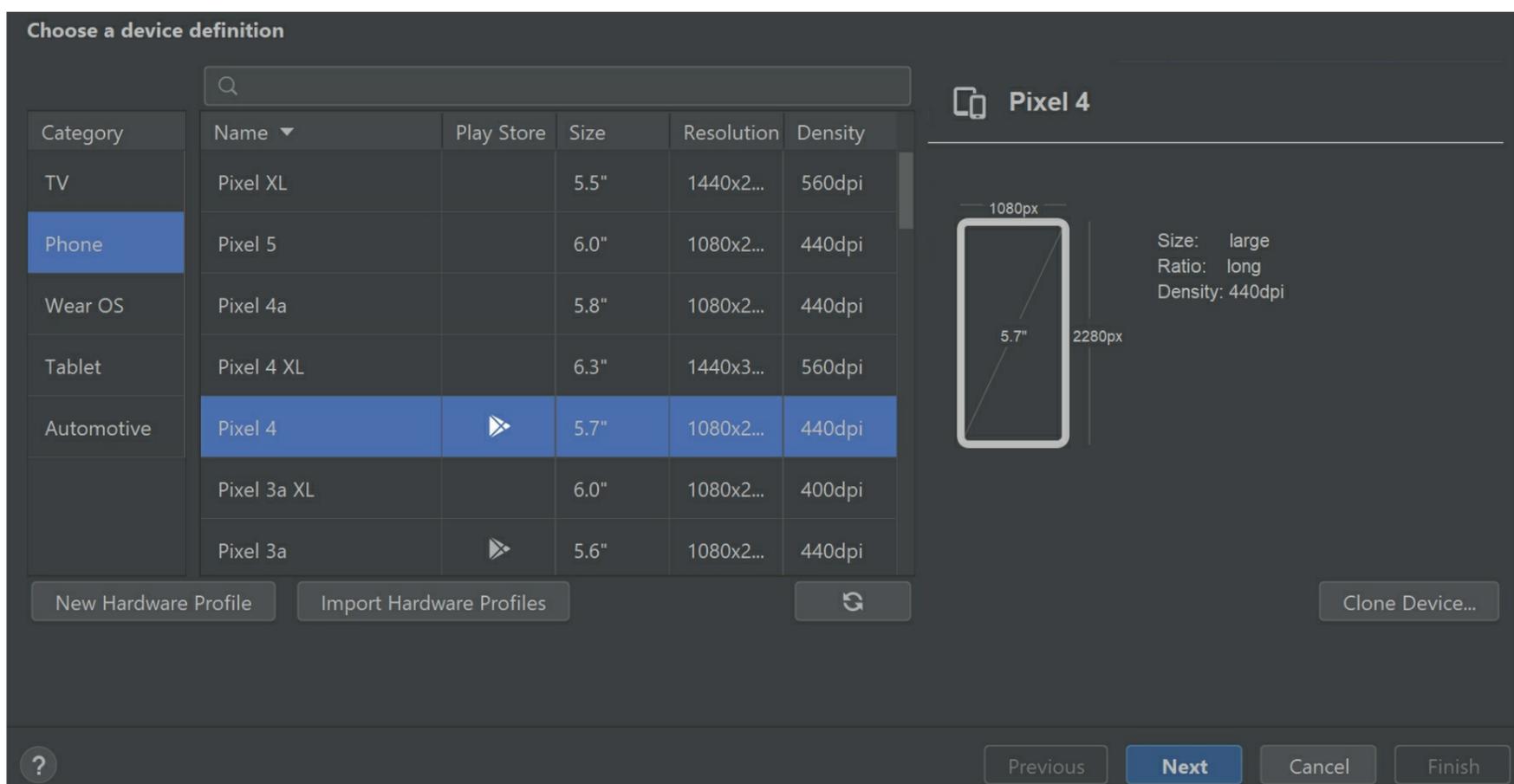
Android Studio features an emulator that allows you to run your app on a variety of virtual devices including mobile phones and tablets. The virtual emulator is a great way of testing your application in different environments, without spending money on physical devices. To create a virtual device, click **Tools > AVD Manager**.



Next, in the Android Virtual Device Manager window, click Create Virtual Device.



You will see a list of possible virtual devices. Each device has a different screen size, resolution and pixel density. You can also click the New Hardware Profile button and customise the device's technical specifications including the screen resolution, storage capacity, and camera availability. For this example, we will select the Pixel 4 phone then click Next. No hardware modifications are required.



You will be prompted to select an API release. Most often you will want to select an API equal to or greater than the minimum API for new apps submitted to the Google Play store. You can find the API requirements by reading the official Android documentation: <https://developer.android.com/google/play/requirements/target-sdk>. At the time of writing, the minimum acceptable API level is 30 (release name R). If you select the API release and the Next button is greyed out, then that means you need to download the API system image by clicking the download button next to the API's release name. Once Android Studio finishes downloading the API system image, the Next button should become available.

**Select a system image**

Recommended x86 Images Other Images

Release Name	API Level	ABI	Target
<b>R</b> Download	30	x86	Android 11.0 (Google Play)
Q Download	29	x86	Android 10.0 (Google Play)
Pie Download	28	x86	Android 9.0 (Google Play)
Oreo Download	27	x86	Android 8.1 (Google Play)
Oreo Download	26	x86	Android 8.0 (Google Play)
Nougat Download	25	x86	Android 7.1.1 (Google Play)
Nougat Download	24	x86	Android 7.0 (Google Play)

**R**



API Level **30**

Android **11.0**

Google Inc.

System Image **x86**

We recommend these Google Play images because this device is compatible with Google Play.

Questions on API level?  
See the [API level distribution chart](#)

**!** A system image must be selected to continue.

Previous Next Cancel Finish

In the following screen, you will get the chance to verify the virtual device configuration. The default configuration is usually fine so if you're happy then press Finish.

**Verify Configuration**

AVD Name Pixel 4 API 30

Pixel 4 5.7 1080x2280 xxhdpi Change...

R Android 11.0 x86 Change...

Startup orientation

Portrait Landscape

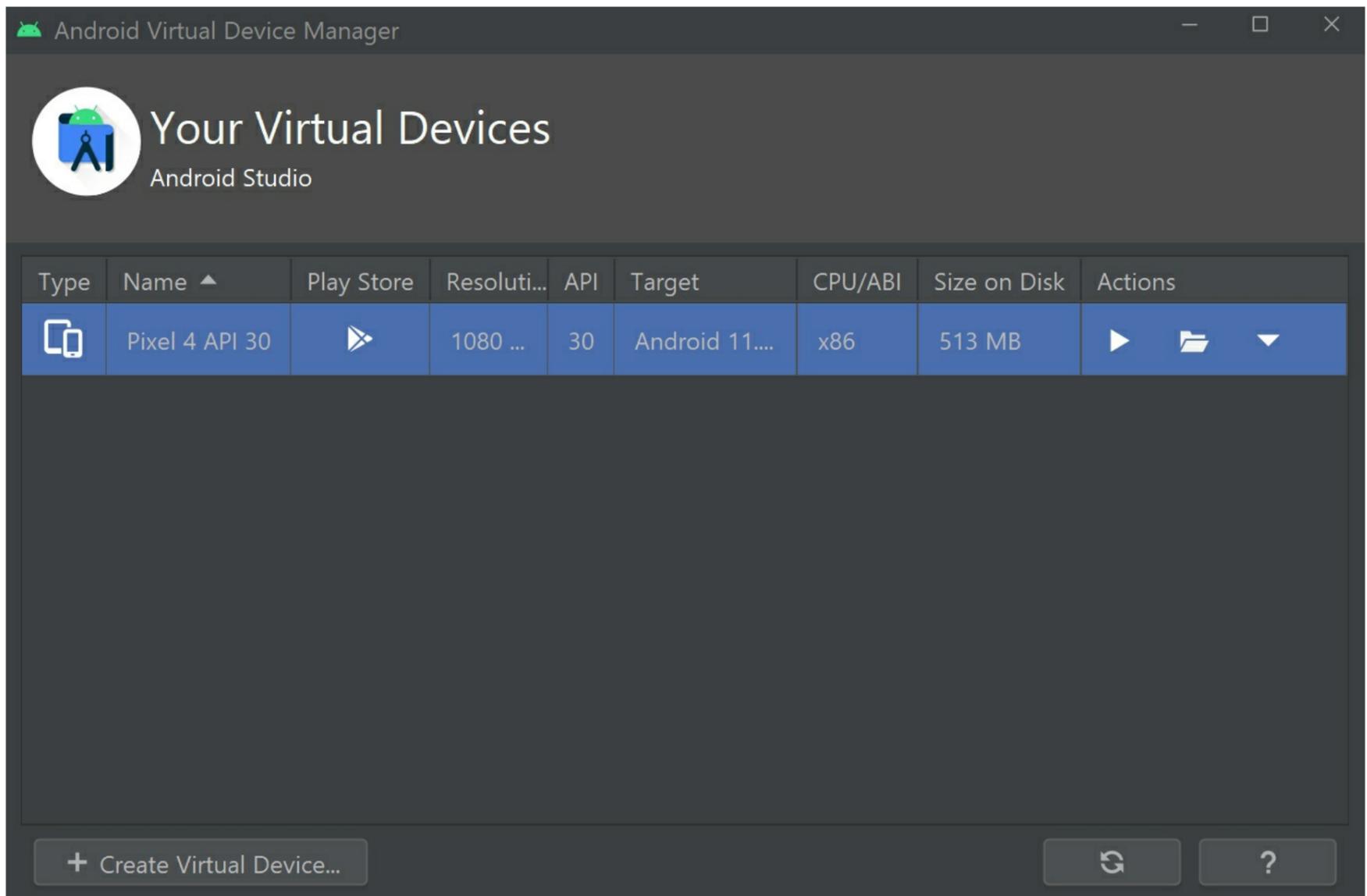
Emulated Performance Graphics: Automatic

Device Frame  Enable Device Frame

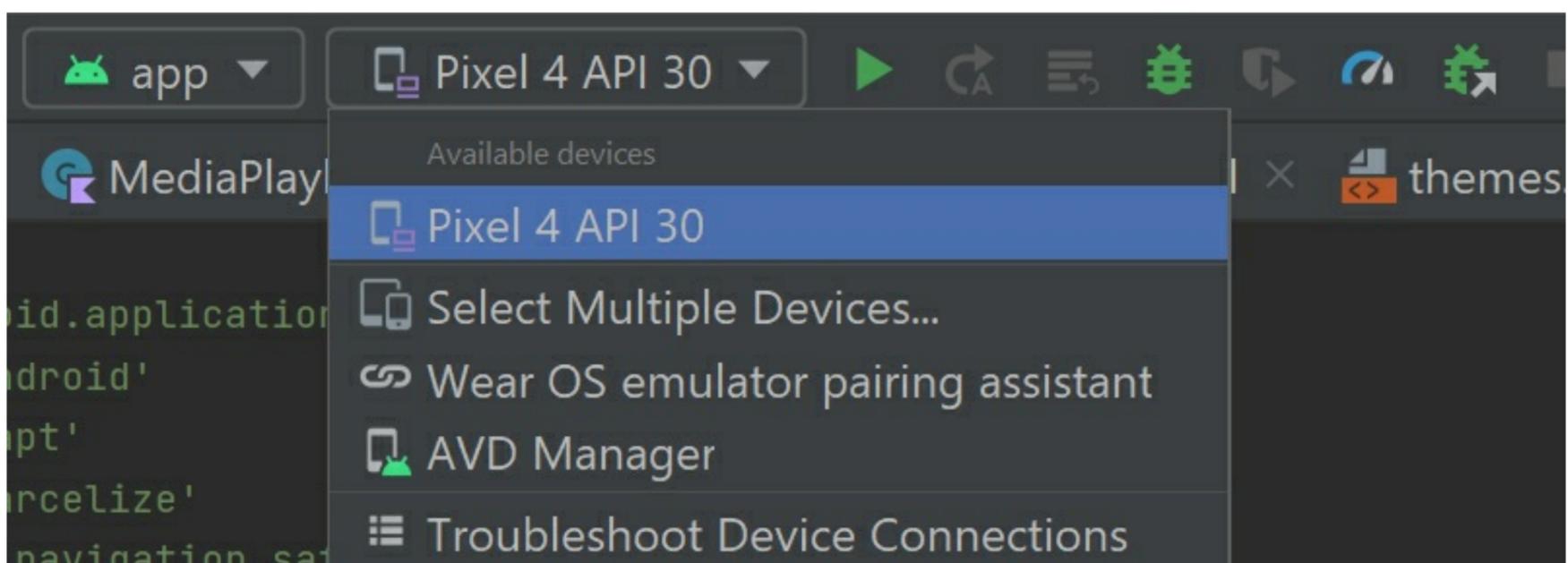
Show Advanced Settings

Previous Next Cancel **Finish**

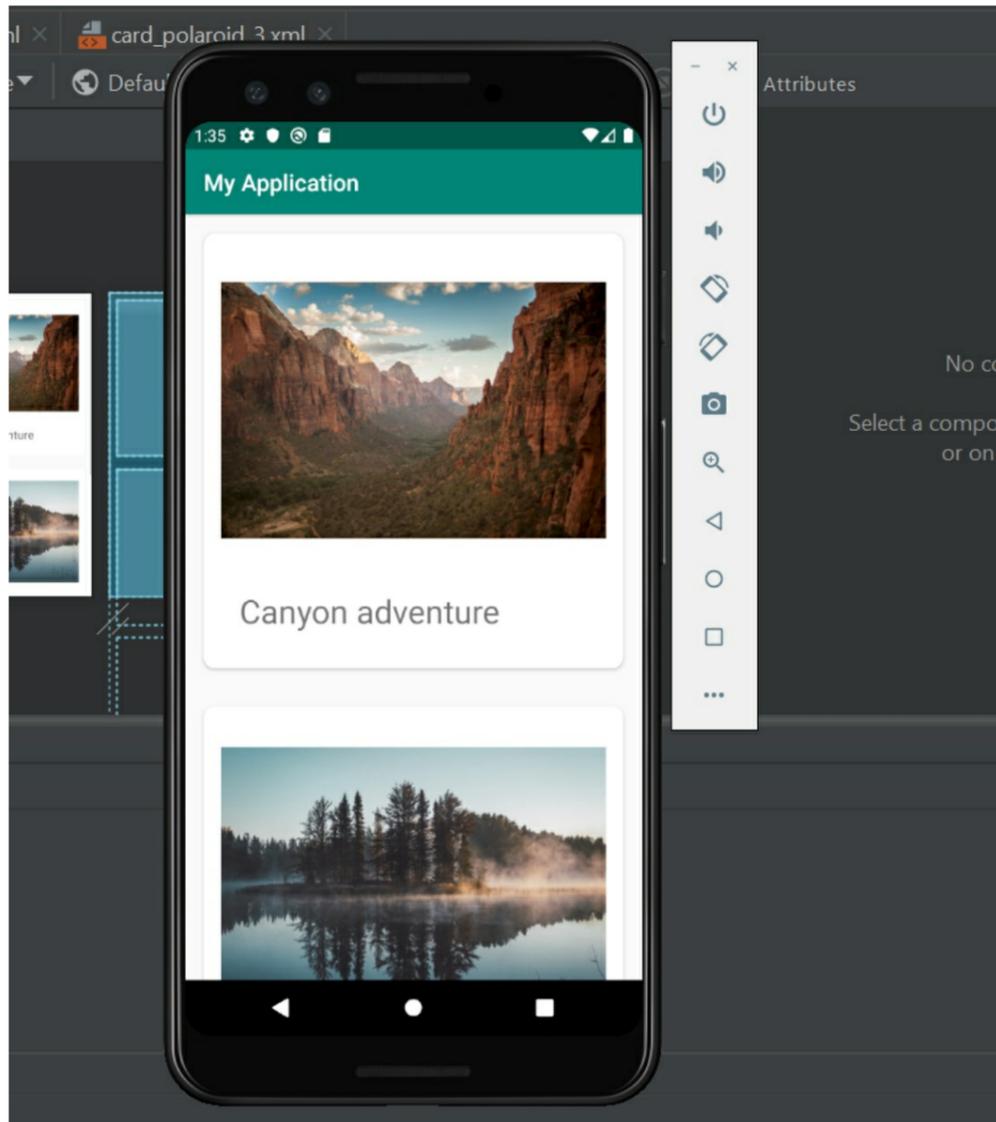
The newly configured virtual device will now be listed in the Android Virtual Device Manager!



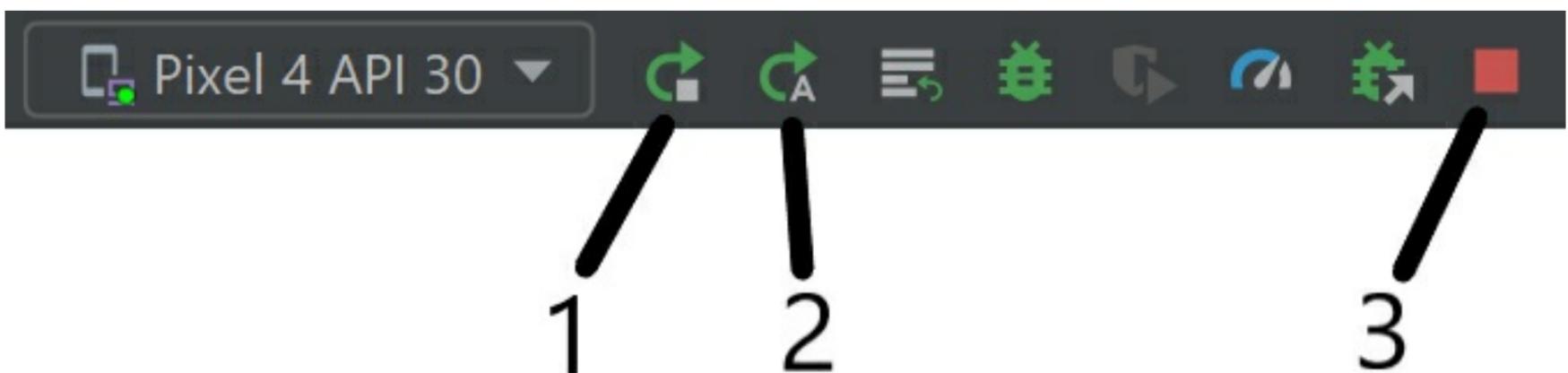
You can now close the Android Virtual Device Manager window. In the Android Studio action bar, you will notice the virtual device is listed as one of the available devices ready to be used.



To run an app, simply press the green play button next to the list of virtual devices. Android Studio will then compile the application's code and install it on the virtual device. You can even interact with the app using your mouse and keyboard!



While the app is running, you may notice some changes to the buttons in the Android Studio action bar, as shown below:



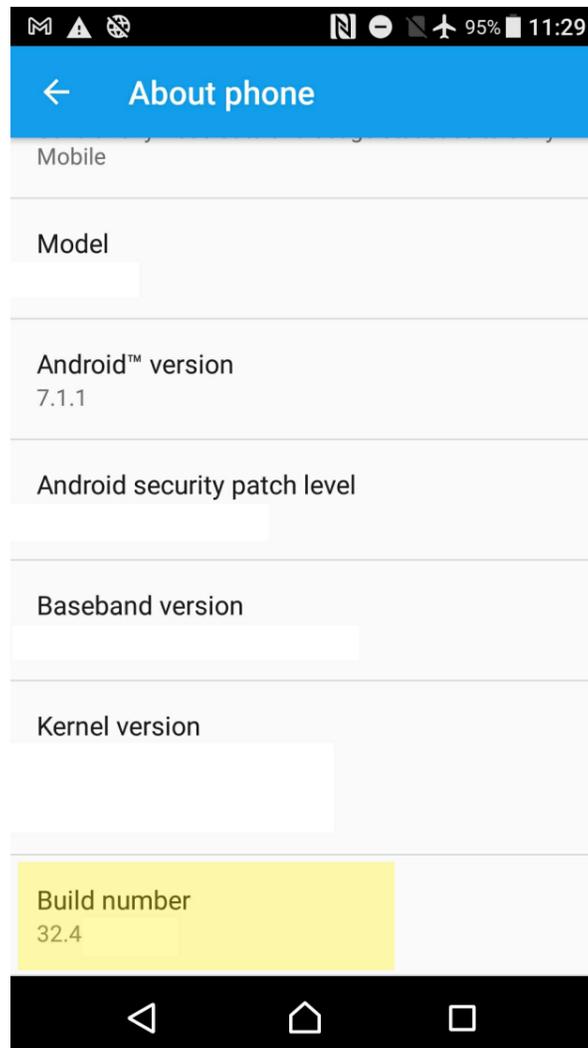
Several key buttons you may like to use at runtime are labelled in the above screenshot:

1. Restart Application - Will recompile and install the application on the device again. Use this button if you make substantial changes to your code, such as adding or deleting methods or altering the structure of a resource file.
2. Apply Changes and Restart Activity - Will restart the current activity and apply any code or resource changes. Faster than reinstalling the application. Use this button if you change the content of your code but not the structure. For example, if you alter the body of a method or change a resource property from one value to another (e.g. changing a Button widget's tint). Don't worry if you're not completely sure when to use this button. If you click this button but a full reinstall is required to apply your code changes, then Android Studio will let you know.
3. Stop Application - Will stop the application but leave the virtual device open. Use this button if you need time to work on your code but don't want the application to continue running in the background. Running an application on a virtual device can be computationally demanding, so stopping the application when it is not required can help improve your computer's performance.

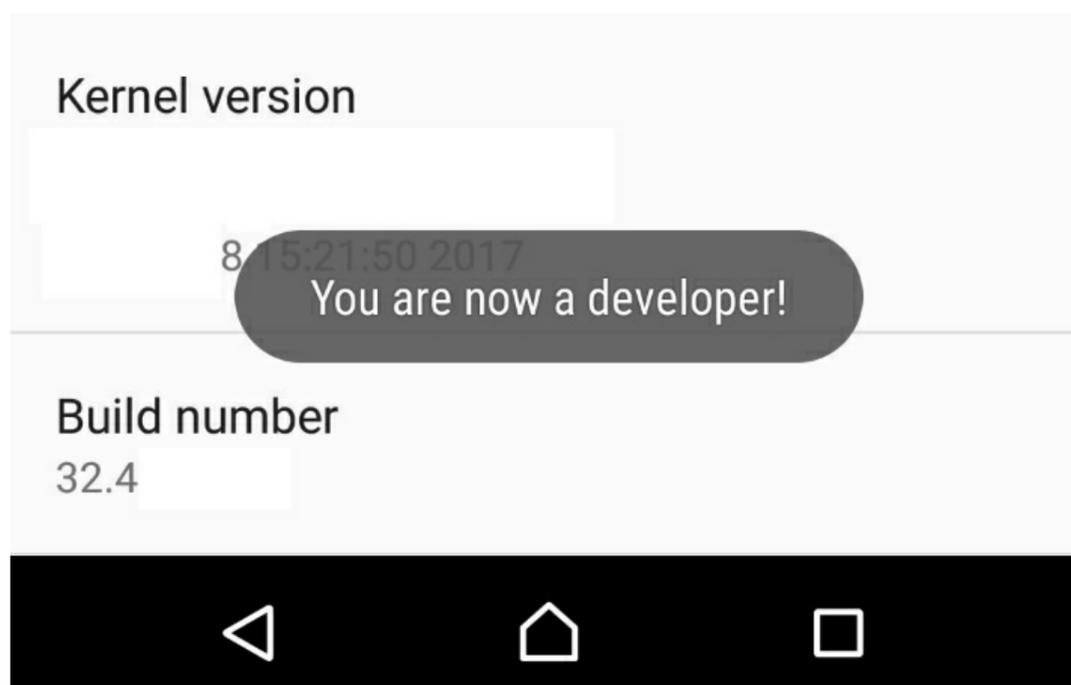
And there we have it! You can now run your app without the need for an actual phone or tablet. It is good practice to run your app regularly and check everything is working as you intended.

## Testing an application on a real device

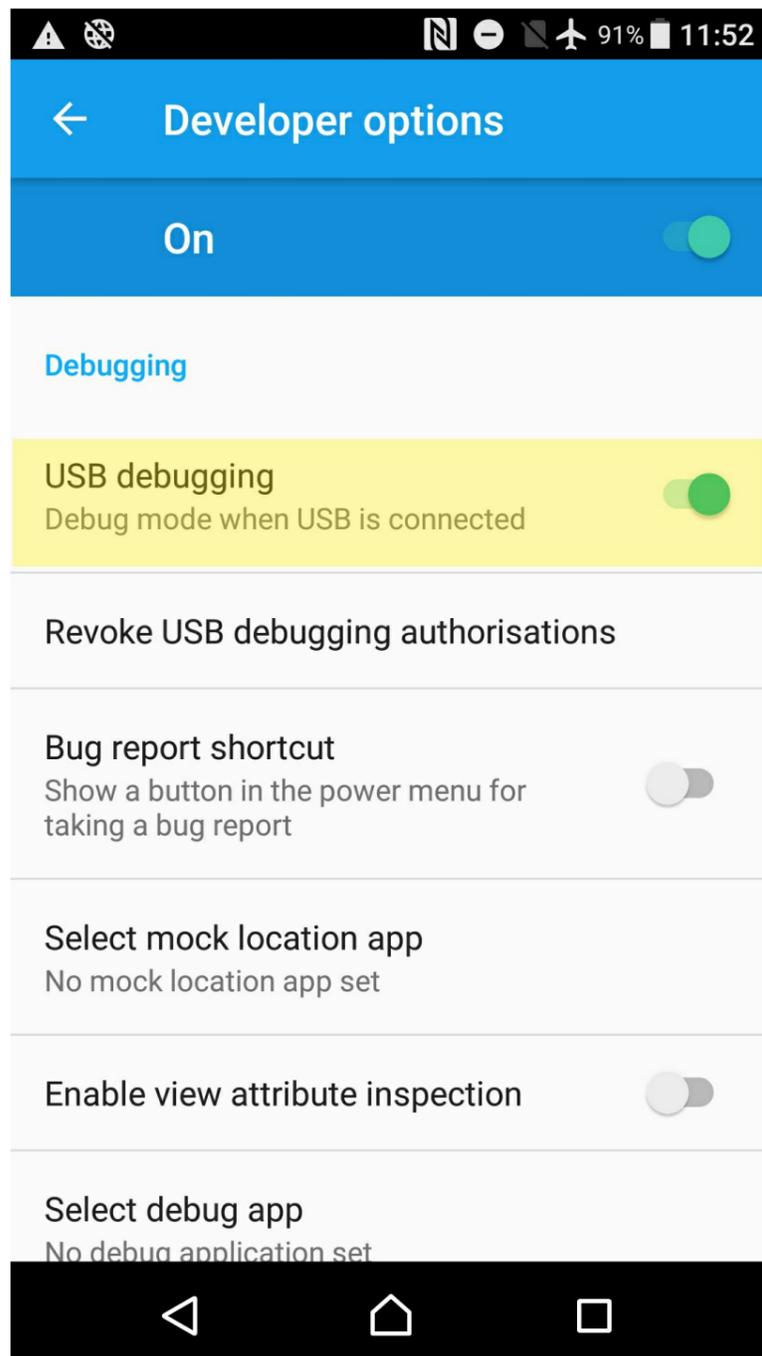
Virtual devices are a convenient method of testing your app on various models without requiring a physical device; however, it is also worthwhile installing your app on a real device so you can try your application first-hand. To install an Android Studio project on a physical device, you must first enable the developer options on that device. To enable the developer options, navigate to the About phone section in the Settings application. Next, locate the build number.



Repeatedly tap the build number until you see a toast notification that reads “You are now a developer!”.



An extra section in the Settings application called Developer options should become available. Open the developer options and ensure USB debugging is switched on. The USB debugging option allows your device to run in debug mode when connected to your computer, which is necessary to install apps via Android Studio.



The next time you plug your device into your computer you should see a prompt asking whether you would like to allow USB debugging. Press OK and Android Studio should detect your device moving forward.

## Allow USB debugging?

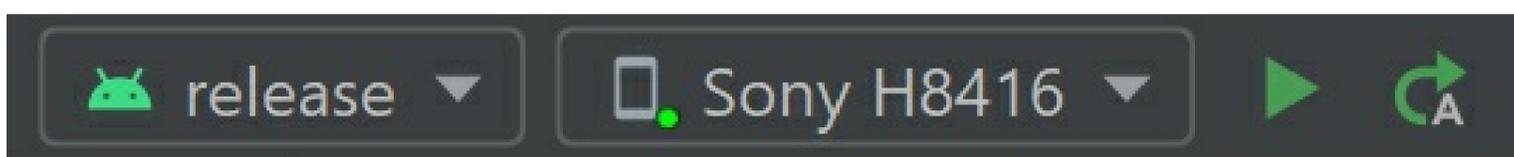
The computer's RSA key fingerprint is:

0F:43:1F:9F

Always allow from this computer

CANCEL OK

If you return to Android Studio you should see your device listed in the dropdown menu that also lists the available virtual devices. To install an app on your device, press the play symbol. You can uninstall the app later in the same way you would uninstall any other app from your device.



## Using the Logcat to track system messages

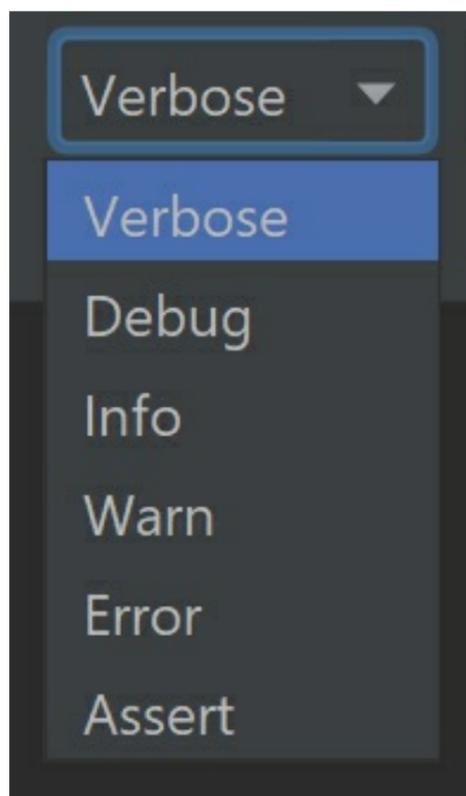
Android Studio contains a Logcat window that allows you to view system messages in real-time. If your app has been installed on a physical device, then the Logcat will also display a history of messages the app has logged since it was last connected to Android Studio. Hence, if you are testing an app and it crashes, simply plug the device back into your computer to view the log report. You will find the option to open the Logcat window in the bottom menu panel in Android Studio.



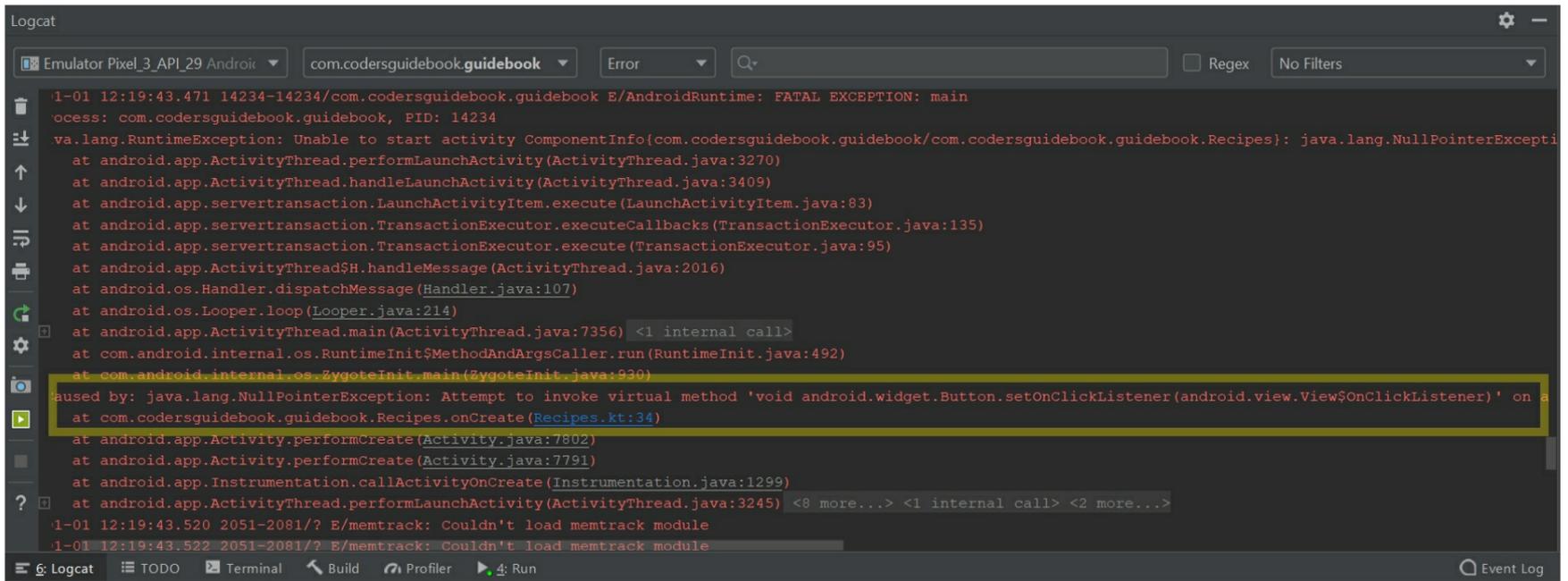
The Logcat sorts the system messages into one of the following categories:

- **Verbose:** Show all messages.
- **Debug:** Show messages relating to potential bugs and runtime processes that may be useful during development.
- **Info:** Show expected messages for regular runtime processes.
- **Warn:** Show potential issues that are not yet errors.
- **Error:** Show issues that have caused errors.
- **Assert:** Show issues that the developer expects should never happen.

The most commonly used message categories when testing applications are debug and error. To filter the Logcat output for a certain type of message, select the relevant category from the dropdown menu at the top of the Logcat window.



If an error causes the app to crash, an effective way to find the cause is to switch the message category to Error and scroll through the output to find the messages that were printed at the time of the crash. If the error is caused by a section of code, then the problematic file and line number will be underlined in blue as shown below.



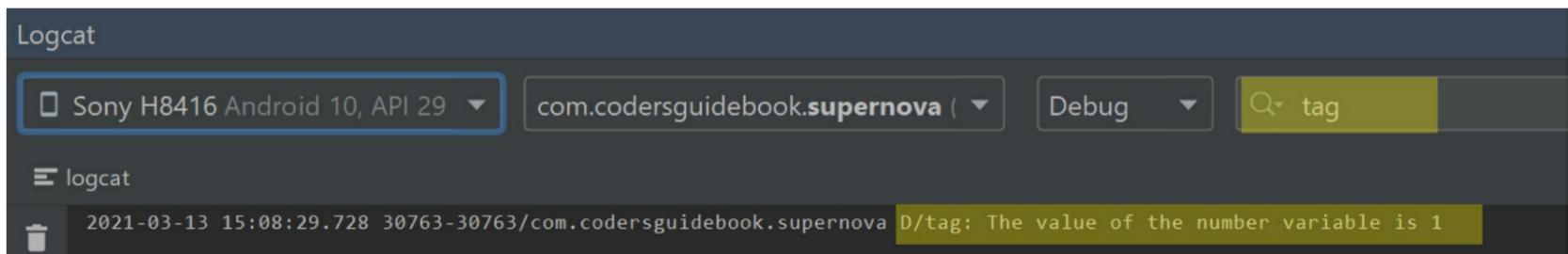
In the above example, the Logcat output tells us a `NullPointerException` caused the app to crash because of an error at line 34 in a file called **Recipes.kt**. If you click the blue underlined text, it will take you to the exact line in the **Recipes.kt** file where the error occurred. You can then review the code and see if you can find a solution. If no solution is immediately apparent, then you can always copy and paste the error description from the Logcat output into a search engine. Another developer has likely had a similar problem and found a solution.

The final topic we will cover in this section is how to manually print output to the Logcat. This is a useful practice for monitoring sections of code and runtime values. For example, the below code demonstrates how to print the contents of a variable called `number` to the Logcat:

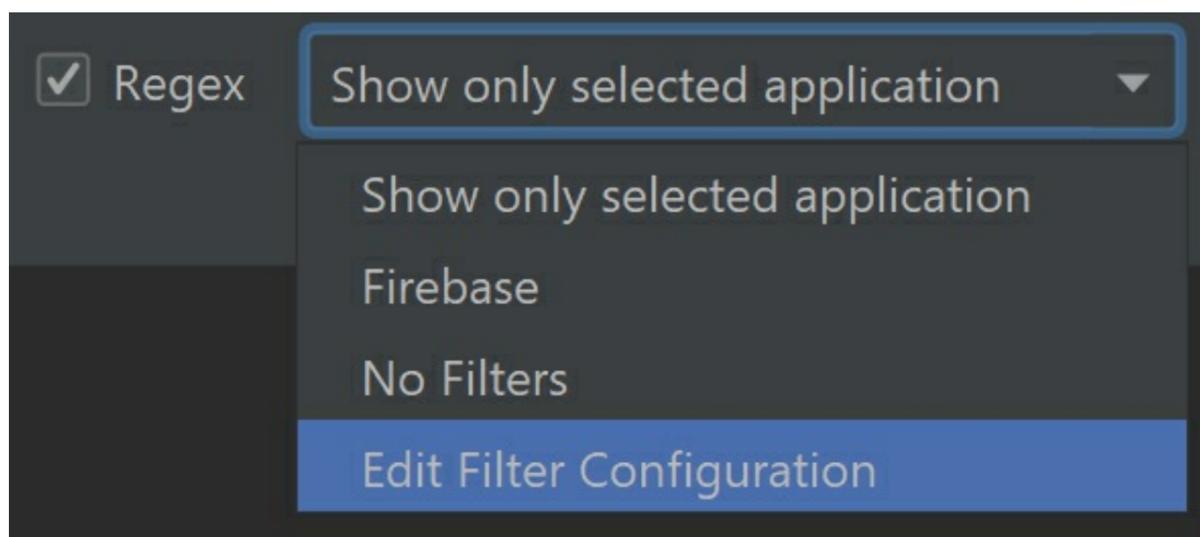
```
val number = 1
```

```
Log.d("tag", "The value of the number variable is $number")
```

To print a message to the Logcat, you call an instance of the `Log` class then insert a full stop followed by the letter for the Logcat channel you would like to print to (d for debug, e for error etc.). The log message is split into two sections: a tag that is used to filter the Logcat output and the message body. In the above example, the tag will be "tag" and the message body will be "The value of the number variable is \$number". The dollar sign indicates that you are referencing the value of a variable. The final output of the above Log message would be "The value of the number variable is 1", as shown below.



The above screenshot also indicates how to use the Logcat search bar to find all the log messages that contain the "tag" keyword. An alternative way of finding all messages that use a certain tag is to create a Logcat filter. To do this, press the Edit Filter Configuration option in the dropdown menu in the top right corner of the Logcat window.



In the Create New Logcat Filter window that opens, you can define a name for the filter, specify the tag that you would like to isolate, and set the Log Level channel you would like to filter (e.g. Debug or Error).

Create New Logcat Filter

+ -

search\_for\_tag

Filter Name: search\_for\_tag

Specify one or several filtering parameters:

Log Tag: tag  Regex

Log Message:  Regex

Package Name:  Regex

PID:

Log Level: Debug

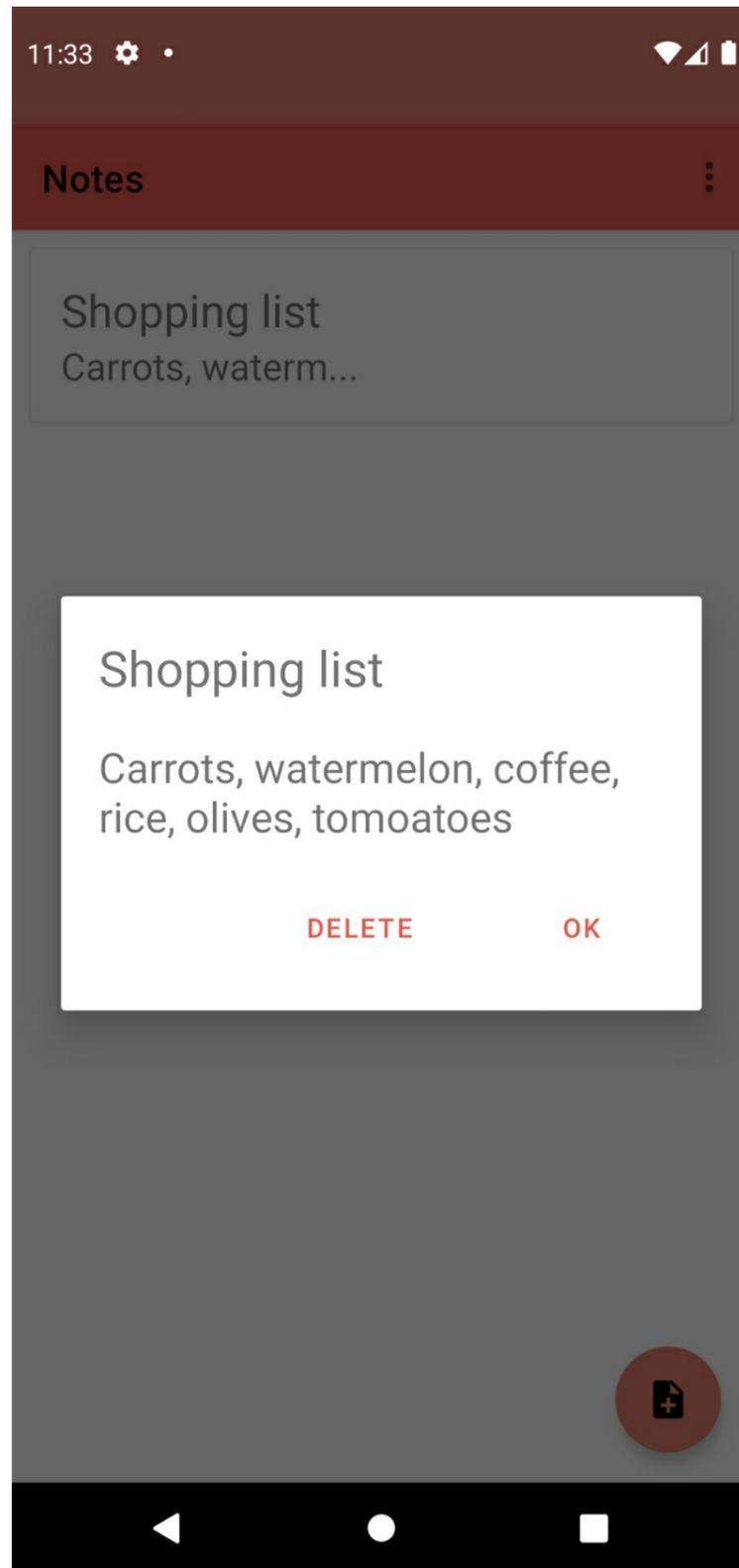
OK Cancel

Once you have filled in all of the details, press OK to save the filter.

# How to create a Notes application

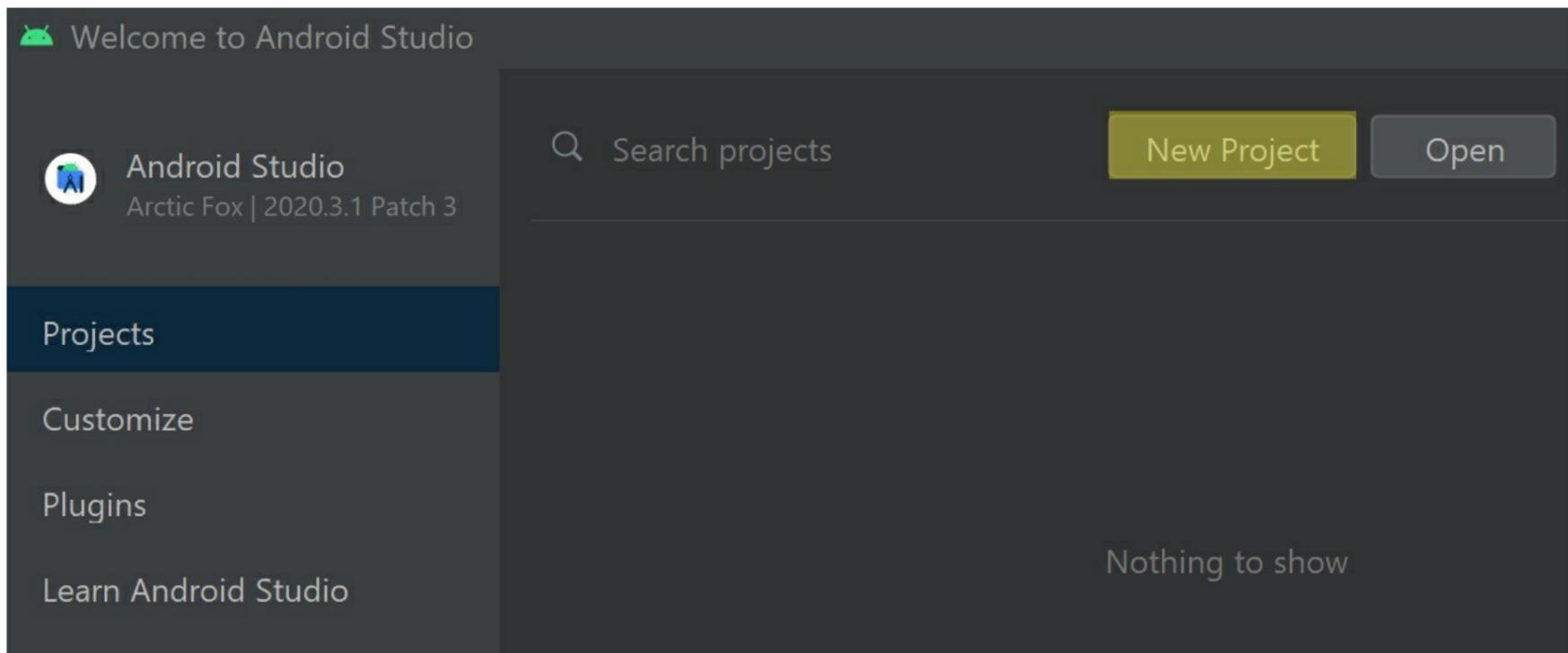
## Introduction

For our first project, we will create a Notes app that will allow the user to record their thoughts and ideas. The user will be able to save and delete notes and personalise the app by switching between light and dark colour themes and adding dividing lines between the note previews on the home screen. In creating this app, you will learn many of the fundamentals of Android programming using Kotlin and XML. These fundamentals will help prepare you for more challenging projects.

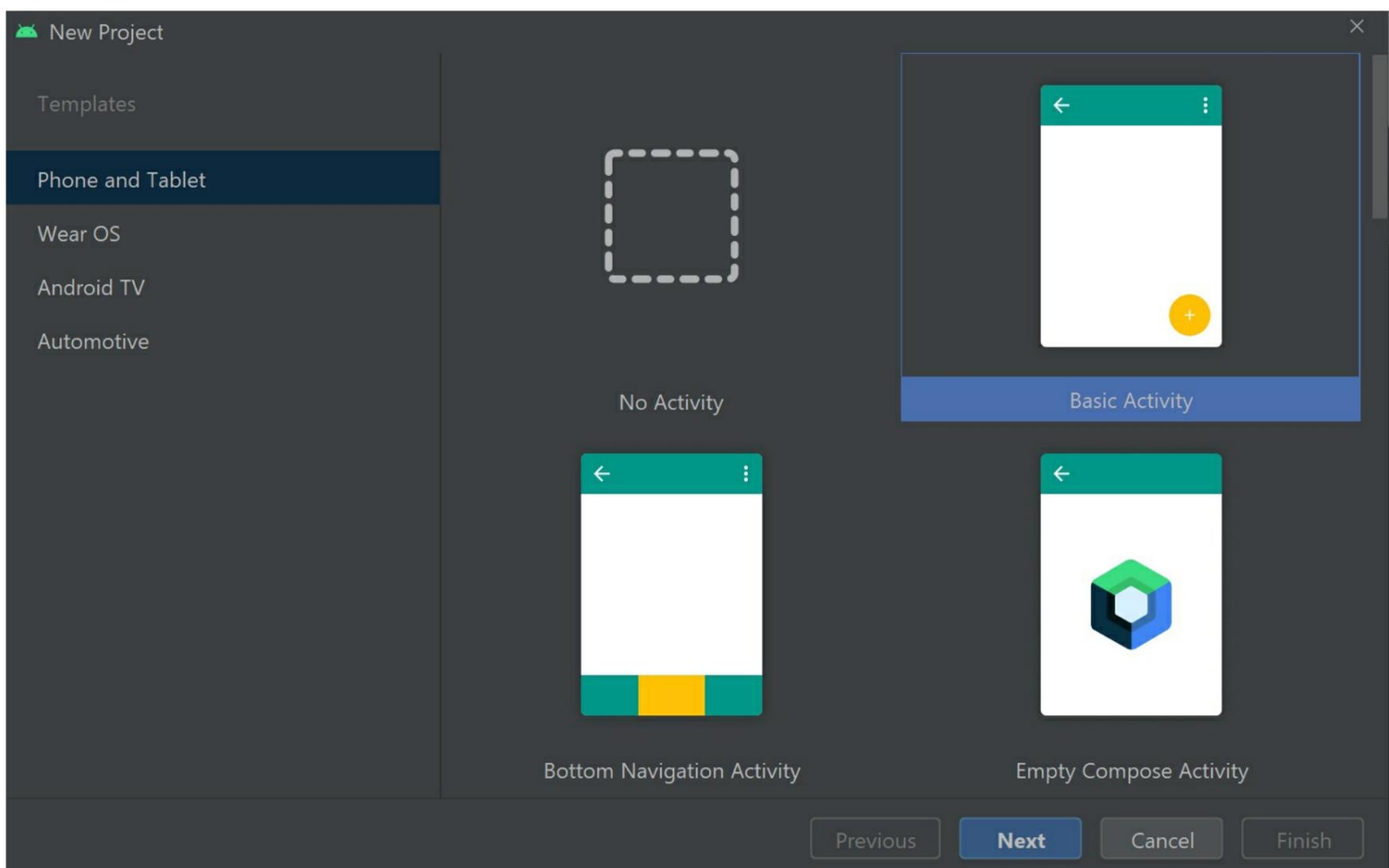


## Getting started

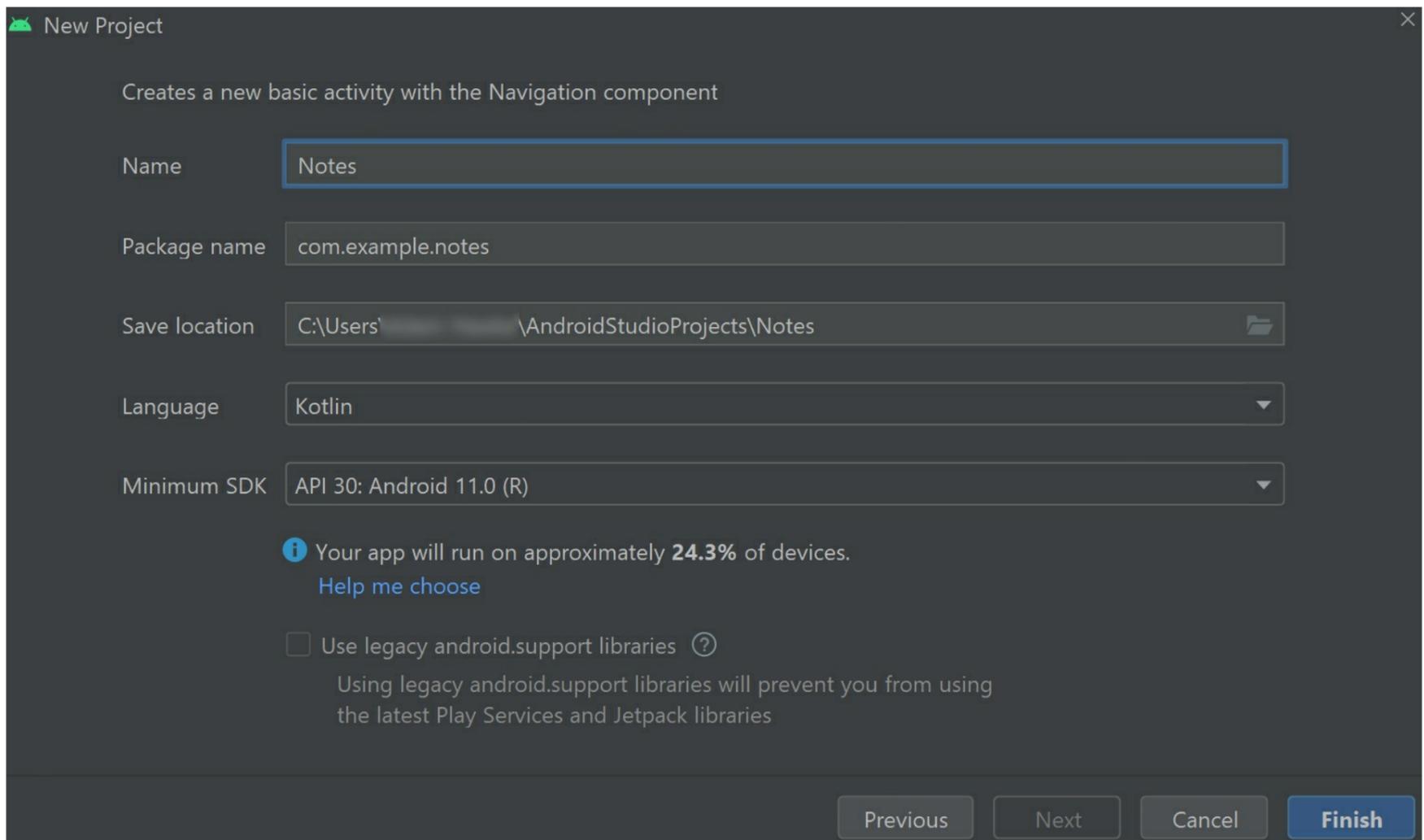
To develop a new Android app, open Android Studio and click the New Project button. Android Studio will automatically save your project as you work so you never need to worry about losing your progress. Also, in future, each time you open Android Studio it will automatically load the last project(s) you were working on so you can pick up right where you left off.



Android Studio offers several ready-made project templates to help get you started. For the Notes app, select the Basic Activity template. The Basic Activity template provides your app with an action bar and a floating action button. In this app, the action bar will allow the user to navigate to the settings page and the floating action button will enable the user to create a new note.

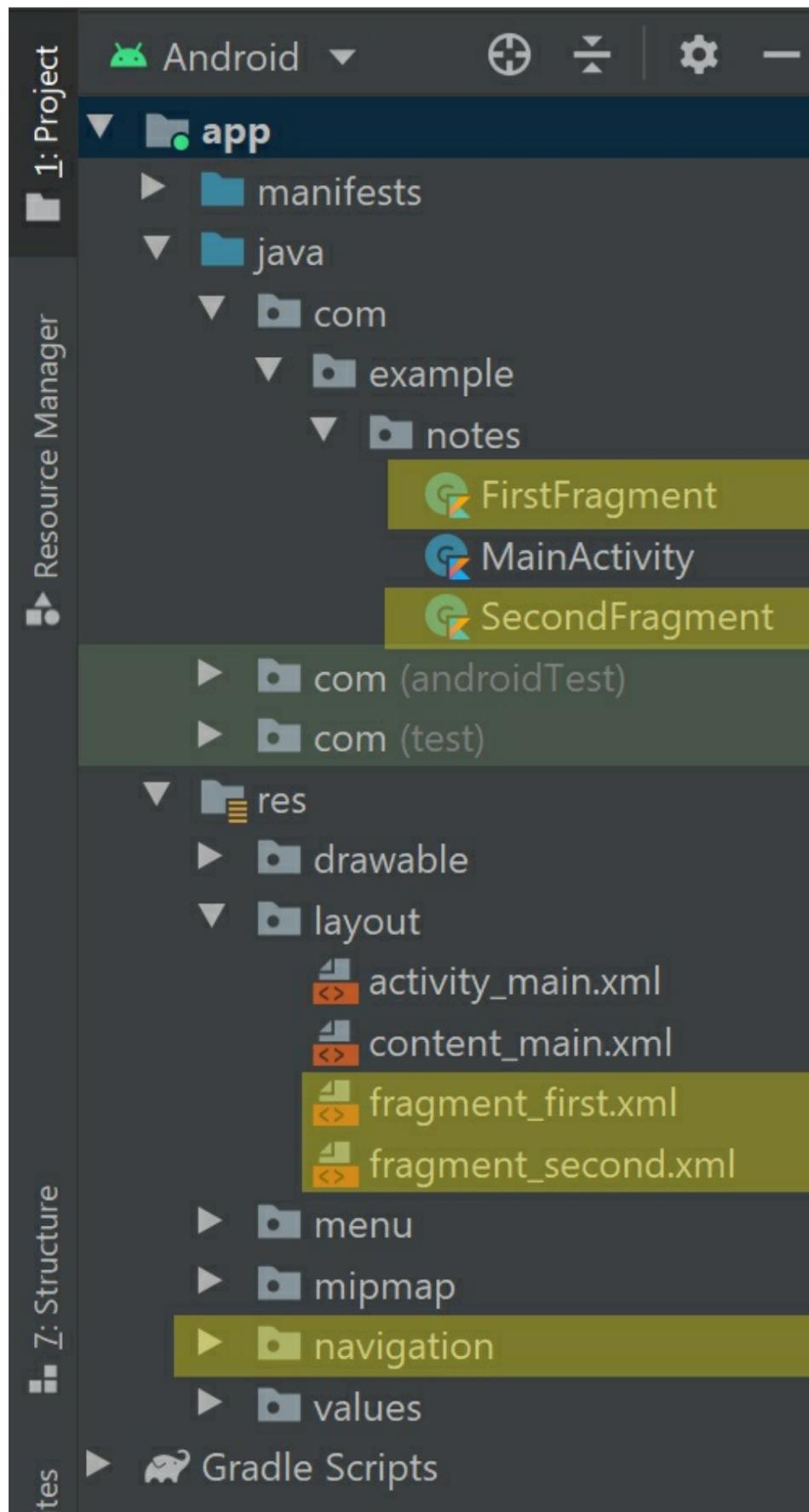


Select the Basic Activity template and press Next. You will be taken to a New Project window, which invites you to define a few details about the project. First, add a name for the project (e.g. Notes). Next, ensure the language is set to Kotlin and the API level is set to 30. Currently, all Android apps must target at least API 30 to be published in the Google Play store. You can find the latest API requirements by referring to the official Android documentation: <https://developer.android.com/distribute/best-practices/develop/target-sdk>. Once all the above information has been entered, press Finish. Android Studio will then set to work and create a project based on the selected template.

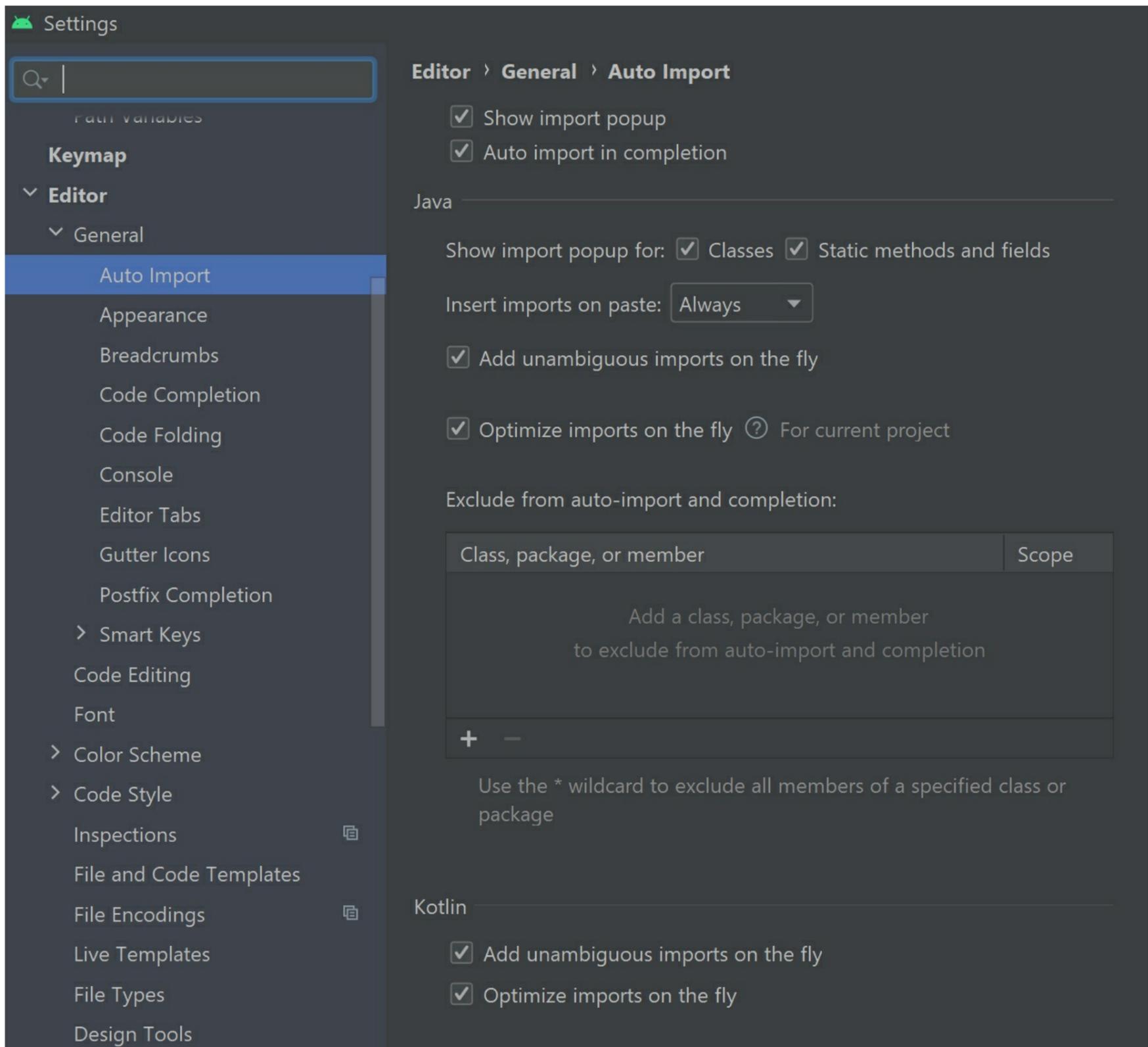


Projects created using the Basic Activity template will automatically contain two fragments and a navigation graph. We will not use fragments or navigation graphs in the Notes app so you can delete the following files and folders if you wish:

- **FirstFragment.kt** (Project > app > java > folder with the name of the project)
- **SecondFragment.kt** (Project > app > java > folder with the name of the project)
- **fragment\_first.xml** (Project > app > res > layout)
- **fragment\_second.xml** (Project > app > res > layout)
- **navigation directory** (Project > app > res)



The Kotlin code files that power the Notes app will often require external resources. Typically, external resources are integrated manually by adding 'import ...' statements to the top of the Kotlin file; however, Android Studio offers a handy alternative called Auto Import, which will attempt to generate the requisite import statements automatically while you code. To enable the Auto Import feature, navigate through **File > Settings**. In the Settings window, navigate through **Editor > General > Auto Import** then select 'Add unambiguous imports on the fly' and 'Optimise imports on the fly' for both Java and Kotlin then press Apply and OK.

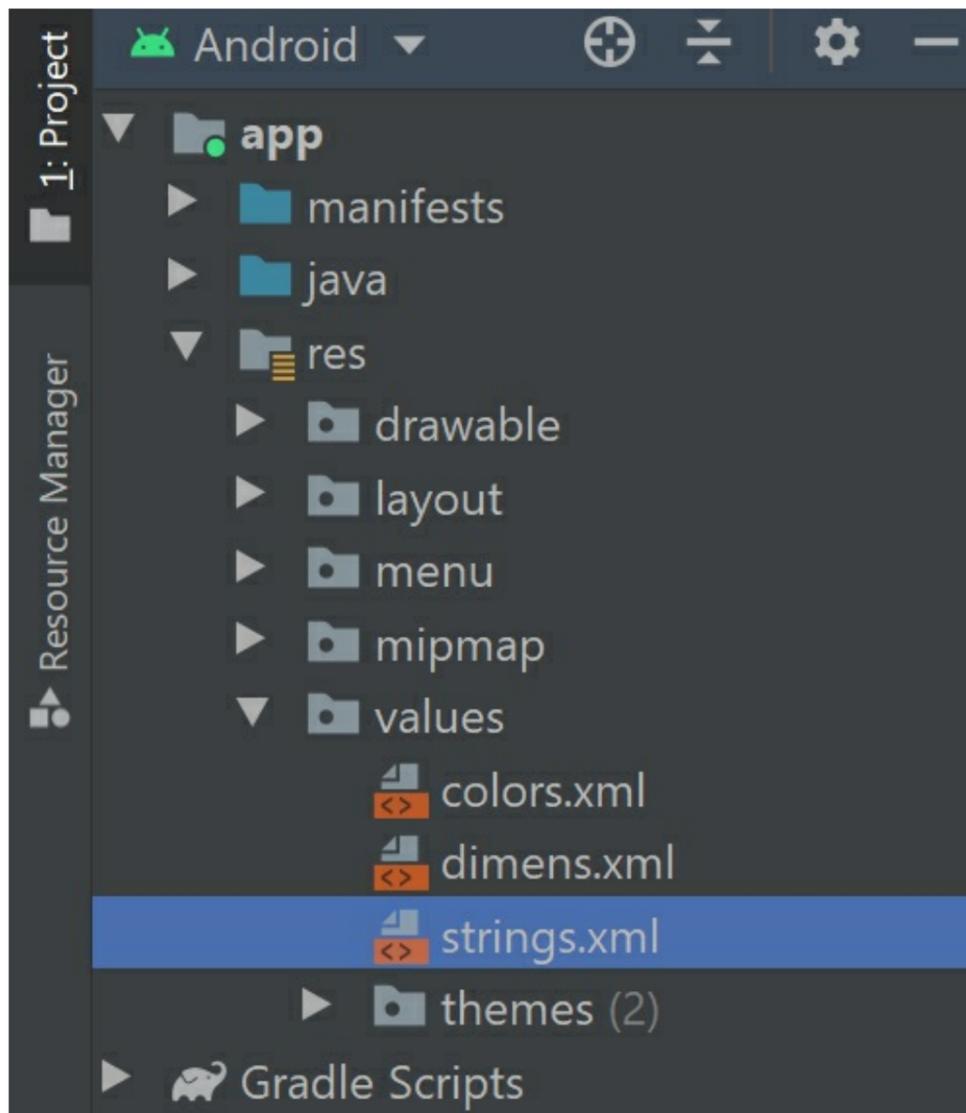


Android Studio should now add most of the necessary import statements automatically. Sometimes there are multiple potential import statements to choose from and the Auto Import feature may not work. In these instances, the requisite import statement(s) will be specified explicitly. You can also refer to the example code that accompanies this book to find the finished files including all import statements.

## Defining the string resources used in the app

Each item of text that the app will display should be stored as a string resource. A single string resource can be used across multiple locations in the app. String resources make it easier to update text because you only need to edit one resource and all areas that use the text will automatically reflect the change. Also, string resources help the app support multiple languages because you can define translations for each string.

When you create a new project, Android Studio will automatically generate a **strings.xml** resource file to store your strings. To locate the **strings.xml** file, navigate through **Project > app > res > values**.



To define all the necessary strings for the Notes app, edit the **strings.xml** file so it reads as follows:

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

  <string name="title">Title</string>
  <string name="contents">Contents</string>
  <string name="cancel">Cancel</string>
  <string name="ok">OK</string>
  <string name="delete">Delete</string>

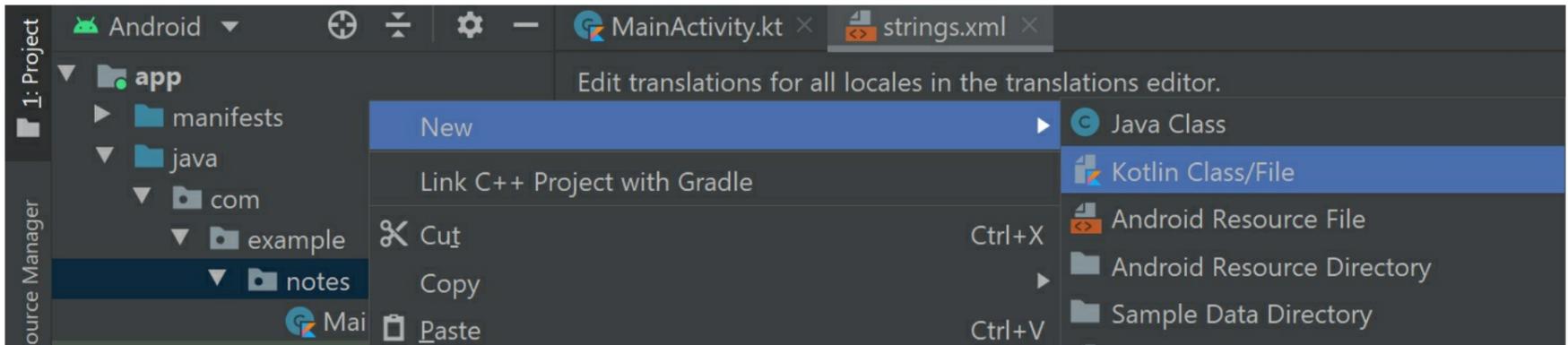
  <string name="select_theme">Switch to night theme?</string>
  <string name="add_dividers">Add dividing lines between notes?</string>

  <string name="add_new_note">Add a new note...</string>
  <string name="note_empty">Check the title and contents fields are not empty.</string>
  <string name="note_saved">Note saved!</string>
  <string name="note_deleted">Note deleted!</string>
</resources>
```

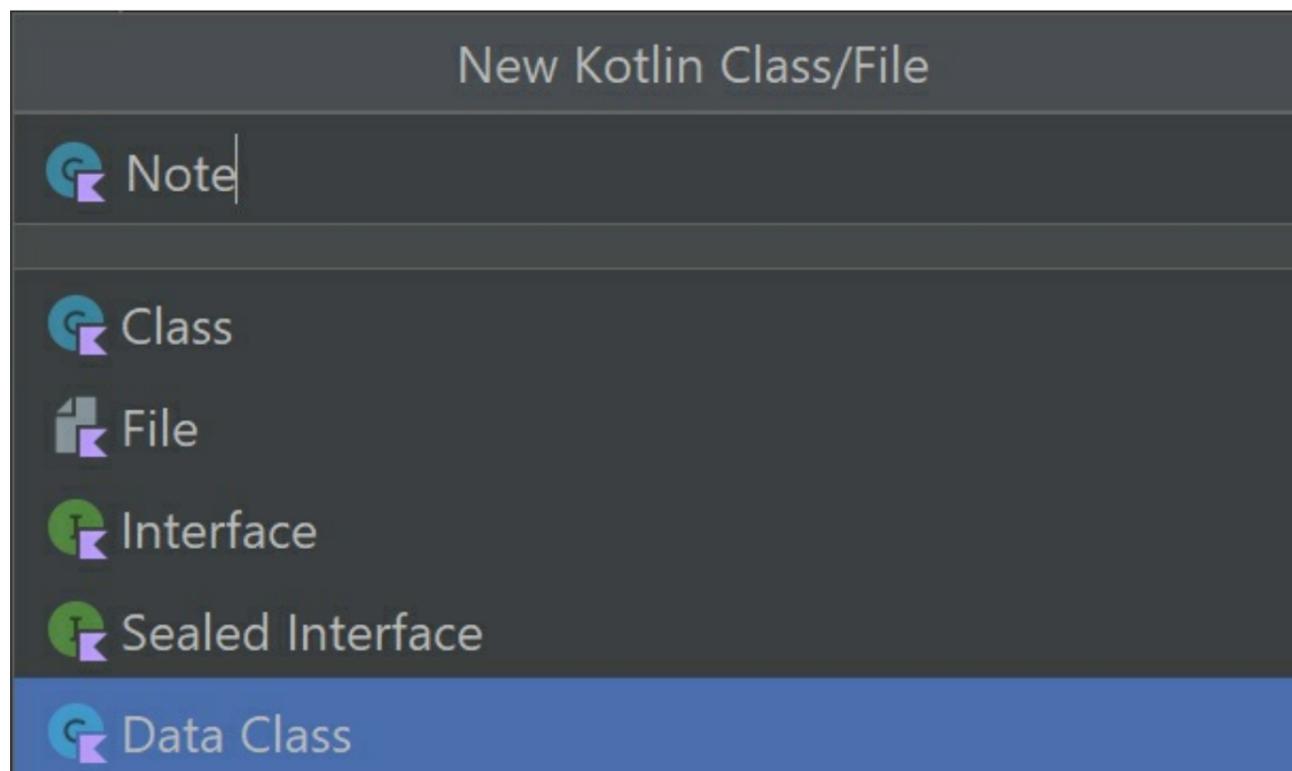
Each string resource contains a name attribute. The name attribute is what you will use to reference the string elsewhere in the app. The text that will be displayed to the user is input between the opening `<string name="">` and closing `</string>` tags.

## Creating the Note data class

The details of each note the user writes will be packaged in a Kotlin class. The class will act as a template and contain fields for the different bits of information such as the note's title and contents. An instance of a class is called an object, and a new object will be created for each note the user saves. To create a new Kotlin class, navigate through **Project > app > java** then right-click the folder with the name of the project. Next, select **New > Kotlin Class/File**



Name the file Note and select Data Class from the list of options.



A file called **Note.kt** should then open in the editor. To define the Note data class, modify the file's code so it reads as follows:

```
data class Note(
    val title: String,
    val contents: String)
```

There are a couple of things to note about the above code. First, we label the class as a data class. Data classes are designed to store information and feature in-built functions to help manage data (see below for more details). The primary constructor of the data class must contain at least one variable. Each variable stores a piece of information. In this case, the title variable will store the name of the note and the contents variable will store the body of the note. Both variables will store data in String format. If the data type declaration has a question mark at the end (e.g. String?) then this means the value of the variable can be null. A variable with a null value is empty or valueless; however, the variable can be assigned a value later if it is initialised using the var keyword. The var keyword means the value of a variable can be changed, while the val keyword means the value of a variable is fixed and can only be set once.

The variables in the Note data class are initialised using the val keyword and the String data type declaration does not feature a question mark. This means the values of each variable may only be set once and cannot be null.

To create an instance of a data class, you must define a value for each field in the primary constructor. The data class instance (also referred to as an object) can be stored in a variable and accessed elsewhere in your code. The example below creates an instance of the Note class with a title value of "To-do list" and contents value of "Cut the grass and go to the shop.". The Note object is stored in a variable called newNote:

```
val newNote = Note("To-do list", "Cut the grass and go to the shop.")
```

Data classes feature a couple of extra in-built functions. For example, you can convert the contents of a data class object to a string using the toString method:

```
newNote.toString()
```

The above code would output the following: **Note(title=To-do list, contents=Cut the grass**

### and go to the shop)

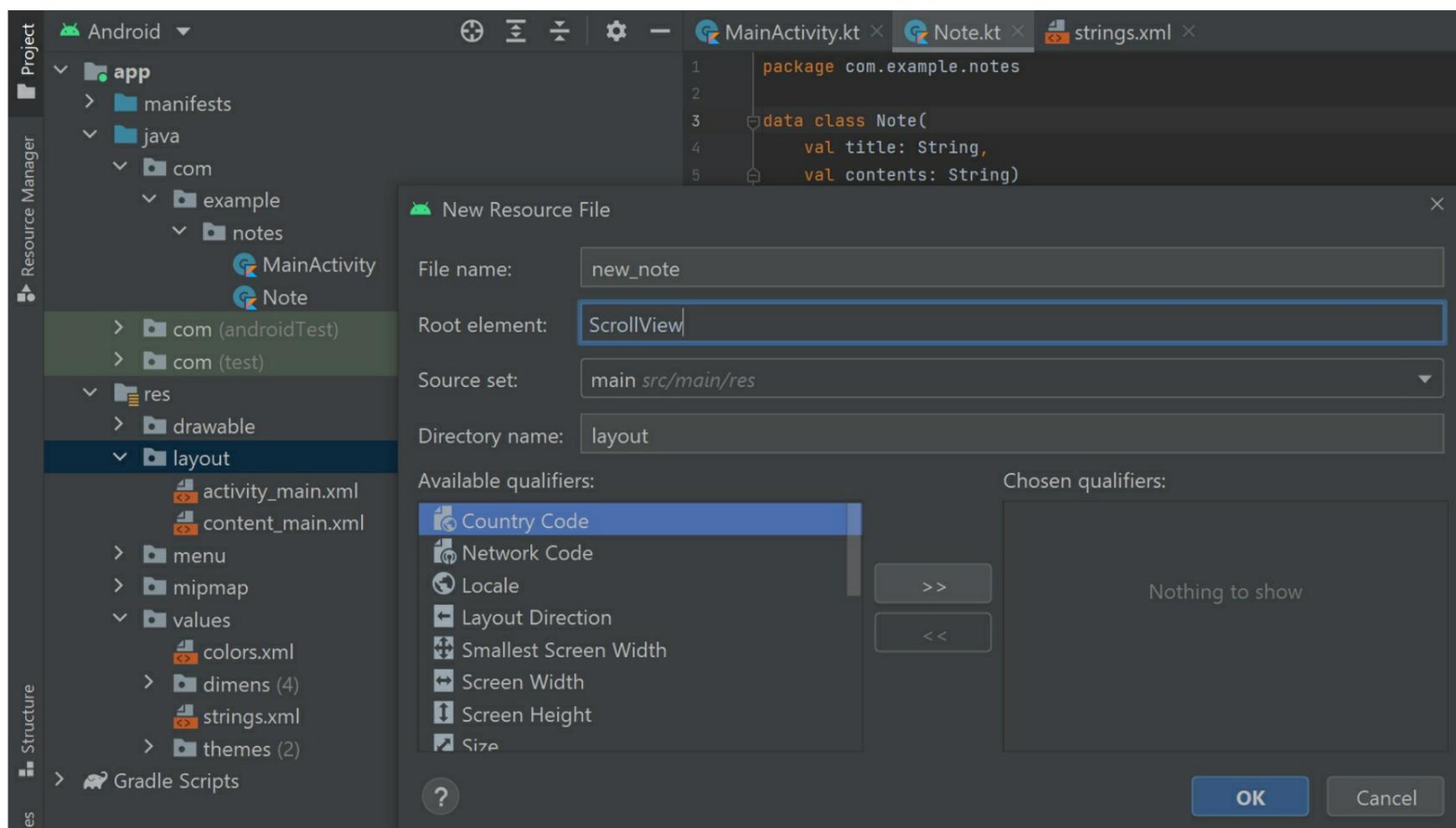
You can also copy data class objects and change their values. For example, imagine we wanted to change the contents of the note to show the to-do list is complete:

```
val updatedNote = newNote.copy(contents = "The to-do list is complete!")
```

The Note object stored in the updatedNote variable would read as follows if it was converted to a string: **Note(title=To-do list, contents=The to-do list is complete!)**

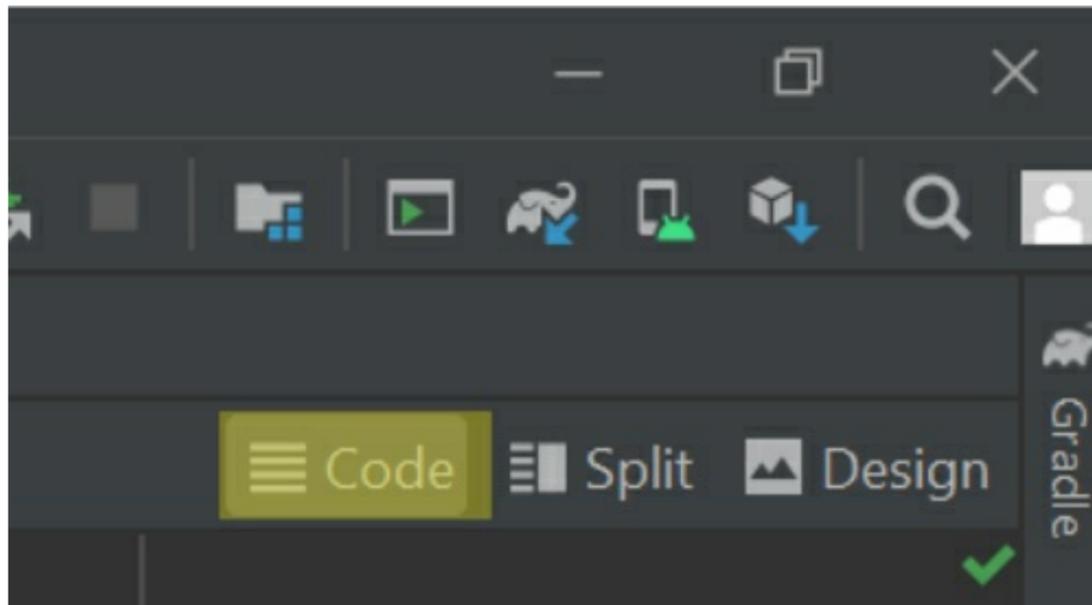
## The layout for writing new notes

The app's user-facing interfaces are defined in layout resource files. Layout resource files coordinate the widgets that display content and handle interactions. For the Notes app, the first layout we create will allow the user to write new notes. To create a layout file, locate and right-click the **layout** directory by navigating through **Project > app > res**. Next, select **New > Layout resource file**. Name the file `new_note` and set the root element to `ScrollView`. `ScrollView`-based layouts allow the user to scroll if the layout's content is too large to fit on the screen. The scroll feature may come in handy if the user writes an especially long note.



A file called **new\_note.xml** should then open in the editor. The first elements we will add to the layout are two `EditText` widgets. `EditText` widgets allow the user to input text. In this case, the widgets will store the title of the note and its contents, respectively. At the bottom of the layout, we will add two button widgets: one to save the note and the other to dismiss the window without saving the note.

It is possible to add widgets to a layout by dragging and dropping them from the Palette; however, it is often quicker to input the XML code for the widgets into the layout directly. To view and edit the code of the **new\_note.xml** layout file, click the Code view button as shown below.



Alternative layout views include Design view, which shows how the layout will appear to the user, and Split view, which integrates both Code view and Design view simultaneously. Split view is handy for seeing how code changes affect the layout in real-time.

To configure the **new\_note.xml** layout file, open the layout in Code view and add the following code between the opening and closing ScrollView tags:

```
<LinearLayout  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:layout_margin="20dp"  
    android:orientation="vertical">
```

```
<EditText  
    android:id="@+id/editTitle"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:inputType="textPersonName"  
    android:hint="@string/title"  
    android:importantForAutofill="no" />
```

```
<EditText  
    android:id="@+id/editContents"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:layout_marginVertical="20dp"  
    android:hint="@string/contents"  
    android:inputType="text"  
    android:importantForAutofill="no" />
```

```
<TableRow  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:gravity="end" >
```

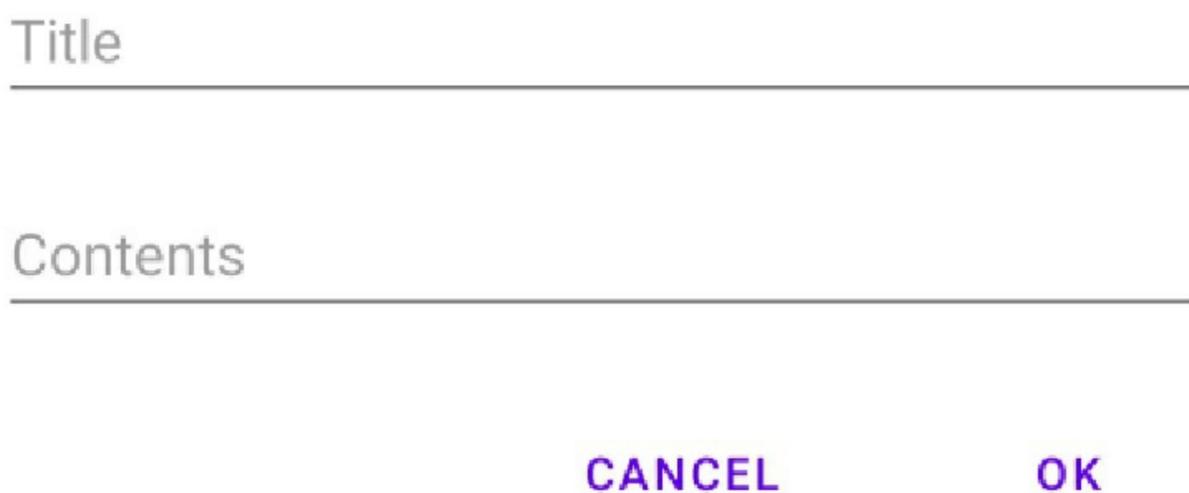
```
<Button  
    android:id="@+id/btnCancel"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="@string/cancel"  
    style="?android:attr/buttonBarButtonStyle" />
```

```
<Button  
    android:id="@+id/btnOK"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginStart="30dp"  
    android:text="@string/ok"  
    style="?android:attr/buttonBarButtonStyle" />
```

```
</TableRow>
</LinearLayout>
```

In the above code, we use a `LinearLayout` widget to align the `EditText` and `Button` widgets vertically. The first `EditText` widget will allow the user to enter the title of the note. The `inputType` attribute for the widget is set to `"textPersonName"`, which restricts the input text to a single line. Meanwhile, the second `EditText` widget will allow the user to enter the contents of the note. The `inputType` attribute for the second `EditText` widget is set to `text`, which unlike the `textPersonName` type has no restrictions on the length of the contents. Both `EditText` widgets feature a `hint` attribute. The `hint` attribute defines a placeholder message that will be visible until the user begins typing. The `hint` attribute indicates what kind of information the user should enter. In this case, the hints are "Title" and "Contents", respectively. The text for the hints is sourced from the string resources we defined earlier in the `strings.xml` file.

At the end of the layout, there are two button widgets: a cancel button that will close the layout and discard the note, and an OK button that will save the note. The code which enables this functionality will be defined later. Both buttons feature a `style` attribute set to `'?android:attr/buttonBarButtonStyle'`, which refers to a ready-made button style provided by Android. The button style will help the buttons to look and behave in a way that is consistent with other apps you may have used. For example, the buttons will have a transparent background and display a ripple effect when pressed. Both buttons are enclosed within a `TableRow` widget to ensure they appear side-by-side in the layout.



The image shows a visual representation of the layout described in the text. It consists of two text input fields, one labeled 'Title' and one labeled 'Contents', each with a horizontal underline. Below these fields are two buttons: 'CANCEL' and 'OK', both in purple text.

## The layout for displaying notes

The app will allow the user to read their saved notes. To view a note, the user simply needs to click a note preview on the app homepage and a dialog window will display the full note. The dialog window will require a new layout resource file. Locate and right-click the **layout** folder (found by navigating through **Project** > **app** > **res**) then select **new** > **Layout resource file**. Name the file `show_note` and set the root element to `ScrollView`. The layout will contain two `TextView` widgets for displaying the title and contents of the note, respectively. It will also feature two buttons that allow the user to delete or close the note. To create the widgets, open the `show_note.xml` file in Code view and add the following code between the opening and closing `ScrollView` tags:

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="20dp"
    android:orientation="vertical">

    <TextView
        android:id="@+id/txtTitle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="26sp" />

    <TextView
        android:id="@+id/txtContents"
        android:layout_width="wrap_content"
```

```
android:layout_height="wrap_content"
android:layout_marginVertical="20dp"
android:textSize="22sp" />
```

```
<TableRow
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="end" >
```

```
<Button
    android:id="@+id/btnDelete"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/delete"
    style="?android:attr/buttonBarButtonStyle" />
```

```
<Button
    android:id="@+id/btnOK"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="30dp"
    android:text="@string/ok"
    style="?android:attr/buttonBarButtonStyle" />
```

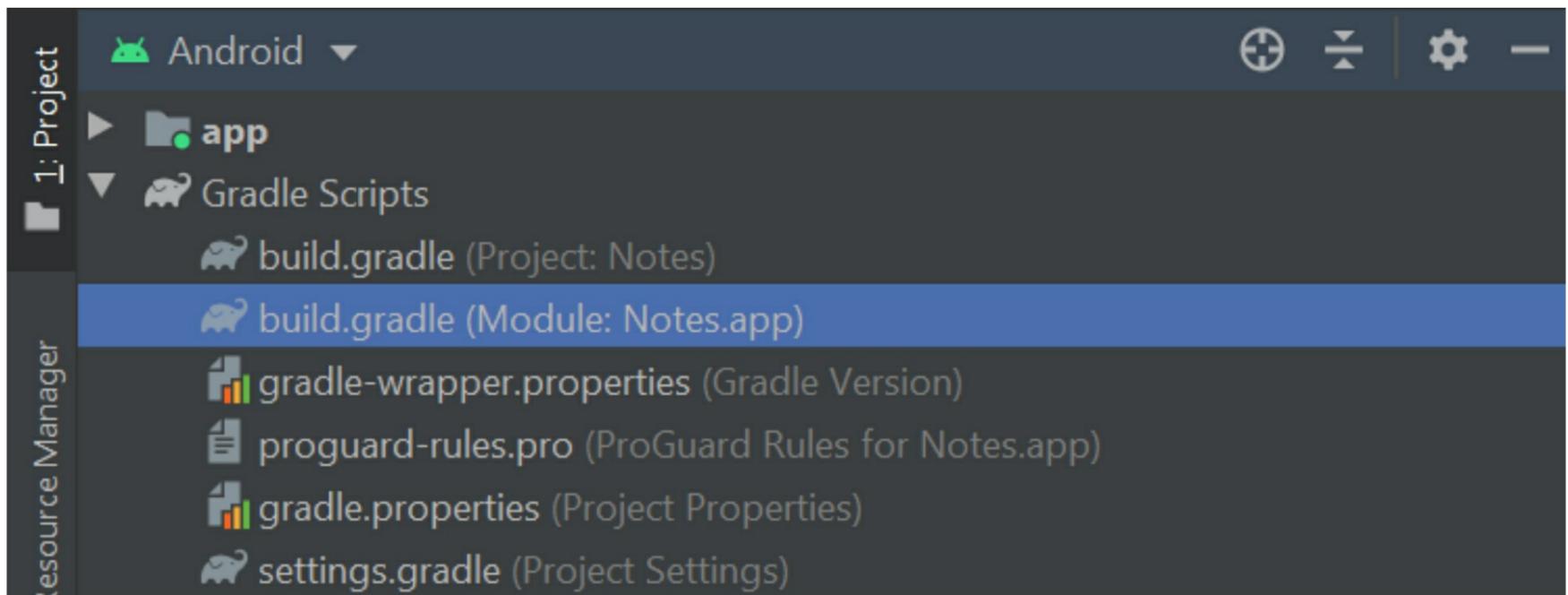
```
</TableRow>
</LinearLayout>
```

The above code is similar to the code we added to the **new\_note.xml** layout but with a few small differences. First, the above layout uses TextView widgets to display the title and contents of the note rather than EditText widgets. Unlike EditText widgets, TextView widgets are read-only and their contents cannot be modified by the user. Text will be programmatically loaded into the TextView widgets based on the opened note. At the bottom of the layout, there is a Delete button that will delete the note and an OK button that will dismiss the dialog window. Like before, the buttons are contained within a TableRow so they appear side-by-side.

We have now created the layouts which allow the user to create and view notes; however, the layouts are not yet operational. To make the layouts functional we need to create two Kotlin class files. The classes will handle user interactions and respond to requests to save and view notes.

## Enabling View Binding

Since we have created a couple of layout files, it is worth discussing a concept called view binding. View binding is a method for allowing Kotlin classes to interact with the contents of a layout. Once view binding is enabled, a bespoke binding class will be generated for each layout resource in the project. The binding class provides access to the layout's constituent widgets. In newer versions of Android Studio, most project templates have view binding automatically enabled. To verify that view binding is enabled, open the Module level **build.gradle** file by navigating through **Project > Gradle Scripts**.



Locate the android element and find the following code:

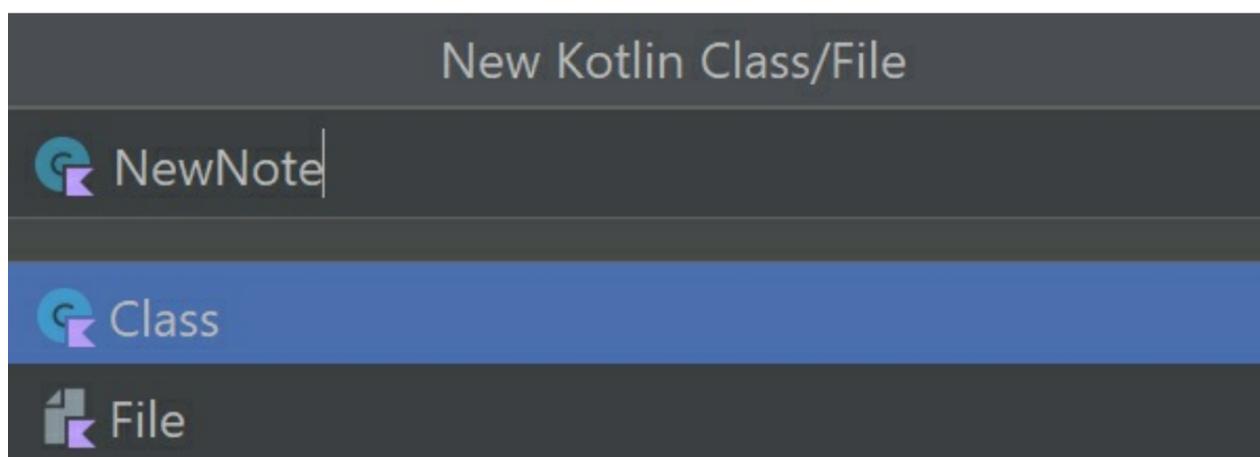
```
buildFeatures {  
    viewBinding true  
}
```

The above code enables view binding. We'll discuss the practicalities of view binding further in upcoming sections. If the above code is not present, then you will need to add it manually. In which case, don't forget to re-sync your project when prompted. The project must be synchronised whenever the gradle files are modified to make sure all the necessary external tools and modules are imported and ready.

Gradle files have changed since last project sync. A project sync may be necessary for the IDE ...[Sync Now](#)

## The NewNote dialog window

To make the `new_note.xml` layout operational, we must create a Kotlin class that saves the user's input as a note. To create a new Kotlin class, navigate through **Project** > **app** > **java** then right-click the folder with your project name. Select **New** > **Kotlin File/Class**, name the file `NewNote` and select Class from the list of options.



Once the `NewNote.kt` file opens in the editor, modify its code so it reads as follows:

```
import androidx.appcompat.app.AlertDialog  
import androidx.fragment.app.DialogFragment  
  
class NewNote : DialogFragment() {  
  
    private var _binding: NewNoteBinding? = null  
    private val binding get() = _binding!!  
  
    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {  
        val callingActivity = activity as MainActivity  
        val inflater = callingActivity.layoutInflater  
        _binding = NewNoteBinding.inflate(inflater)  
  
        val builder = AlertDialog.Builder(callingActivity)  
            .setView(binding.root)  
            .setMessage(resources.getString(R.string.add_new_note))  
  
        // TODO: Configure the buttons here  
  
        return builder.create()  
    }  
  
    override fun onDestroyView() {  
        super.onDestroyView()  
        _binding = null  
    }  
}
```

In the above code, the `NewNote` class extends the `DialogFragment` class. When one class extends another, it imports all the data from that class. Importing the data from another class is a principle of object-oriented programming called inheritance, and a feature of programming languages like Kotlin. In this instance, we use inheritance to grant the `NewNote` class access to the variables and methods from the `DialogFragment` class. The `NewNote` class will use the inherited data to float on top of the rest of the app as a pop-up window.

In your code files, you can introduce comments to prevent lines of code from being executed. For example, you might like to prevent a problematic block of code from being run or leave a message for other developers who read your code. Some people use comments to explain what a method or a block of code does.

In Kotlin, you can comment out a single line by preceding the line of code with two forward slashes //, or comment out multiple lines by enclosing the lines in /\* and \*/.

```
// A single-line comment in Kotlin
```

```
/* A multiple line
comment in Kotlin.
fun commentedOutFunction(): String {
    return "I'm commented out"
}
*/
```

In XML (the language we use to write our resource files), you can comment out a single line or multiple lines by enclosing the line(s) in <!-- and -->

```
<!-- I'm a comment in an XML file -->
```

During the development process, you can also use comments to remind yourself to complete a task or fix an issue. For example, you can begin the comment message with the phrase TODO, if the comment refers to a task that you will complete later, or FIXME, if the comment refers to an issue that you need to fix. As you progress through the sections in this book, you will often see TODO comments in the code excerpts as we build the projects in stages.

```
// TODO: Sort the list alphabetically
```

```
// FIXME: The method cannot handle null values
```

At any point during the development process, you can view all the TODO and FIXME comments throughout your project by opening the TODO tab at the bottom of the Android Studio window.

The contents of the pop-up window will be sourced from the **new\_note.xml** layout via the layout's binding class. To access the binding class for a given layout, type each word from the layout name without spaces and punctuation then add Binding to the end. For example, in the above code, the binding class for the **new\_note.xml** layout is called **NewNoteBinding**. The binding class is accessed via two binding variables: `_binding`, which will initialise the binding class, and `binding`, which will provide access to the layout's contents. Two variables are used to avoid having to implement null checks when interacting with the binding class. For example, the `_binding` variable could be null and unusable, while the `binding` variable is explicitly declared as non-null. As long as you only refer to the `binding` variable once the `_binding` variable has been initialised, then you can be confident that the layout is accessible.

To generate the dialog window, the root element of the **new\_note.xml** layout (the `ScrollView` widget) is loaded into an instance of the `AlertDialog` class, which is a native Android class. When building the `AlertDialog` instance, we also add the message "Add a new note...", which will appear as a title above the rest of the content in the dialog window. The dialog window will feature a Cancel button and an OK button, as specified in the `new_note` layout. To make the buttons operational, replace the TODO comment in the `onCreateDialog` method with the following code:

```
binding.btnCancel.setOnClickListener {
    dismiss()
}
```

```
binding.btnOK.setOnClickListener {
    val title = binding.editTitle.text.toString()
    val contents = binding.editContents.text.toString()

    if (title.isNotEmpty() && contents.isNotEmpty()) {
        val note = Note(title, contents)
        callingActivity.createNewNote(note)
    }
}
```

```
Toast.makeText(callingActivity, resources.getString(R.string.note_saved), Toast.LENGTH_SHORT).show()
```

```
dismiss()
```

```
} else Toast.makeText(callingActivity, resources.getString(R.string.note_empty), Toast.LENGTH_LONG).show()
```

The above code accesses the buttons via the `new_note` layout's binding class. To respond to clicks by the user, we must assign each button an `onClick` listener. For the Cancel button, we simply instruct its `onClick` listener to dismiss the dialog window without saving any of the text input by the user. Meanwhile, the OK button will save the note before dismissing the dialog window. To save the note, the `onClick` listener retrieves the text that the user input into the `editTitle` and `editContents` `EditText` widgets, packages the text in a `Note` object and stores the object in a variable called `newNote`. The `newNote` variable is sent to a method named `createNewNote`, which will be located in a Kotlin file called **MainActivity.kt**. The `createNewNote` method does not yet exist so it will be highlighted in red. Once defined, however, the `createNewNote` method will save the `Note` object inside the app.

In the Kotlin code throughout this book, you may notice expressions such as the following:

```
if (title.isNotEmpty() && contents.isNotEmpty()) {
```

The `if` expression runs a block of code if the equation written inside the brackets is true. For example, in the above code, the `if` block will run if the values of the `title` and `contents` variables are not empty. Furthermore, you can also add an `else` block, which will run if the equation written inside the brackets is false.

```
if (title.isNotEmpty() && contents.isNotEmpty()) {
```

```
    // I will run if the equation is true
```

```
} else {
```

```
    // I will run if the equation is false
```

```
}
```

If you need to evaluate multiple scenarios then you could write a `when` block instead of an `if` expression. A `when` block will iterate through each scenario until it finds a true scenario. For example, the `when` block below evaluates the value of `x` and runs various methods based on whether `x` is equal to 1, 2 or 3. If `x` is not equal to any of those options then the `else` block will run.

```
when (x) {
```

```
    1 -> methodOne()
```

```
    2 -> methodTwo()
```

```
    3 -> methodThree()
```

```
    else -> methodElse()
```

```
}
```

In the comparative equations used in `if` and `when` blocks, you may notice the use of operators such as the `||` symbol when comparing two parameters. The `||` symbol is called a logical OR operator. It returns a value of 'true' if at least one side of the equation is correct. In other words the expression `x == 1 || x == 3` will return 'true' if `x` is equal to 3 (because the right side of the equation is correct) and 'false' if `x` is equal to 2 (because neither side of the equation is correct).

Alternative operators include:

**AND operator (&&)** - Will return 'true' only if both sides of the equation are correct.

```
// will return TRUE
```

```
5 < 6 && 7 < 8
```

```
// will return FALSE
```

```
5 < 6 && 8 < 7
```

**EQUALITY operator (==)** - Will return 'true' only if both sides of the equation are equal.

```
// will return TRUE
3 + 6 == 9

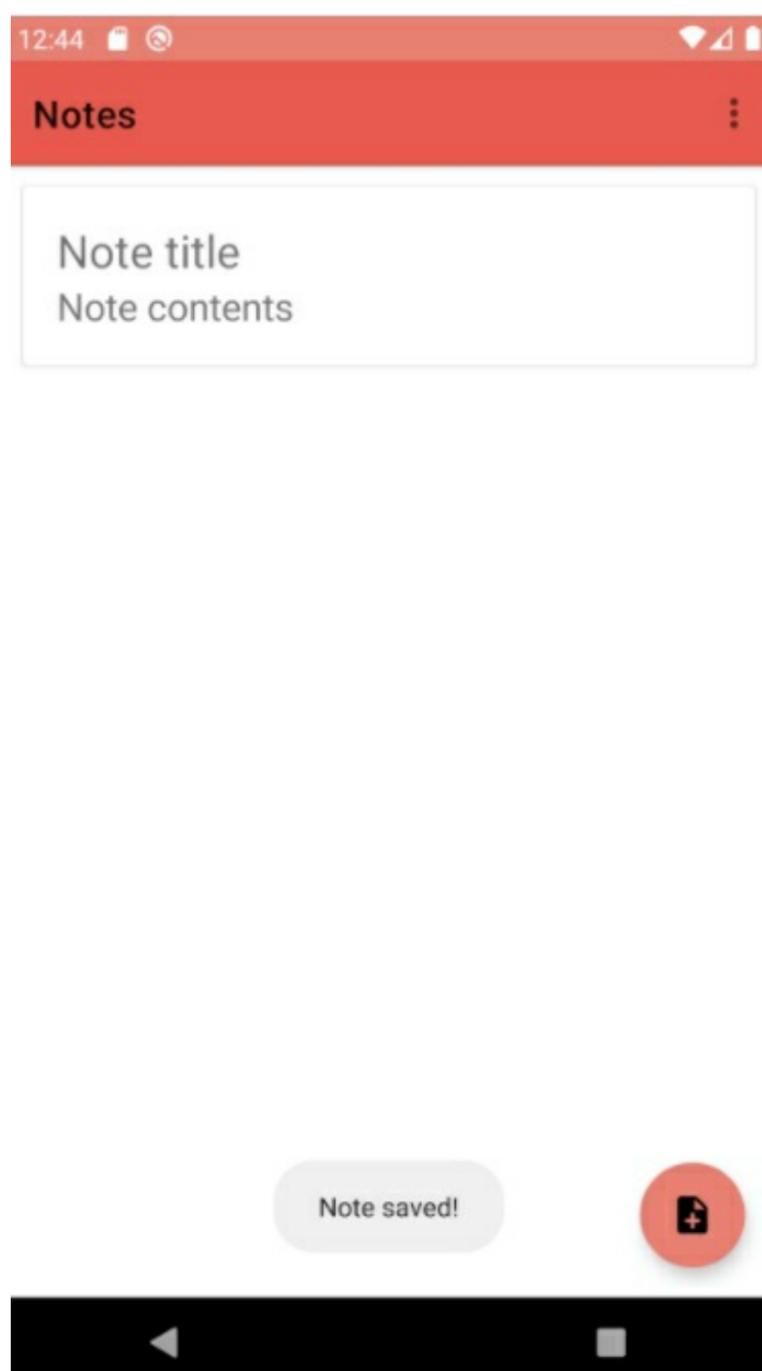
// will return FALSE
3 + 6 == 10

INEQUALITY operator (!=) - Will return 'true' only if both sides of the equation are not equal.

// will return TRUE
3 + 6 != 10

// will return FALSE
3 + 6 != 9
```

The OK button code also checks whether the contents of the EditText widgets are empty. If either widget is empty then the note will not be saved. Instead, a toast notification will inform the user that not all the required fields have been filled in. On the other hand, if the note is successfully saved, then the dialog window will be dismissed and a toast notification will confirm the note has been saved, as shown below.



The final part of the NewNote class is a method called onDestroyView. The onDestroyView method refers to a stage in the DialogFragment lifecycle that runs when the dialog window is shutting down. When the onDestroyView stage occurs, the `_binding` variable is set to null to prevent interactions with a user interface that no longer exists. Fragments can take longer to close than layouts, so access to the layout's binding class should be revoked as a safety precaution.

## The ShowNote dialog window

Similar to how we used the NewNote class to make the `new_note.xml` layout operational, we also need to make a

class for the **show\_note.xml** layout. The new class will load notes and allow the user to delete them if they wish. Like before, navigate through **Project > app > java** then right-click the folder with your project name. Select **New > Kotlin File/Class**. Name the file ShowNote and select Class from the list of options. Next, modify the contents of the **ShowNote.kt** file so it reads as follows:

```
import androidx.appcompat.app.AlertDialog
import androidx.fragment.app.DialogFragment

class ShowNote(private val note: Note, private val index: Int) : DialogFragment() {

    private var _binding: ShowNoteBinding? = null
    private val binding get() = _binding!!

    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {

        val callingActivity = activity as MainActivity
        val inflater = callingActivity.layoutInflater
        _binding = ShowNoteBinding.inflate(inflater)

        val builder = AlertDialog.Builder(callingActivity)
            .setView(binding.root)

        binding.txtTitle.text = note.title
        binding.txtContents.text = note.contents

        binding.btnOK.setOnClickListener{
            dismiss()
        }

        binding.btnDelete.setOnClickListener{
            callingActivity.deleteNote(index)

            Toast.makeText(callingActivity, resources.getString(R.string.note_deleted),
                Toast.LENGTH_SHORT).show()

            dismiss()
        }

        return builder.create()
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

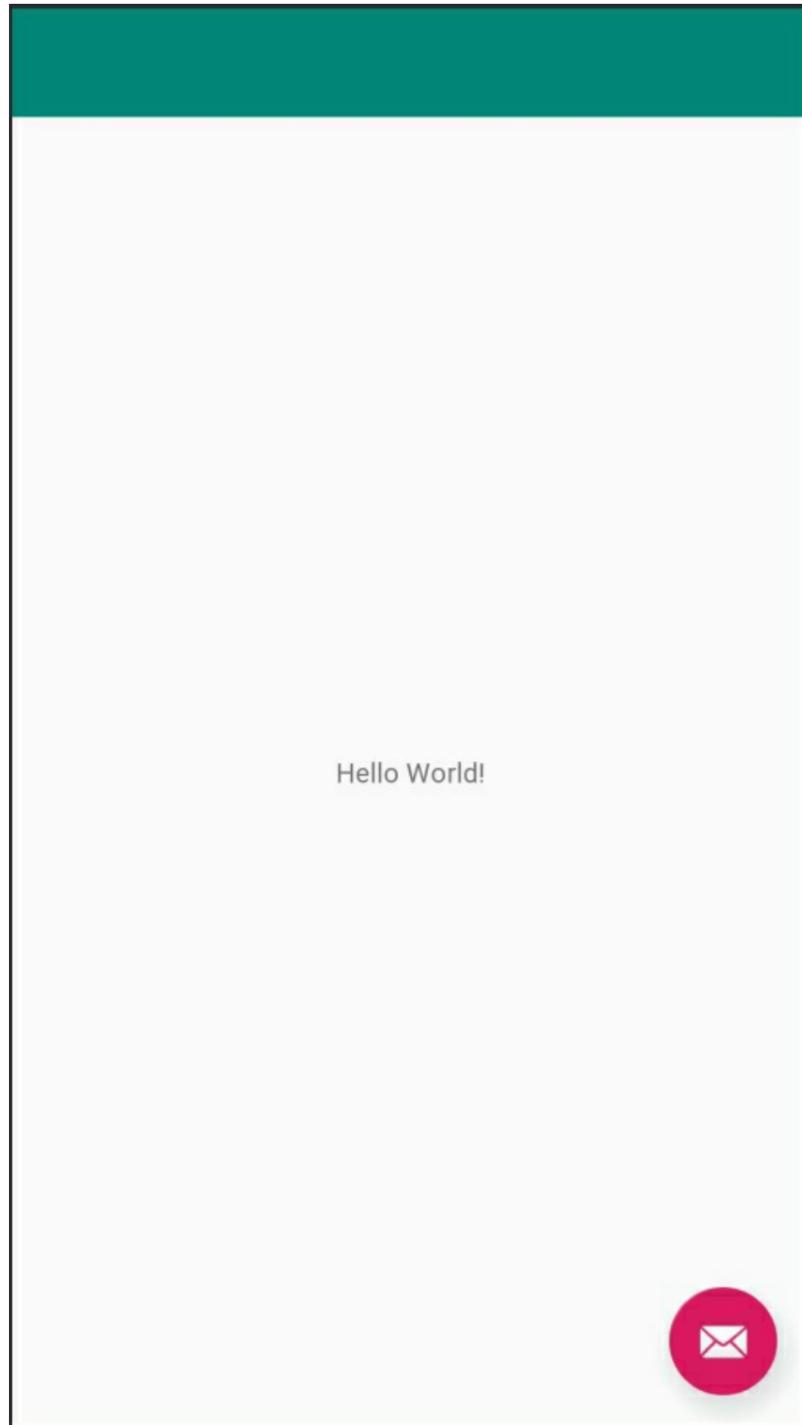
Similar to the NewNote class, the ShowNote class extends the DialogFragment class and inherits its data. Unlike the NewNote class, however, the ShowNote class contains two parameters (note and index) in its primary constructor. The parameters are marked as private, which means their values are only accessible to the ShowNote class. The note variable will store the Note object being displayed, while the index variable will contain an integer (an integer is a whole number) that identifies the Note object's position in the overall list of notes. The values of both parameters will be supplied by the MainActivity class when the dialog is created. The ShowNote class will then use this information to display the note to the user.

Next, an instance of the AlertDialog class is built and used to load the **show\_note.xml** layout file as a dialog window. The show\_note layout contains two TextView widgets, which will display the title and contents of the note, respectively. The note data is extracted from the Note object that is supplied in the ShowNote class's primary constructor and loaded into the TextView widgets so the user can read the note.

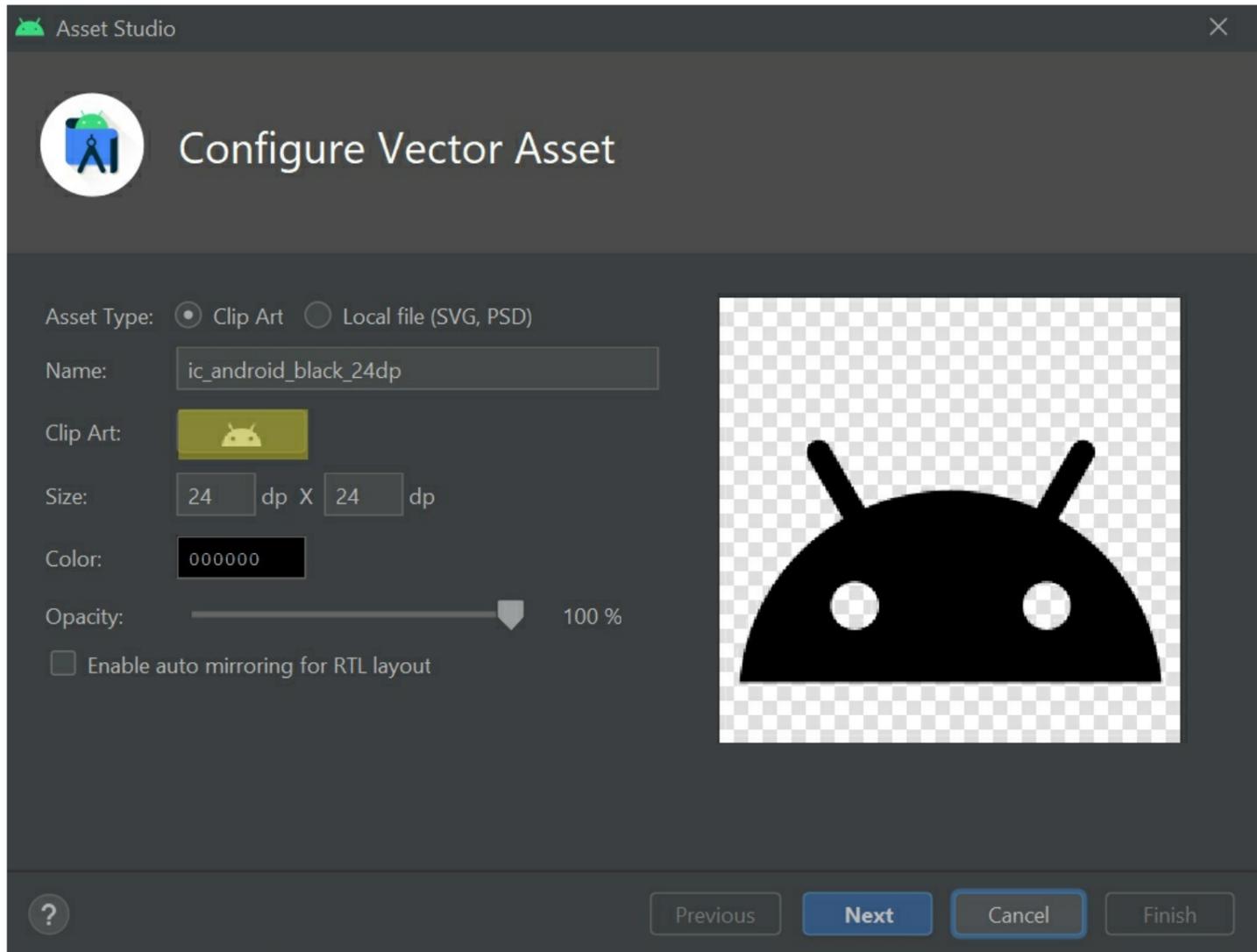
At the bottom of the show\_note layout, there is an OK button and a Delete button. The OK button will dismiss the dialog window and close the note, while the Delete button will run a method from the MainActivity class called deleteNote. The deleteNote method will delete the associated Note object from the app. Once this operation is complete, a toast notification will inform the user that the note has been deleted.

## The add new note floating action button

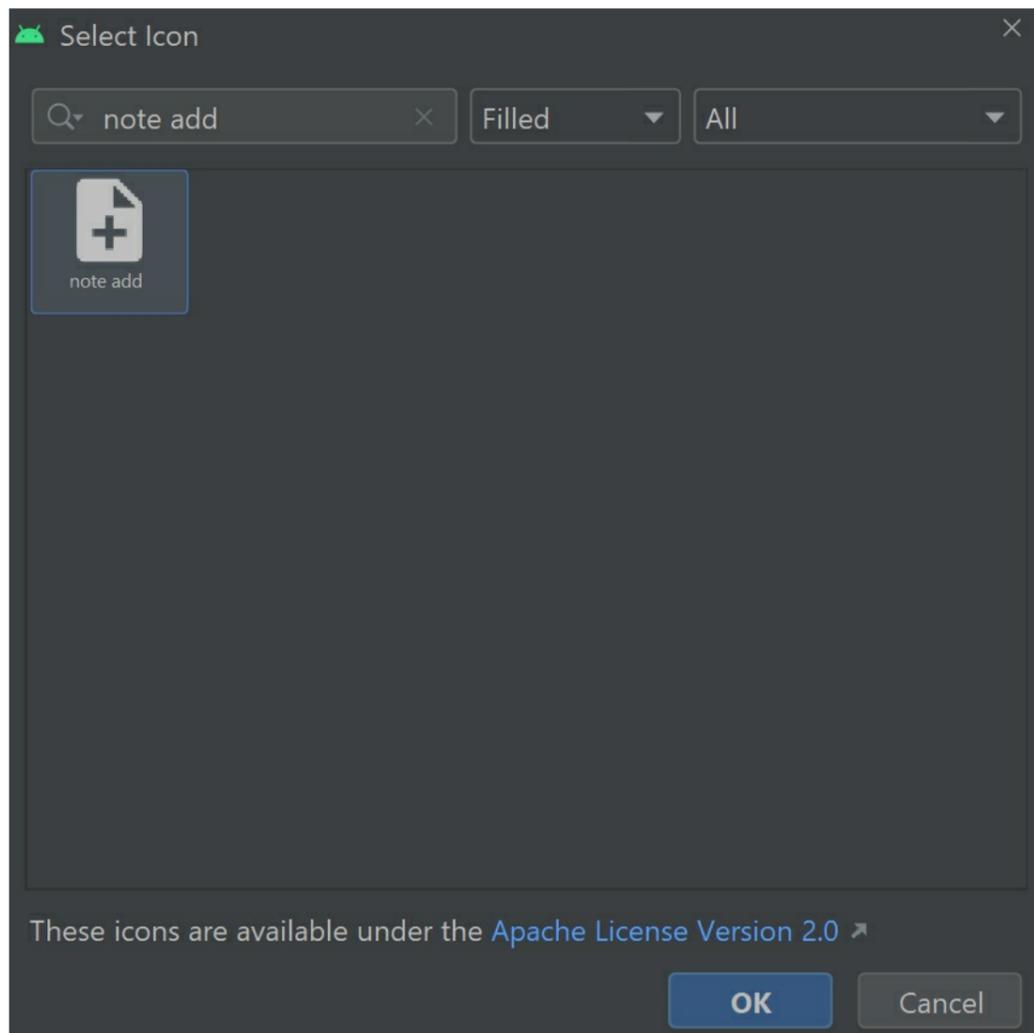
In this section, we'll turn our attention to the app homepage. The homepage will contain previews of all the user's notes and a floating action button that will allow the user to create a new note. By default, the first layout file to open when an app is launched is a readymade file called **activity\_main.xml**. Locate and open this file by navigating through **Project > app > res > layout**. When you create a project using the Basic Activity template, the activity\_main layout will often look like this:



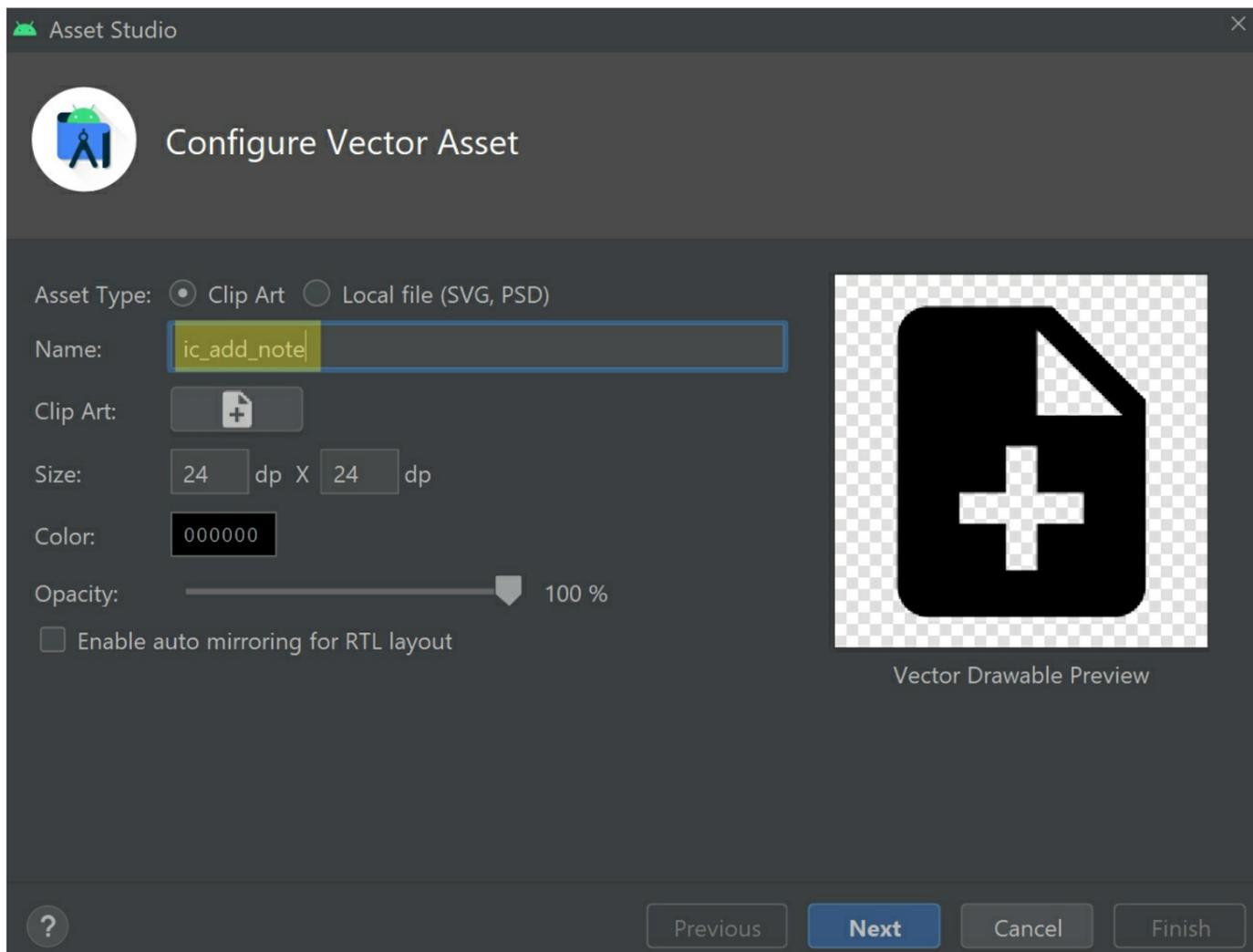
Needless to say, we're going to want to make a couple of changes. First, let's modify the floating action button, which is the pink circle with the mail icon in the bottom-right corner. The mail icon may be suitable for an email app but not the Notes app. To fix this, we will make a new icon using the Vector Asset Studio. To open the Vector Asset Studio, navigate through **Project > app** then right-click the **res** folder and select **New > Vector Asset**. In the Asset Studio window, click the image of the Android next to the phrase 'Clip Art:'.



In the Select Icon window, search for and select the 'note add' icon then press OK.



When you return to the Asset Studio window, set the name to ic\_add\_note. Once that is done, press Next followed by Finish to save the icon.



Returning to the **activity\_main.xml** file, switch the layout to Code view and find the `FloatingActionButton` element. Change the value of the `srcCompat` attribute to `@drawable/ic_add_note` to replace the mail icon with our new 'add note' icon. Also, add a content description attribute to let users who use screen readers know what the button does: `android:contentDescription="@string/add_new_note"`. Altogether, the code for the floating action button should now read like this:

```
<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|end"
    android:layout_margin="@dimen/fab_margin"
    android:contentDescription="@string/add_new_note"
    app:srcCompat="@drawable/ic_add_note" />
```

While the **activity\_main.xml** file is open, it is worth removing the `android:theme` attribute from the `AppBarLayout` element and the `app:popupTheme` attribute from the `Toolbar` element. Otherwise, these attributes will interfere with the day and night themes which we will implement later.

When clicked, the floating action button should open the `NewNote` dialog window. To achieve this, open the **MainActivity.kt** file (**Project > app > java > name of your project**) and edit the `onCreate` method so it reads as follows:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)
    setSupportActionBar(binding.toolbar)

    binding.fab.setOnClickListener {
        NewNote().show(supportFragmentManager, "")
    }
}
```

The `onCreate` method refers to the first stage of the activity's lifecycle. It will run when the activity has been launched and the components of the activity's user interface become operational. The user interface for the `MainActivity` class is defined in the **activity\_main.xml** layout. The above code initialises the `activity_main` layout's

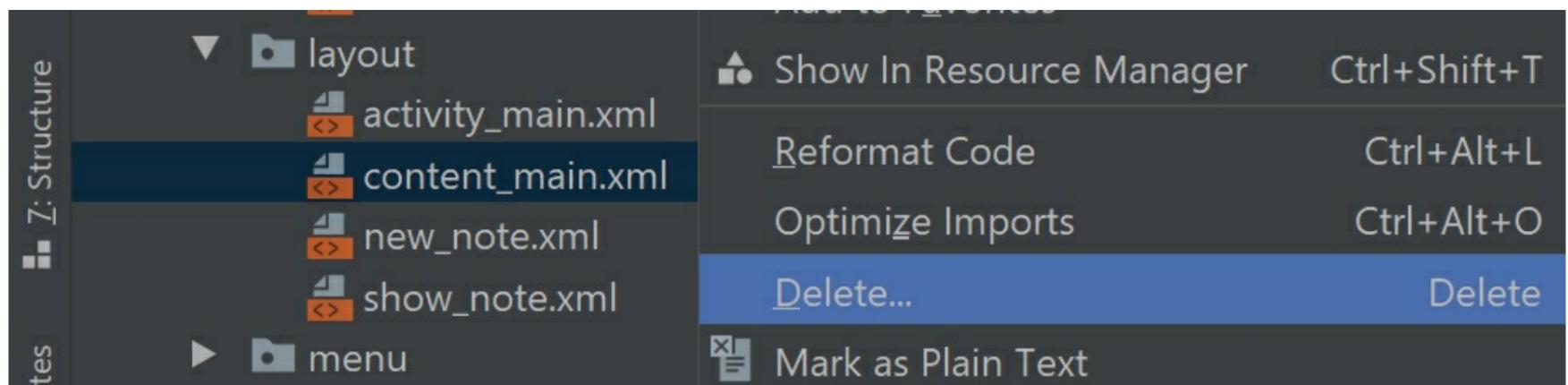
binding class and directs MainActivity to source its content and toolbar from the layout. Next, an onClick listener is assigned to the floating action button. If the floating action button is clicked, then the NewNote dialog fragment will be displayed using the DialogFragment class's inbuilt show method.

## Displaying note previews in a RecyclerView widget

A preview of each note will be displayed on the app homepage. If the user clicks a preview, then the full note will be loaded using the ShowNote dialog window. The collection of note previews must update dynamically as the user creates and deletes notes. To facilitate this, we will use a widget called a RecyclerView. RecyclerView widgets display lists of content. Typically, each list item will use the same layout, and the RecyclerView widget efficiently recycles this layout for all list items. To add the RecyclerView widget to the app homepage, open the **activity\_main.xml** layout (**Project** > **app** > **res** > **layout**) and replace the line that reads `<include layout="@layout/content_main" />` with the following code:

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/recyclerView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior" />
```

The above step replaces the reference to a layout file called **content\_main.xml** with a RecyclerView widget. In projects created using the Basic Activity template, Android Studio will typically split the homepage into two layouts: **activity\_main**, which contains static objects that rarely change as the user navigates around the activity, and **content\_main**, which is more dynamic and displays the content the user is viewing. Often, this content is provided by a fragment, which represents a navigational destination within the parent activity. In this app, the MainActivity activity will not be split into multiple navigational destinations. For this reason, it is simpler to remove the reference to the **content\_main** layout and insert the RecyclerView directly. Also, we should delete the **content\_main.xml** layout file itself. To do this, find the file in the **layout** directory, right-click it and press Delete.



The width and height of the RecyclerView are set to **match\_parent**, which means the RecyclerView will occupy the maximum amount of available space; however, we still want to leave a small gap at the top to ensure the RecyclerView is not obscured by the toolbar. To handle this, the layout behaviour of the RecyclerView is set to **AppBarLayout.ScrollingViewBehavior**. This behaviour helps coordinate layout components and the toolbar to ensure they do not obstruct one another.

The RecyclerView will display a preview of every note the user has saved. Note previews will require a layout, so right-click the **layout** directory and select **New** > **Layout resource file**. Name the file **note\_preview** then press OK. A layout file called **note\_preview.xml** should open automatically. Switch the layout to Code view then replace all its code with the following:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.cardview.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_margin="10dp"
    card_view:cardElevation="2dp" >
```

```
<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
```

```
android:padding="16dp">
```

```
<TextView  
    android:id="@+id/viewTitle"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:textSize="24sp" />
```

```
<TextView  
    android:id="@+id/viewContents"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:textSize="20sp" />
```

```
</LinearLayout>  
</androidx.cardview.widget.CardView>
```

The root element of the `note_preview` layout is a `CardView` widget. `CardView`-based layouts appear slightly elevated above their containing view group (the `RecyclerView` in this instance). They provide a useful way of displaying list items while keeping the style of each item consistent. In the above code, two `TextView` widgets are added to the `CardView`. The `TextView` widgets will display the title of the note and a preview of its contents, respectively. The `TextView` widgets are packaged inside a `LinearLayout` widget so they align themselves vertically within the layout.

The list of note previews will be loaded into the `RecyclerView` using an adapter, which requires a new class. Create a Kotlin class called `NoteAdapter` in the usual way (right-click on **Project** > **app** > **java** > **name of your project** then select **New** > **Kotlin Class/File**) and modify the class's code so it reads as follows:

```
class NoteAdapter(private val mainActivity: MainActivity):  
    RecyclerView.Adapter<NoteAdapter.ViewHolderNote>() {  
  
    var noteList = mutableListOf<Note>()  
  
    inner class ViewHolderNote(view: View) :  
        RecyclerView.ViewHolder(view),  
        View.OnClickListener {  
  
        internal var mTitle = view.findViewById<View>(R.id.viewTitle) as TextView  
        internal var mContents = view.findViewById<View>(R.id.viewContents) as TextView  
  
        init {  
            view.isClickable = true  
            view.setOnClickListener(this)  
        }  
  
        override fun onClick(view: View) {  
            mainActivity.showNote(layoutPosition)  
        }  
    }  
  
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolderNote {  
        return ViewHolderNote(LayoutInflater.from(parent.context).inflate(R.layout.note_preview, parent, false))  
    }  
  
    override fun onBindViewHolder(holder: ViewHolderNote, position: Int) {  
        val note = noteList[position]  
  
        holder.mTitle.text = note.title  
        holder.mContents.text = if (note.contents.length < 15) note.contents  
            else note.contents.substring(0, 15) + "..."  
    }  
  
    override fun getItemCount(): Int = noteList.size  
}
```

The `NoteAdapter` class's primary constructor features a parameter called `mainActivity`, which will hold a reference to the `MainActivity` class. Inside the `NoteAdapter` class, there is a variable called `noteList`, which will contain the

details of every note the user has saved. The list is a mutable list, which means items can be added, removed and changed. Next, an inner class called `ViewHolderNote` is established. Inner classes can access data such as variables and methods from the outer class and vice versa, even if that content is marked as private. In this instance, the `ViewHolderNote` inner class will initialise the components of the `note_preview.xml` layout so they can be used by the adapter. The `ViewHolderNote` inner class also defines what action should occur when a note preview is clicked. If the user clicks a note preview, then a `MainActivity` method called `showNote` will load the full note using the `ShowNote` dialog fragment.

Next, we define several methods that shape how the `RecyclerView` widget operates. The `onCreateViewHolder` method tells the `NoteAdapter` to use the `note_preview.xml` layout for every item that is loaded into the `RecyclerView`. Meanwhile, the `onBindViewHolder` method retrieves the title and contents of each `Note` object in the `noteList` list and uses this information to populate the `TextView` widgets in the `note_preview` layout. For this purpose, an `if` expression is used to assess the length of the contents string. If the contents string is less than 15 characters long then the full contents will be displayed in the preview. Otherwise, the contents string will be shortened to 15 characters using Kotlin's `substring` method because we only want to display a small preview of the note's contents. An ellipsis is appended to the end of the shortened contents string to show the user that we are displaying only a preview of the note. Finally, a method called `getItemCount` is defined, which will calculate how many items are loaded into the `RecyclerView`. In this case, the number of items will equal the size of the list of notes.

## Initialising the `RecyclerView` widget, adapter and dialog windows

In this section, we will add the code to the `MainActivity.kt` file that is required to bind the `NoteAdapter` adapter to the `RecyclerView` widget and allow the `NewNote` and `ShowNote` dialog windows to function. First, however, we need to tidy up the `MainActivity` class and remove some code that was autogenerated by Android Studio. For example, we will not use the `appBarConfiguration` variable so you can delete this variable from the top of the class:

```
private lateinit var appBarConfiguration: AppBarConfiguration
```

Also, if you see a function called `onSupportNavigateUp` then remove that function too. Both the `appBarConfiguration` variable and `onSupportNavigateUp` function help coordinate navigation around the app; however, the only destination in the Notes app is the app homepage so navigation is not required. Once the surplus code has been removed, add the following variable below the binding variable at the top of `MainActivity` to store a reference to the `NoteAdapter` class:

```
private lateinit var adapter: NoteAdapter
```

The adapter variable features the `lateinit` modifier, which indicates that it is a non-null variable and its value will be assigned later. An alternative to the `lateinit` modifier could be to define the variable conventionally but assign it a null value as shown in this example: `private var adapter: NoteAdapter? = null`. The drawback of this approach is it requires you to always accommodate for the value of the variable potentially being null. The `lateinit` modifier removes this consideration because the app will assume the variable has been initialised and assigned a value. If you attempt to reference a `lateinit` variable in your code before the variable has been initialised then an error called an exception will occur and the app may crash. Hence, it is important to assign `lateinit` variables a value before they are referenced elsewhere in your code.

Moving on, we'll now set up the `NoteAdapter` adapter and `RecyclerView` widget. To do this, add the following code to the bottom of the `onCreate` method:

```
adapter = NoteAdapter(this)
```

```
binding.recyclerView.layoutManager = LinearLayoutManager(applicationContext)
```

```
binding.recyclerView.itemAnimator = DefaultItemAnimator()
```

```
binding.recyclerView.adapter = adapter
```

The adapter variable is initialised by building an instance of the `NoteAdapter` class and passing a value of 'this' (in this instance 'this' refers to the `MainActivity` class) for the `mainActivity` parameter in the `NoteAdapter` class's primary constructor. Next, we set up the `RecyclerView` widget. `RecyclerView` widgets use a layout manager to organise their content. The list of note previews will be stacked linearly one by one, so the `LinearLayoutManager` is best. Other layout managers such as `GridLayoutManager` are also available (we will use the `GridLayoutManager` in the Camera app project). Next, an instance of the `DefaultItemAnimator` class is applied to the `RecyclerView`, to provide some basic animations when items are added, removed or changed. Finally, the `NoteAdapter` adapter is assigned to manage the content of the `RecyclerView`.

Moving on, we'll now discuss two methods that will save and delete the user's notes, respectively. The methods

will also notify the NoteAdapter adapter of any changes. Add the following code below the onOptionsItemSelected function in the MainActivity class:

```
fun createNewNote(n: Note) {  
    adapter.noteList.add(n)  
    adapter.notifyItemInserted(adapter.noteList.size - 1)  
    saveNotes()  
}
```

```
fun deleteNote(index: Int) {  
    adapter.noteList.removeAt(index)  
    adapter.notifyItemRemoved(index)  
    saveNotes()  
}
```

The createNewNote method is called by the NewNote dialog window when the user attempts to save a note. The title and contents of the note (as entered by the user) are packaged in a Note object and sent to the createNewNote method. The createNewNote method adds the Note object to the end of the list of notes stored in the NoteAdapter instance and runs a method automatically programmed into all adapters called notifyItemInserted. The notifyItemInserted method tells the RecyclerView widget that a new item has been inserted at a given index. In this case, the item has been added to the end of the list, so the index will be the size of the list minus one. The reason for this is that the index of the first item in a list is 0, so we must subtract one from the size of the list to get the index of the last item. The deleteNote method does the opposite of the createNewNote method: it removes the Note object at a given index from the adapter then calls notifyItemRemoved to update the RecyclerView. Whenever a note is saved or deleted, a method called saveNotes will update the list of notes saved in the application.

Finally, let's define a method called showNote which will load the full version of a note and display it to the user. Add the following code below the deleteNote method:

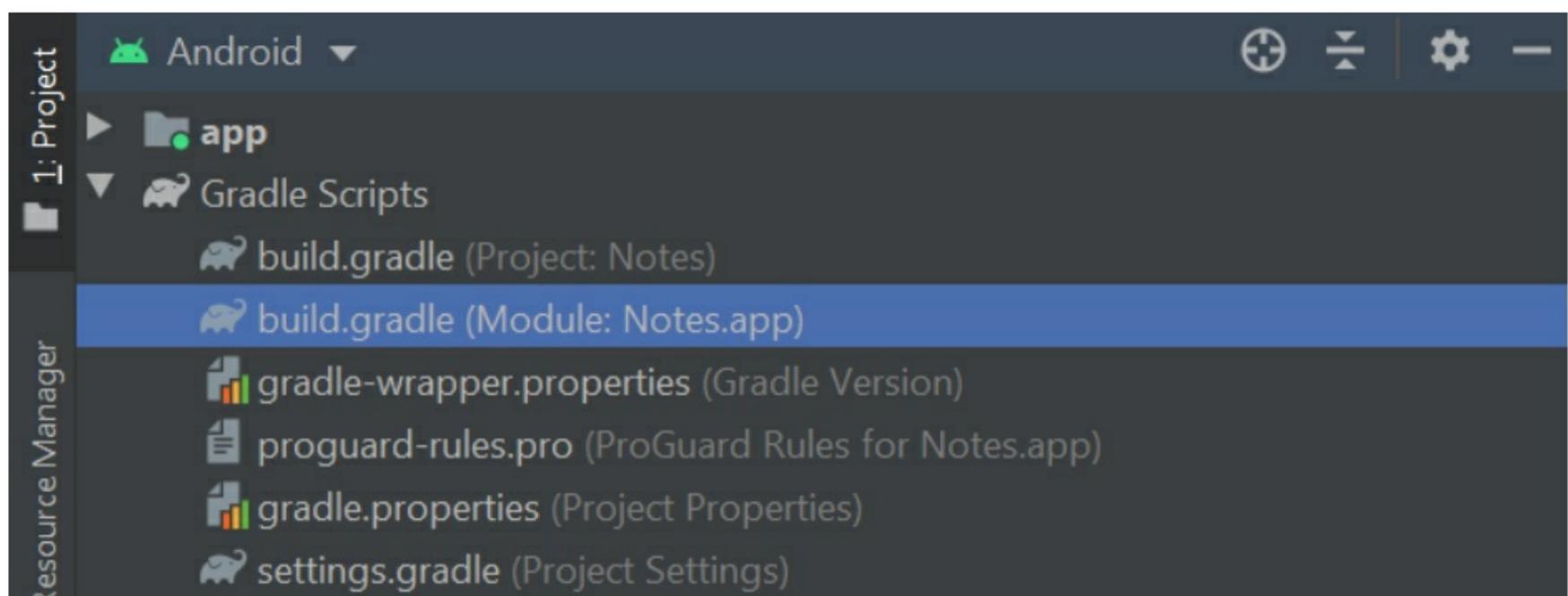
```
fun showNote(index: Int) {  
    val dialog = ShowNote(adapter.noteList[index], index)  
    dialog.show(supportFragmentManager, "")  
}
```

The showNote method sends the user's selected Note object and its index within the overall list of notes to the ShowNote dialog window. The ShowNote class will use this information to display the full note and handle requests to delete it.

## Using JSON to save the user's notes

The user's notes will be stored in JavaScript Object Notation (JSON) format, which is a text-based system for handling data. The list of Note objects saved by the user will be converted to a JSON string and saved internally in the app. When the app is launched, the JSON string will be converted back to a list of Note objects that can be used by the app. The conversion of objects to and from JSON is known as serialisation and deserialisation, respectively. To facilitate these processes, we will use a Java library called GSON

(<https://github.com/google/gson/blob/master/UserGuide.md>). To add GSON to the app, open the module-level **build.gradle** file by navigating through **Project > Gradle Scripts**.



Locate the dependencies section and add the following implementation statement to import the GSON library into the project:

```
implementation 'com.google.code.gson:gson:2.8.8'
```

Don't forget to re-sync your project when prompted!

Gradle files have changed since last project sync. A project sync may be necessary for the IDE ...[Sync Now](#)

Let's now write the code that will convert the user's notes into JSON and save the JSON string internally within the app. Return to the **MainActivity.kt** file and add the following companion object below the list of variables at the top of the class:

```
companion object {  
    private const val FILEPATH = "notes.json"  
}
```

Companion objects are initialised when the outer class (MainActivity in this instance) loads. In this companion object, a variable called FILEPATH is defined which contains the name of the file that will store the JSON string. Other methods in the MainActivity class can refer to the FILEPATH variable to ensure they are all interacting with the same file. The method which saves the user's notes is called saveNotes. To define the saveNotes method, add the following code below the showNote method:

```
private fun saveNotes() {  
    val notes = adapter.noteList  
    val gson = GsonBuilder().create()  
    val jsonNotes = gson.toJson(notes)  
  
    var writer: Writer? = null  
    try {  
        val out = this.openFileOutput(FILEPATH, Context.MODE_PRIVATE)  
  
        writer = OutputStreamWriter(out)  
        writer.write(jsonNotes)  
    } catch (e: Exception) {  
        writer?.close()  
    } finally {  
        writer?.close()  
    }  
}
```

The saveNotes method retrieves the full list of Note objects from the NoteAdapter class and uses the GsonBuilder class to convert the list to a JSON string. The Writer and OutputStreamWriter classes then encode the JSON string as a stream of data that can be written to a file within the app. The details of the file are defined in a variable called out, which stores an instance of the FileOutputStream class. The FileOutputStream class is initialised using the openFileOutput method and by supplying the name of the file and an operating mode. In this instance, the operating mode is set to private, which means the **notes.json** file will only be accessible to this application.

The workflow which writes data to the **notes.json** file is enclosed in a try/catch block because an error could occur. If an error occurs, then the workflow will throw an exception, which contains information about what went wrong. Typically, exceptions cause the application to crash; however, the crash can be avoided by 'catching' the exception(s). In this instance, the catch block catches every variety of exception, although you can also specify the specific types of exception you wish to catch. In the saveNotes method, if an exception is thrown then we simply close the writer and safely abandon the process without crashing the application. If the list of Note objects is successfully written to the **notes.json** file without any exceptions being thrown, then the finally block will run and the writer will be closed similar to if the catch block had run. Closing the writer once it is no longer required helps maintain the security and performance of the application.

The list of Note objects stored in the **notes.json** file will be retrieved each time the app is launched. This operation is managed by a method called retrieveNotes, which can be defined by adding the following code below the saveNotes method:

```
private fun retrieveNotes(): MutableList<Note> {  
    var noteList = mutableListOf<Note>()  
    if (this.getFileStreamPath(FILEPATH).isFile) {
```

```

var reader: BufferedReader? = null
try {
    val fileInput = this.openFileInput(FILEPATH)
    reader = BufferedReader(InputStreamReader(fileInput))
    val stringBuilder = StringBuilder()

    for (line in reader.readLine()) stringBuilder.append(line)

    if (stringBuilder.isNotEmpty()){
        val listType = object : TypeToken<List<Note>>() {}.type
        noteList = Gson().fromJson(stringBuilder.toString(), listType)
    }
} catch (e: Exception) {
    reader?.close()
} finally {
    reader?.close()
}
}
return noteList
}

```

The retrieveNotes method will return a mutable list of Note objects. It does this by defining a variable called noteList, which will store the list of Note objects. Next, it checks whether a file exists at the location specified in the FILEPATH variable. If no file exists, then an empty list is returned by the method; however, if a file does exist, then the file's contents are decoded using the BufferedReader and InputStreamReader classes. The decoded contents are collated using an instance of the StringBuilder class and converted to a list of Note objects using GSON. A catch block is provided to intercept any exceptions that occur and close the BufferedReader. The BufferedReader is also closed if the process completes successfully because it is no longer required.

The retrieveNotes method will need to run when the app is launched so the notes can be displayed on the app homepage. To arrange this, add the following code to the bottom of the onCreate method:

```

adapter.noteList = retrieveNotes()
adapter.notifyItemRangeInserted(0, adapter.noteList.size)

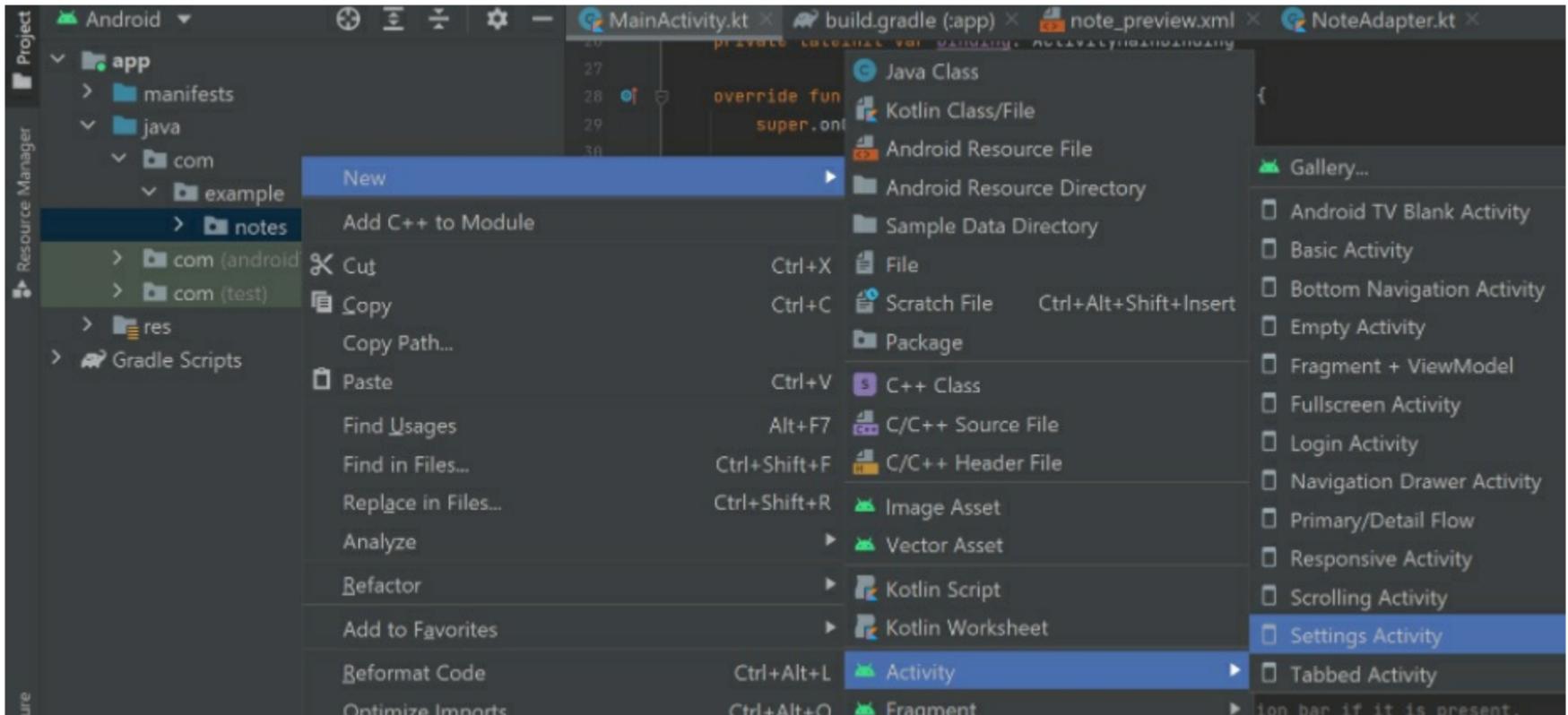
```

The list of notes returned by the retrieveNotes method is loaded into the NoteAdapter instance and displayed to the user using the adapter's notifyItemRangeInserted method. The notifyItemRangeInserted method notifies the adapter that one or more items have been added to the RecyclerView. In this instance, we are loading notes into an empty RecyclerView, so the new items will be inserted at index 0 (the first position in the RecyclerView) and the number of items will equal the size of the list of notes. In this way, we use the notifyItemRangeInserted method to load the entire list of notes into the RecyclerView widget and make them visible to the user.

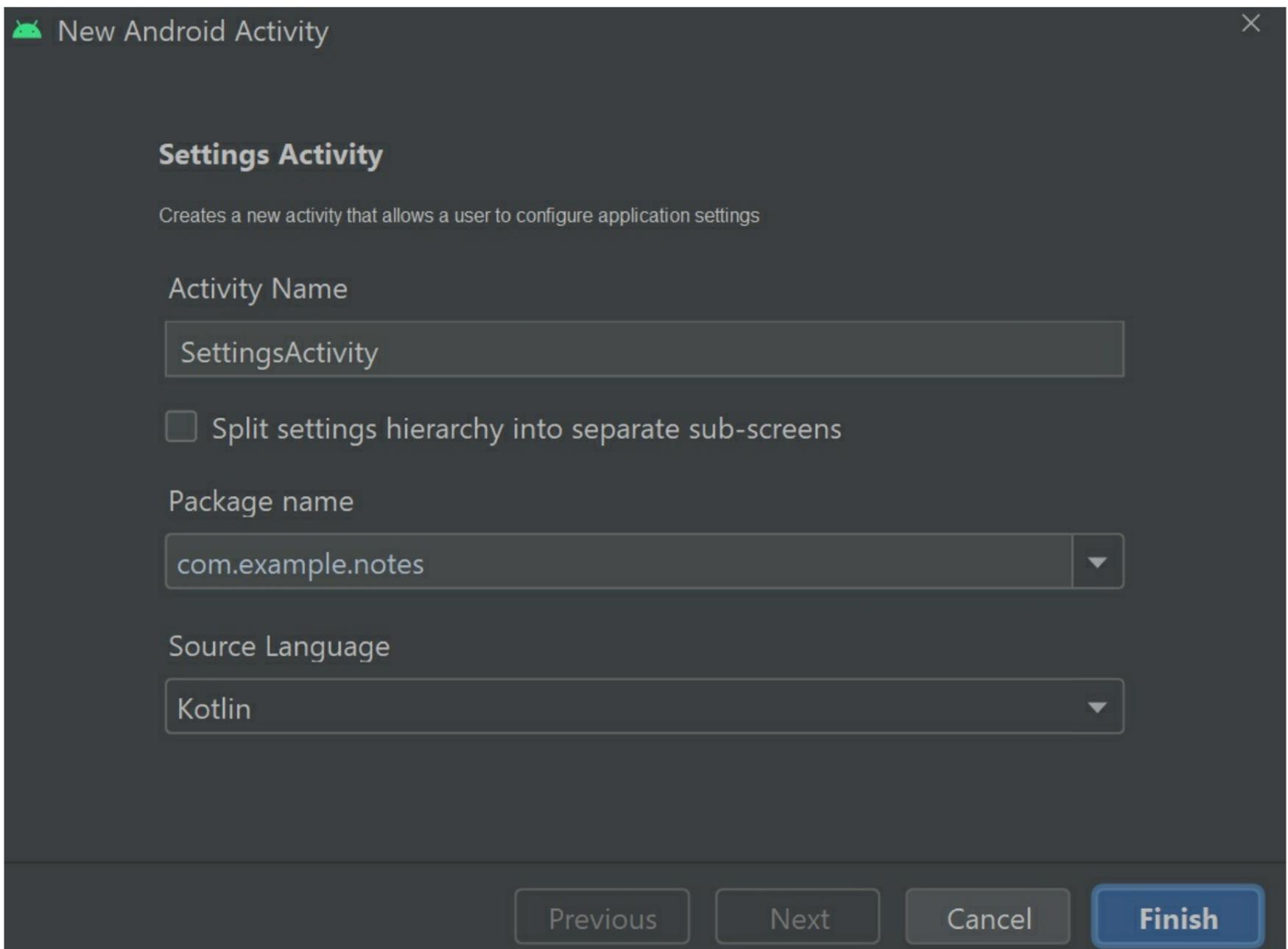
And that's it! The app will now save the user's notes when it is closed and retrieve them again when it is opened. The remainder of this guide will focus on building a Settings page and allowing the user to personalise the app.

## Creating the Settings activity and preferences file

The Notes app will contain a settings page that allows users to add and remove dividing lines between notes on the home screen and toggle the theme between day and night modes. The settings page will be managed by a distinct activity. To create a new activity, navigate through **Project > app > java** then right-click the folder with the name of your project. Select **New > Activity > Settings Activity**.



Name the activity `SettingsActivity` then press Finish.



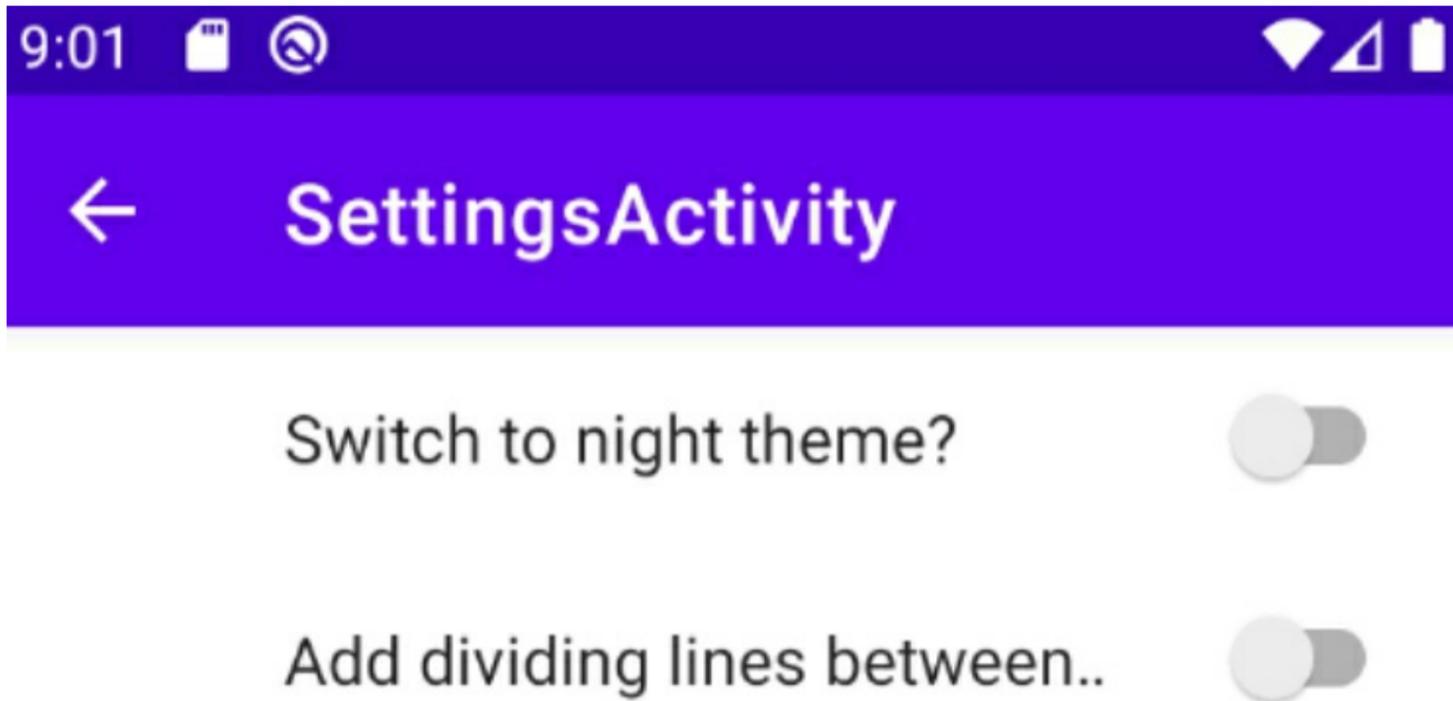
A file called **SettingsActivity.kt** should then open in the editor. The first task will be to ensure the user can find their way back to the app homepage. Android will automatically generate a back button in the top left corner of the app bar; however, we need to instruct the button to return the user to the MainActivity activity when clicked. To do this, add the following code below the onCreate method:

```

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    if (item.itemId == android.R.id.home) finish()
    return super.onOptionsItemSelected(item)
}

```

The above code uses the finish command to close the SettingsActivity activity when the home button is pressed and return the user to the app homepage.



Moving on, let's define the code that handles changes to the user's preferences. When the SettingsActivity class was created, Android Studio should automatically have generated a preferences XML file. To locate the file, navigate through **Project** > **res** > **xml** and open the file called **root\_preferences.xml**. Switch the preferences file to Code view and replace the code in the file with the following:

```
<PreferenceScreen xmlns:app="http://schemas.android.com/apk/res-auto">
```

```
<SwitchPreferenceCompat
  app:key="theme"
  app:defaultValue="false"
  app:title="@string/select_theme" />
```

```
<SwitchPreferenceCompat
  app:key="dividingLines"
  app:defaultValue="false"
  app:title="@string/add_dividers" />
```

```
</PreferenceScreen>
```

The above code defines two SwitchPreferenceCompat elements, which the user can toggle between on and off states. The first switch allows the user to enable and disable the night theme, while the second switch allows the user to add and remove dividing lines from between notes on the app homepage. Each switch returns a boolean (a true or false value) depending on whether the switch is activated or not. The user's selections are stored internally in a shared preferences file which can be accessed from anywhere within the app. To change these preferences, we must add some code to the SettingsActivity and MainActivity classes. First, return to the **SettingsActivity.kt** file and edit the SettingsFragment class's code so it reads as follows:

```
class SettingsFragment : PreferenceFragmentCompat(), SharedPreferences.OnSharedPreferenceChangeListener {
```

```
  override fun onCreatePreferences(savedInstanceState: Bundle?, rootKey: String?) {
    setPreferencesFromResource(R.xml.root_preferences, rootKey)
```

```
    preferenceManager.sharedPreferences?.registerOnSharedPreferenceChangeListener(this)
  }
```

```
  override fun onResume() {
    super.onResume()
    preferenceManager.sharedPreferences?.registerOnSharedPreferenceChangeListener(this)
  }
```

```
  override fun onPause() {
    super.onPause()
    preferenceManager.sharedPreferences?.unregisterOnSharedPreferenceChangeListener(this)
  }
```

```

override fun onSharedPreferenceChanged(
    sharedPreferences: SharedPreferences?,
    key: String?
) {
    sharedPreferences?.run {
        when (key) {
            "theme" -> {
                val nightThemeSelected = sharedPreferences.getBoolean(key, false)
                if (nightThemeSelected)
                    AppCompatActivity.setDefaultNightMode(AppCompatActivity.MODE_NIGHT_YES)
                else AppCompatActivity.setDefaultNightMode(AppCompatActivity.MODE_NIGHT_NO)
            }
        }
    }
}
}
}

```

The SettingsFragment fragment handles changes to the user's preferences in real-time. It does this by registering an onSharedPreferenceChange listener to the shared preferences file when the fragment is created. The listener will monitor the user's preferences and respond to changes. If the user navigates away from the fragment or closes the app while the fragment is active then the onPause stage of the fragment lifecycle will run and the onSharedPreferenceChange listener will be unregistered; however, if the user then returns to the fragment the onResume stage of the fragment lifecycle will run and the listener will be registered again.

The onSharedPreferenceChanged method at the bottom of the fragment responds to changes in the user's preferences. If the changed preference has the key theme then the method uses Android's AppCompatActivity class to activate or deactivate the night theme as necessary. We could implement code to handle changes to the dividingLines preference here; however, this would not be particularly effective because note previews are not displayed in the SettingsActivity activity. Changes to the dividingLines preferences are better handled by the MainActivity class. Return to the **MainActivity.kt** file and add the following code to the list of variables at the top of the class:

```
private lateinit var sharedPreferences: SharedPreferences
```

The above variable will provide access to the shared preferences file that was modified by the settings activity. To initialise the variable, add the following code to the bottom of the onCreate method:

```
sharedPreferences = PreferenceManager.getDefaultSharedPreferences(this)
```

Note you may need to import the SharedPreferences and PreferenceManager classes manually by adding the following import statements to the top of the file:

```
import android.content.SharedPreferences
import androidx.preference.PreferenceManager
```

The user's preferences should be applied each time the app is launched. To handle this, add the following code to the MainActivity class below the onCreate method:

```

override fun onStart() {
    super.onStart()

    val nightThemeSelected = sharedPreferences.getBoolean("theme", false)
    if (nightThemeSelected) AppCompatActivity.setDefaultNightMode(AppCompatActivity.MODE_NIGHT_YES)
    else AppCompatActivity.setDefaultNightMode(AppCompatActivity.MODE_NIGHT_NO)

    val showDividingLines = sharedPreferences.getBoolean("dividingLines", false)
    if (showDividingLines) binding.recyclerView.addItemDecoration(DividerItemDecoration(this,
        LinearLayoutManager.VERTICAL))
    else if (binding.recyclerView.itemDecorationCount > 0) binding.recyclerView.removeItemDecorationAt(0)
}
}

```

This onStart method refers to a stage of the activity lifecycle that runs when the activity is displayed to the user, such as when the user returns from the SettingsActivity activity. In this instance, we instruct the onStart method to retrieve the user's theme preference from the shared preferences file and use the AppCompatActivity class's setDefaultNightMode method to activate or deactivate the night theme accordingly. Next, the user's dividingLines

preference is retrieved. If the `dividingLines` preference is set to true, then a `DividerItemDecoration` instance is applied to the `RecyclerView` widget from the `activity_main.xml` layout. The `DividerItemDecoration` instance will add a dividing line between each note preview. Conversely, if the `dividingLines` preference is false, then the dividing lines are removed from the `RecyclerView`.

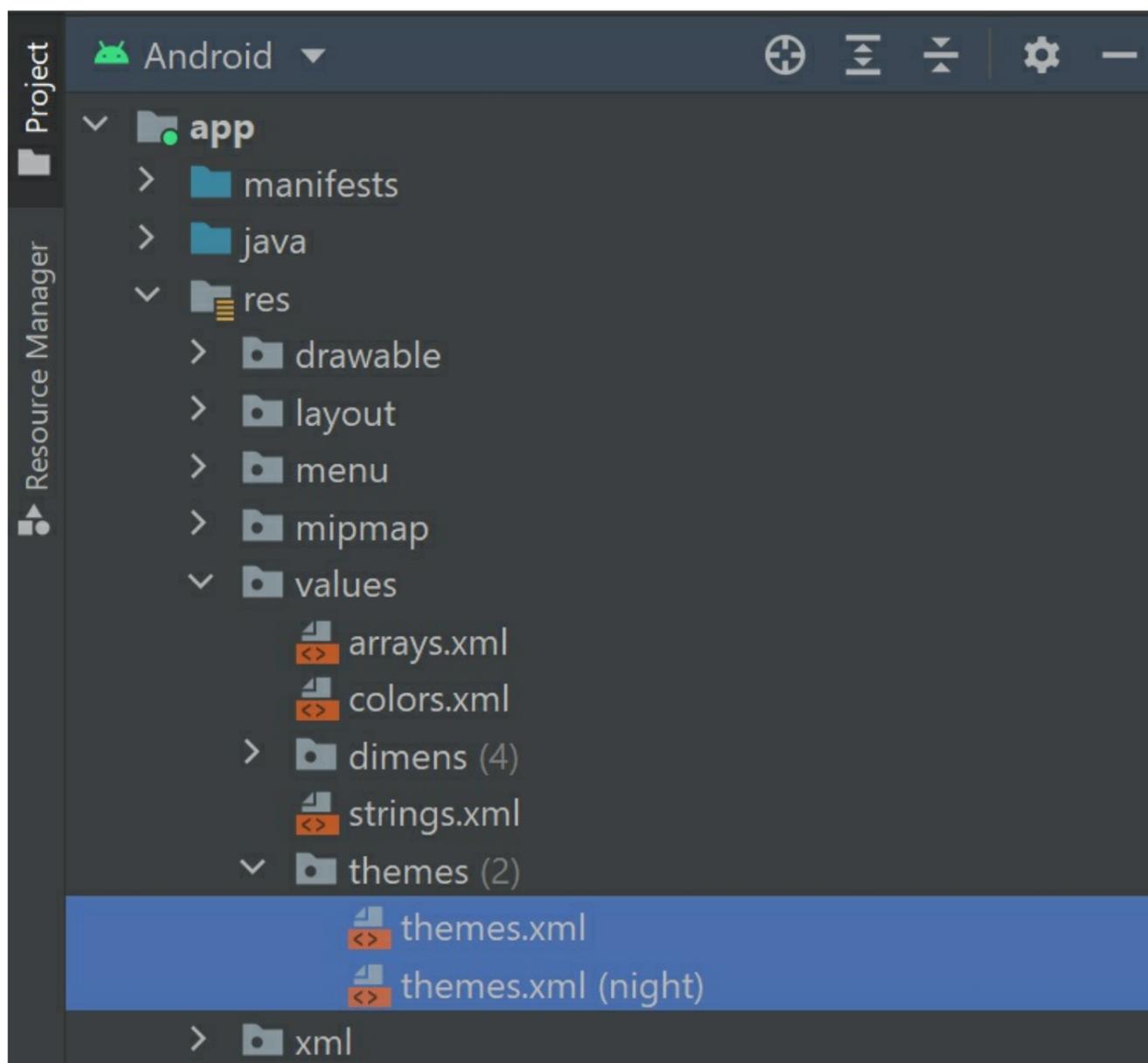
The last thing we'll do is configure the toolbar menu so that the user can navigate from the homepage to the settings page. Android Studio helpfully generates a settings page menu item as part of the Basic Activity project template. To make the menu item operational, locate the `onOptionsItemSelected` method and edit the return block so it reads as follows:

```
return when (item.itemId) {
    R.id.action_settings -> {
        val intent = Intent(this, SettingsActivity::class.java)
        startActivity(intent)
        true
    }
    else -> super.onOptionsItemSelected(item)
}
```

The block of code defined above listens for clicks on menu items and runs the appropriate actions based on the menu item's ID. If the user clicks on the settings menu item, then an intent will launch the `SettingsActivity` activity. In Android programming, an intent describes an action that should be performed. Often the intent will involve another activity, service or application. Once the `SettingsActivity` activity has been launched, the `when` block returns a value of `true` to signal that no further processing is required.

## Designing the themes

In this section, we will customise the night and day themes. When the project was created, Android Studio will have generated two resource files to store the details of the day and night themes. To locate the theme resource files, navigate through **Project** > **res** > **values** > **themes**. You can distinguish between the two files because the night theme file will have the word `night` in brackets at the end:



First, open the day `themes.xml` file and locate the style element that has a name attribute set to `Theme.Notes`. The

Theme.Notes style element defines the base theme for the application. We want to import a readymade theme and override some colours, so edit the style element so it reads as follows:

```
<style name="Theme.Notes" parent="Theme.MaterialComponents.DayNight">  
  <item name="colorPrimary">@color/colorPrimary</item>  
  <item name="colorPrimaryDark">@color/colorSecondary</item>  
  <item name="colorSecondary">@color/colorSecondary</item>  
</style>
```

In the above code, the parent attribute of the opening style tag imports the information from a readymade Material Design theme called DayNight. You can learn more about the DayNight theme here: <https://material.io/develop/android/theming/dark/>. It uses a readymade set of colours which you can find below in the colour chart (1: Day mode, 2: Night mode):

1	2
Primary 500 #6200EE	Primary 200 #BB86FC
Primary Variant 700 #3700B3	Primary Variant 700 #3700B3
Secondary 200 #03DAC6	Secondary 200 #03DAC6
Secondary Variant 900 #018786	Secondary Variant 200 #03DAC6
Background #FFFFFF	Background #121212
Surface #FFFFFF	Surface #121212
Error #B00020	Error #CF6679
On Primary #FFFFFF	On Primary #000000
On Secondary #000000	On Secondary #000000
On Background #000000	On Background #FFFFFF
On Surface #000000	On Surface #FFFFFF
On Error #FFFFFF	On Error #000000

It is possible to override the colours used in a theme. To do this, simply insert an item into the base style element as shown in the above example. The name of the item should be the theme attribute that you wish to override. If you are looking to override a colour then begin the item name with 'color' followed by the name of the colour. For example, to override the secondary variant colour then the item name should be 'colorSecondaryVariant'. The colour that you wish to use instead will sit between the item tags and take the format `@color/colorPrimary`, where 'colorPrimary' is the name of the colour in the **colors.xml** resource file. We will discuss the **colors.xml** file shortly. For now, you can either just copy and paste the code from above or wait until you test the app yourself and make a

note of which colours you would like to change.

### What to do if parent="Theme.MaterialComponents.DayNight" causes an error.

If references to the DayNight MaterialComponents theme are highlighted as errors then this suggests the necessary files have not been imported properly. To fix this, navigate through **Project > Gradle Scripts** and open the two **build.gradle** files (Project: Notes and Module: app). In the Project **build.gradle** file, check that the repositories section includes the reference to Google as shown below:

```
allprojects {
    repositories {
        google()
        jcenter()
    }
}
```

Next, in the Module **build.gradle** file, check that the following implementation command is present in the dependencies section:

```
implementation 'com.google.android.material:material:1.5.0'
```

If you need to make any changes then remember to resync the project when prompted by Android Studio.

If the above steps do not fix the problem then you may like to check out the 'Getting started with Material Components for Android' guide found here:  
<https://material.io/develop/android/docs/getting-started/>

Moving on, let's customise the night theme. Open the night version of the **themes.xml** file and edit the Theme.Notes style element so it reads as follows:

```
<style name="Theme.Notes" parent="Theme.MaterialComponents">
    <item name="colorPrimary">@color/nightColorPrimary</item>
    <item name="colorSecondary">@color/nightColorSecondary</item>
</style>
```

For the night theme, the parent attribute of the base style element is set to incorporate a night theme from Material Design. The style element proceeds to override the theme's primary and secondary colours with our selections.

Once the night theme file is set up, we can turn our attention to the **colors.xml** file. The **colors.xml** file will contain the details of any custom colours used in the app. To locate **colors.xml** file, navigate through **Project > app > res > values**. Next, replace the color elements in the file with the following code:

```
<color name="colorPrimary">#E85A4F</color>
<color name="colorSecondary">#E98074</color>

<color name="nightColorPrimary">#272727</color>
<color name="nightColorSecondary">#84C9FB</color>
```

Each color item has a name attribute, which is used by other files to access the hexadecimal (HEX) code found between the color tags. You can find the HEX code for many common colours by using the colour picker and table on our website: <https://codersguidebook.com/how-to-build-a-website/colour-picker>.

## Summary

Congratulations on completing the Notes app! In creating this app, you have covered the following skills and topics:

- Create a new application using the Basic Activity project template.
- Use string, colour and theme resources to display text and customise the appearance of the app.
- Build a user interface layout using widgets such as Button, EditText, TextView and RecyclerView.
- Register onClick listeners and respond to user interactions with widgets and menu items.
- Explore a principle of object-oriented programming called inheritance to import the data from one class into another class.
- Use the DialogFragment and AlertDialog classes to display a pop-up window.
- Create, store and decode JSON strings.
- Intercept exceptions that might otherwise cause the app to crash.
- Write and retrieve data from a file internally within the app.
- Utilise a Settings Activity to allow the user to customise the app based on their preferences.
- Import readymade themes from Material Design and customise the colours used in those themes.

# How to create a Camera application

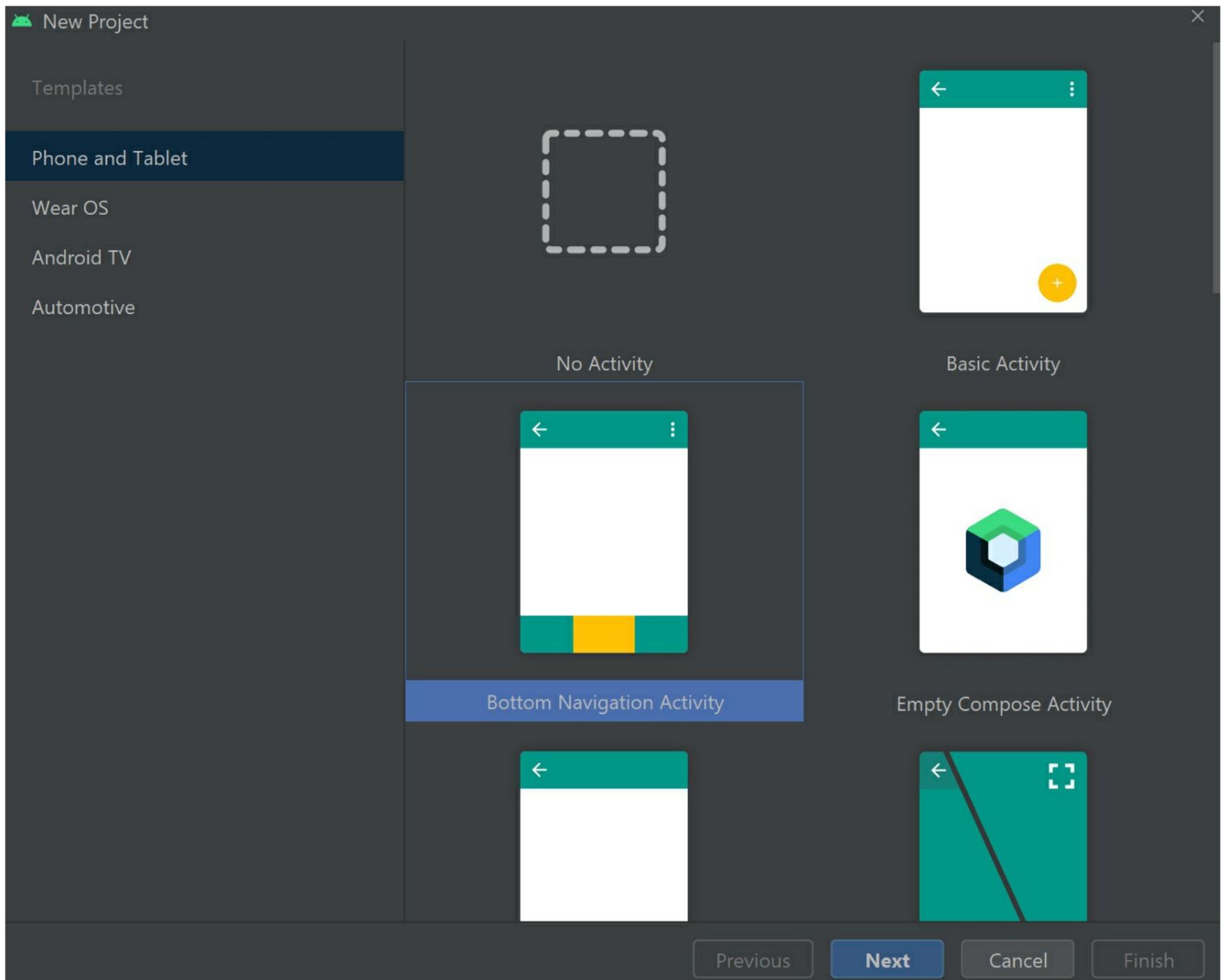
## Introduction

For this project, we will create an app that allows the user to capture photos using the camera on their device. The app will also display a gallery of all the images on their device and allow the user to apply filters or delete images if they wish. In creating this app, you will learn how to use the CameraX library to capture photos, apply an image rendering module called Glide to display and edit images and interact with files using the scoped storage framework. The scoped storage framework is a recent Android development designed to protect the user's privacy by limiting an app's access to files it did not create.



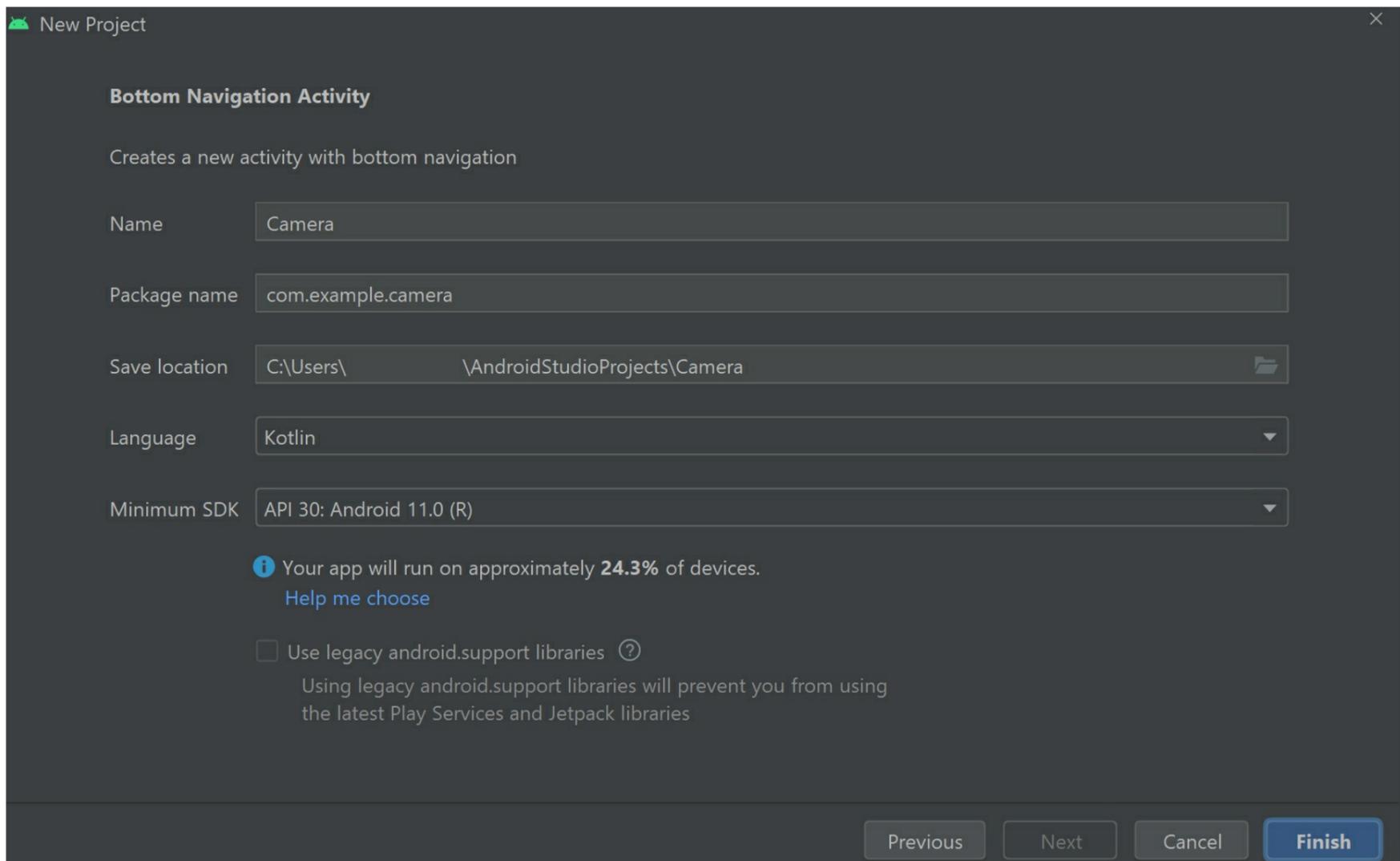
## Getting started

Let's begin. Open Android Studio and create a new project using the Bottom Navigation Activity project template.



The Bottom Navigation Activity project template provides your app with a navigation bar at the bottom of the screen, as well as several readymade fragments. Each fragment represents a different destination in the app, which the user will be able to navigate using the navigation bar. In this app, there will be separate fragments for the camera and image gallery.

In the Create New Project window, specify a name for the project (e.g. Camera), set the language to Kotlin and use API level 30.

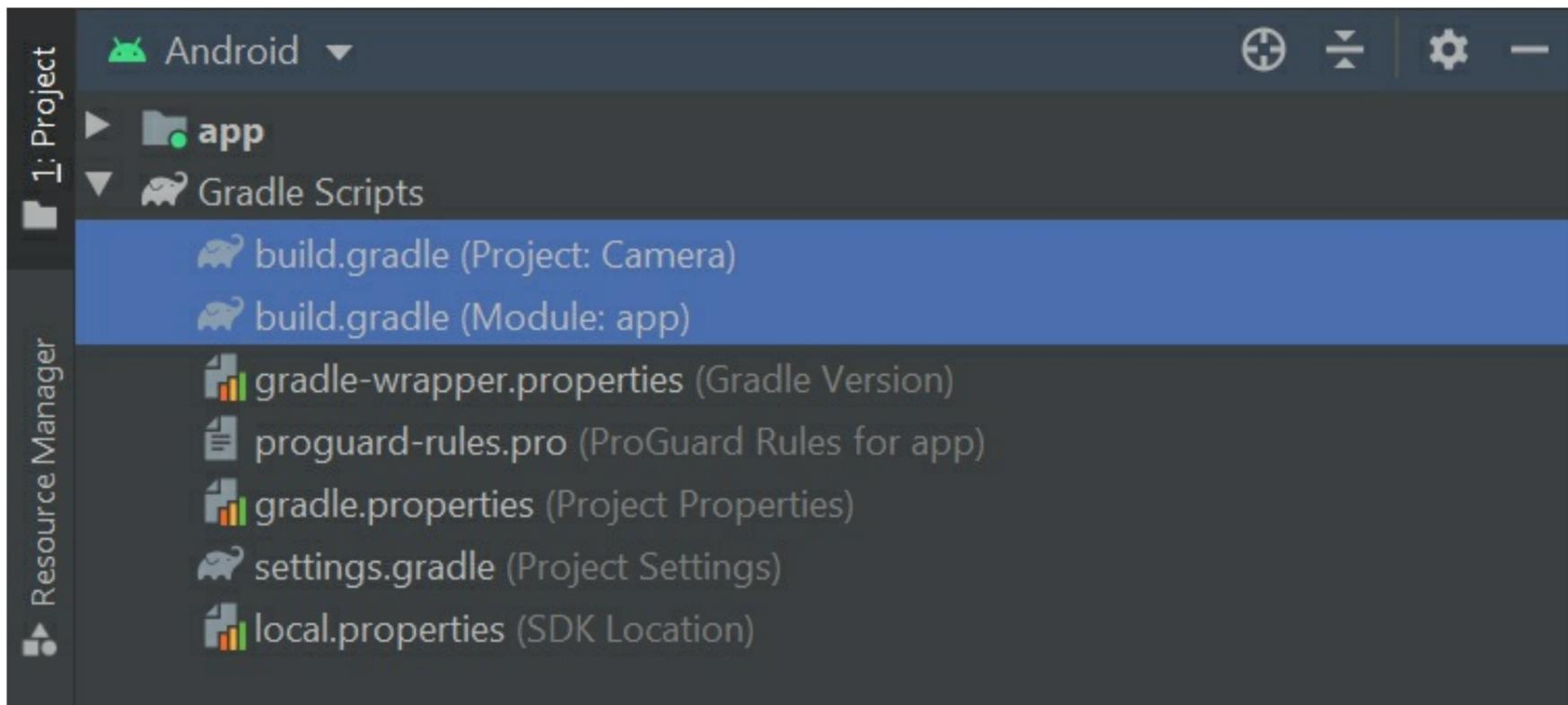


As recommended when creating the Notes app, you may like to enable Auto Imports to direct Android Studio to automatically add any necessary import statements to your Kotlin files as you code. These import statements are essential for incorporating the external classes and tools required for the app to run. To enable Auto Imports, open Android Studio's Settings window by clicking **File > Settings**. In the Settings window, navigate through **Editor > General > Auto Import** then select 'Add unambiguous imports on the fly' and 'Optimise imports on the fly' for both Java and Kotlin then press Apply and OK.

Android Studio should now add most of the necessary import statements to your Kotlin class files automatically. Sometimes there are multiple classes with the same name and the Auto Import feature will not work. In these instances, the requisite import statement(s) will be specified explicitly in the example code. You can also refer to the finished project code which accompanies this book to find the complete files including all import statements.

## Configuring the Gradle Scripts

For the app to perform all the operations we want it to, we must manually import several external packages using a toolkit called Gradle. To do this, navigate through **Project > Gradle Scripts** and open both the Project and Module **build.gradle** files:



In the Project-level **build.gradle** file, add the following classpath to the dependencies element:

```
classpath "androidx.navigation:navigation-safe-args-gradle-plugin:2.4.1"
```

The above classpath is required to use a feature called safe args, which is a method for transferring data between destinations in the app.

Next, switch to the Module-level **build.gradle** file and add the following lines to the plugins element at the top of the file:

```
id 'kotlin-parcelize'
id 'androidx.navigation.safeargs.kotlin'
```

Finally, refer to the dependencies element and add the following code to the list of implementation statements:

```
def archLifecycle_version = '2.2.0'

implementation "androidx.lifecycle:lifecycle-extensions:$archLifecycle_version"
implementation "androidx.lifecycle:lifecycle-common-java8:2.4.1"
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$archLifecycle_version"

implementation 'com.github.bumptech.glide:glide:4.11.0'
implementation 'jp.wasabeef:glide-transformations:3.3.0'
implementation 'jp.co.cyberagent.android.gpuimage:gpuimage-library:1.4.1'

def camerax_version = "1.1.0-beta01"
implementation "androidx.camera:camera-camera2:$camerax_version"
implementation "androidx.camera:camera-lifecycle:$camerax_version"
implementation "androidx.camera:camera-view:$camerax_version"
```

The above implementation statements enable your app to access several lifecycle tools, an image rendering tool called Glide, a Glide add-on package called Glide Transformations which we will use to apply filters to images (see: <https://github.com/wasabeef/glide-transformations>), and Android's CameraX library.

We're now finished with the Gradle Scripts files. Don't forget to re-sync your project when prompted!

Gradle files have changed since last project sync. A project sync may be necessary for the IDE ... [Sync Now](#)

Over time, you may notice some parts of your Gradle Scripts files are highlighted in yellow as follows:

```
implementation 'com.github.bumptech.glide:glide:4.9.0'
```

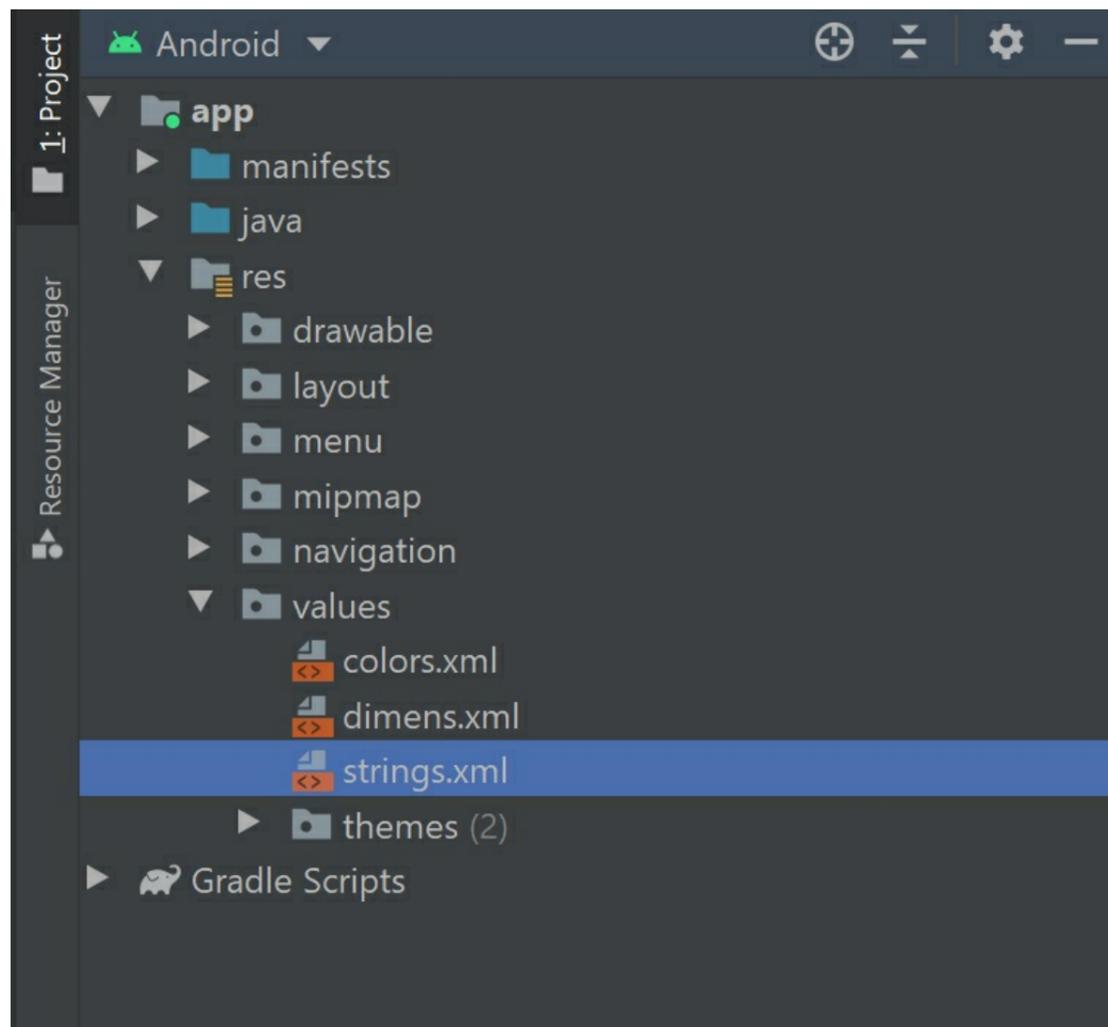
This often means the version number is outdated. If you hover over the highlighted text with your mouse then Android Studio will likely allow you to use the latest version number. It is usually fine to follow Android Studio's

recommendation then re-sync your Gradle files; however, you may want to double-check any affected features continue to work as expected.

```
implementation 'com.github.bumptech.glide:glide:4.9.0'  
// https://github.com/timusus/Re  
implementation 'com.simplecityap  
A newer version of com.github.bumptech.glide:glide than 4.9.0 is available: 4.11.0  
Change to 4.11.0 Alt+Shift+Enter More actions... Alt+Enter
```

## Defining the string resources used in the app

Each item of text that the app will display to the user should be stored as a string resource. A single string can be used in multiple locations across the app, which makes it easier to edit the text also because you only have to change one string resource, rather than each instance of the text throughout the app. Android Studio will often generate a resource file called **strings.xml** to store your strings when you create a new project. To locate the **strings.xml** file, navigate through **Project > app > res > values**.



Update the contents of the **strings.xml** file so it contains the following strings:

```
<resources>  
<string name="app_name">Camera</string>  
<string name="capture_photo">Capture a photo</string>  
<string name="delete_dialog_title">Delete?</string>  
<string name="delete_dialog_message">Delete %1$s?</string>  
<string name="delete_dialog_positive">Delete</string>  
<string name="delete_dialog_negative">Cancel</string>  
<string name="delete_image">Delete image</string>  
<string name="image">Image</string>  
<string name="ok">OK</string>  
<string name="save">Save</string>  
<string name="camera">Camera</string>  
<string name="gallery">Gallery</string>  
<string name="error_connecting_camera">There was an error connecting to the camera.</string>  
<string name="error_saving_photo">There was an error saving the photo.</string>  
<string name="photo_saved">Photo saved!</string>  
<string name="permission_required">The app requires camera and storage permissions to run.</string>
```

```
</resources>
```

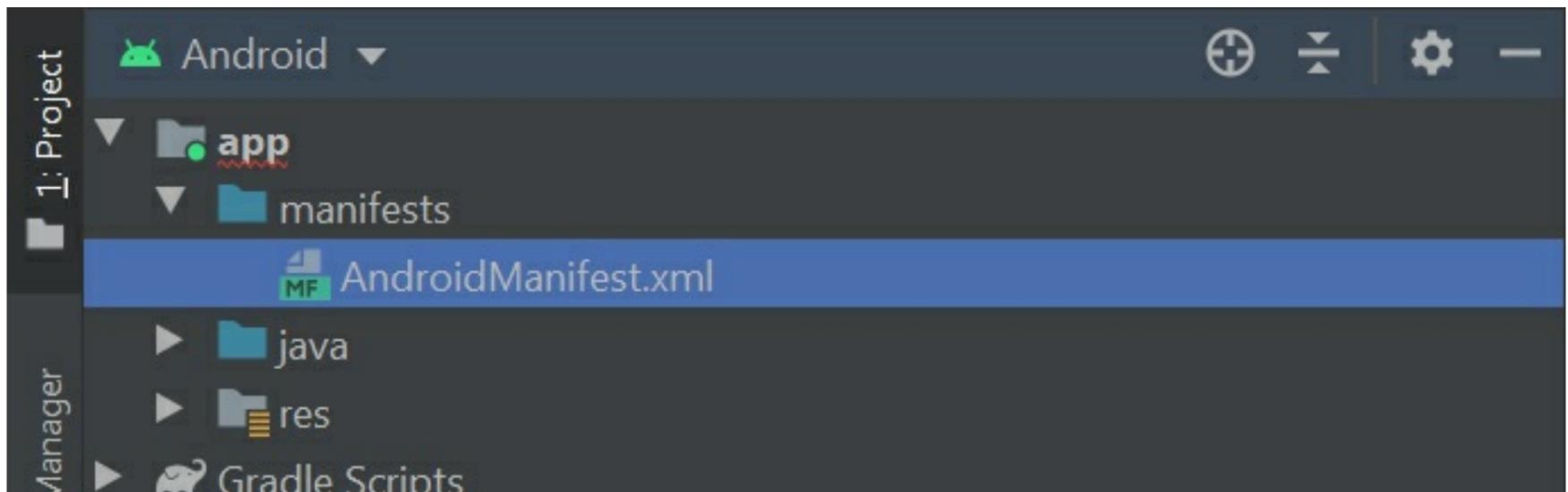
Each string resource contains a name attribute, which is what we will use to reference the string elsewhere in the app. The text that will be displayed to the user is input between the opening `<string name="">` and closing `</string>` tags. You may notice that the `delete_dialog_message` string contains the following code: `%1$s`. This code represents an argument that must be supplied when the string is created. The `'%1'` part represents the argument number, so will increase incrementally for each new argument that is added to the string e.g. `'%1'`, `'%2'`, `'%3'` etc. The second part of the argument indicates what data type is expected: `'$s'` represents a string, while `'$d'` represents a decimal integer and `'$.nf'` represents a floating number rounded to a certain number of decimal places (replace `'n'` with the number of decimal places e.g. `'$.2f'`). Returning to the `delete_dialog_message` string, we can see now it expects one string argument. You can use Kotlin code to build the string and supply the argument value(s):

```
getString(R.string.delete_dialog_message, "IMG_200222.png")
```

Supplying arguments allows you to create dynamic string messages. For example, the above Kotlin code imports the `delete_dialog_message` string and supplies `"IMG_200222.png"` as argument 1. The `delete_dialog_message` string incorporates the argument into the output text, so the final string would be `"Delete IMG_200222.png?"`.

## Requesting permission to access the device's camera and storage

The application will require permission from the user to access the device's cameras and files. All required permissions must be declared in the application's manifest file, which the Play store will use to inform potential user's what permissions they will need to provide if they install your app. To configure the manifest file, navigate through **Project > app > manifests** and open the file called **AndroidManifest.xml**.



Next, add the following code above the opening application element:

```
<uses-permission android:name="android.permission.CAMERA" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

The above items define each permission required by the app. In this case, the app will require access to the device's cameras and storage.

Let's now implement the code which prompts the user to grant the permissions. To handle all permission-related processes, we will create an object called `CameraPermissionHelper` in the `MainActivity` class. The `CameraPermissionHelper` object will contain methods for initiating and handling permissions-related requests. Locate and open the **MainActivity.kt** file by navigating through **Project > app > java > name of the project**. Next, add the following code below the `onCreate` method to define the `CameraPermissionHelper` object:

```
object CameraPermissionHelper {
    private const val CAMERA_PERMISSION = Manifest.permission.CAMERA
    private const val READ_PERMISSION = Manifest.permission.READ_EXTERNAL_STORAGE

    fun hasCameraPermission(activity: Activity): Boolean {
        return ContextCompat.checkSelfPermission(activity, CAMERA_PERMISSION) ==
            PackageManager.PERMISSION_GRANTED
    }

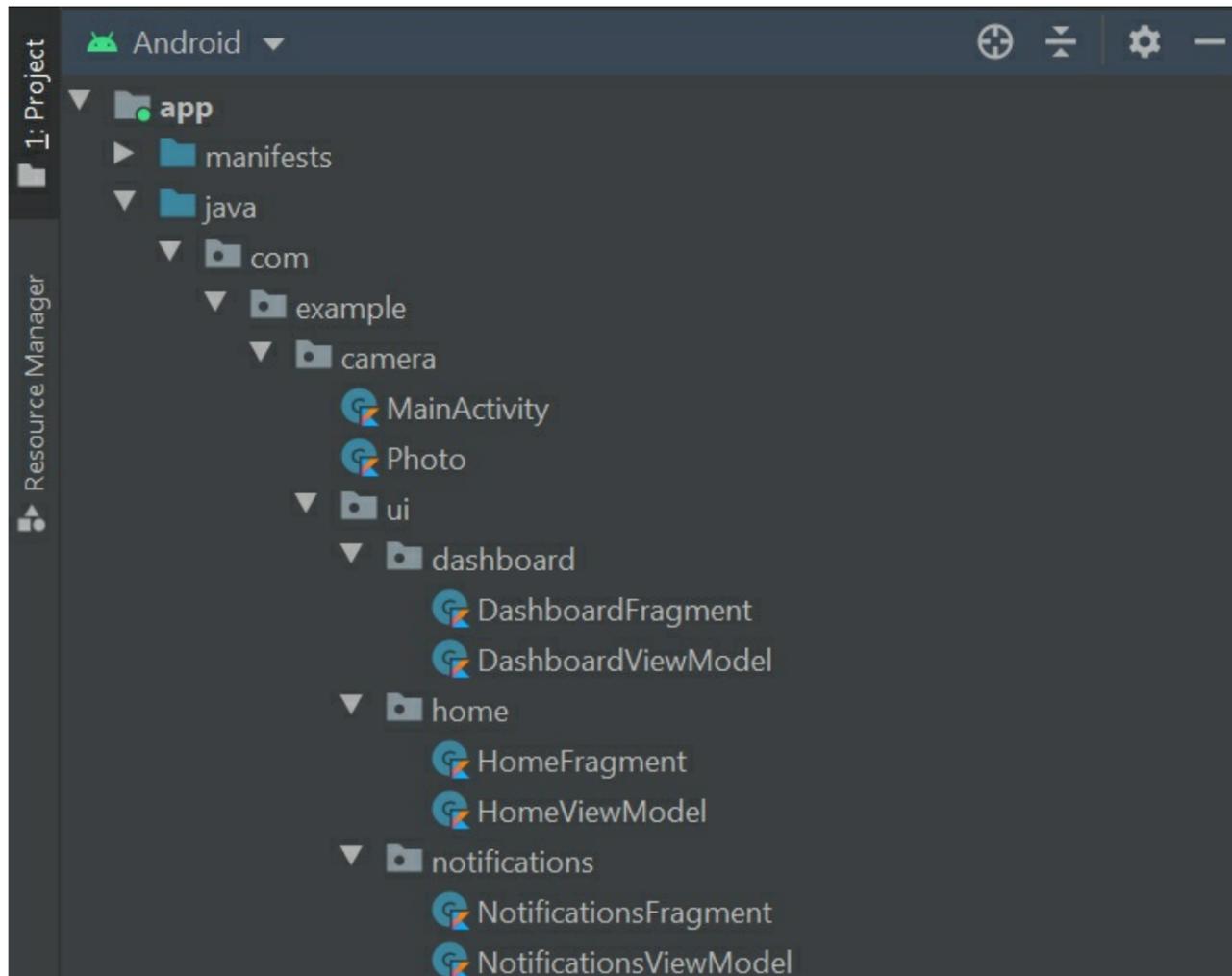
    fun hasStoragePermission(activity: Activity): Boolean {
        return ContextCompat.checkSelfPermission(activity, READ_PERMISSION) ==
            PackageManager.PERMISSION_GRANTED
    }
}
```



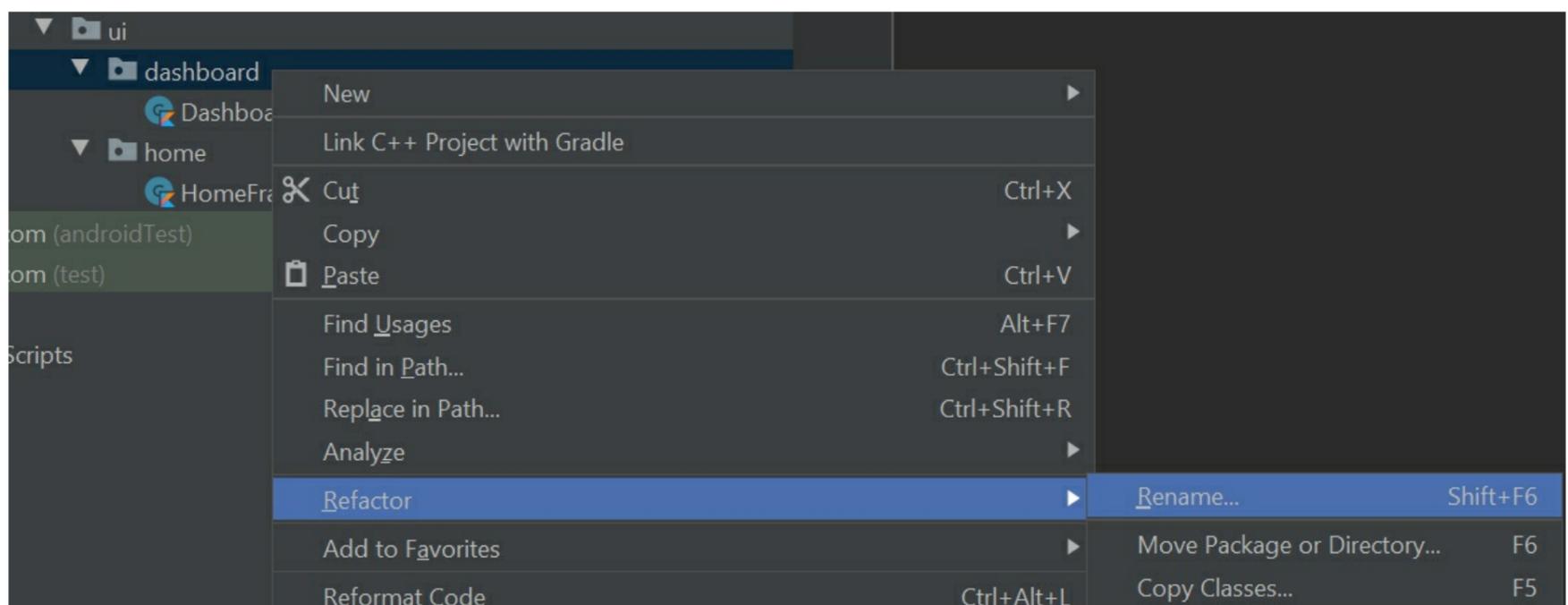
check whether all required permissions have been granted. If either method returns a value of false, then this means the user has refused to grant permission. Consequently, the app will run the CameraPermissionHelper object's requestPermissions method again and show the rationale for the permission request. If the hasCameraPermission and hasStoragePermission methods both return values of true, then the recreate method will reload the activity because the necessary user permissions have been granted.

## Setting up the Camera fragment and layout

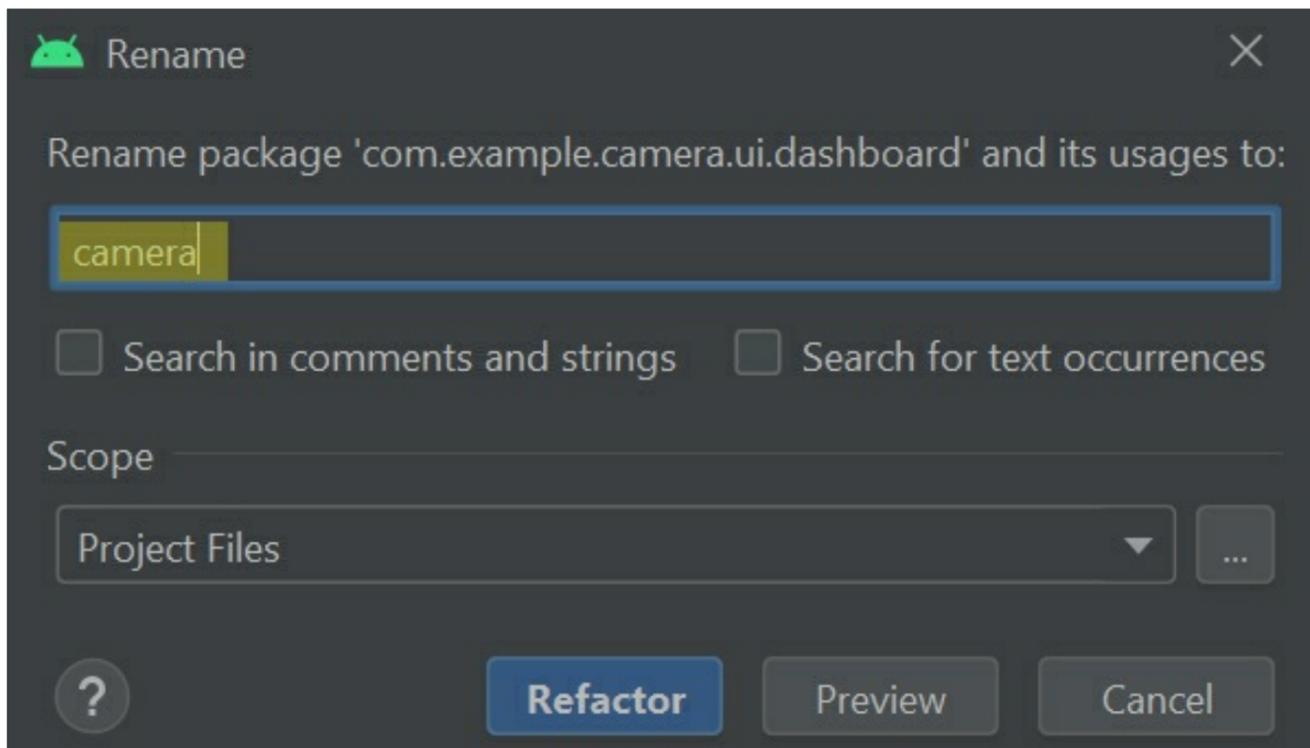
In this section, we will create a fragment that opens the rear-facing camera on the user's device and allows them to take photos. Android Studio automatically generated three fragments when the project was created. You can locate them by navigating through **Project > app > java > name of project > ui**



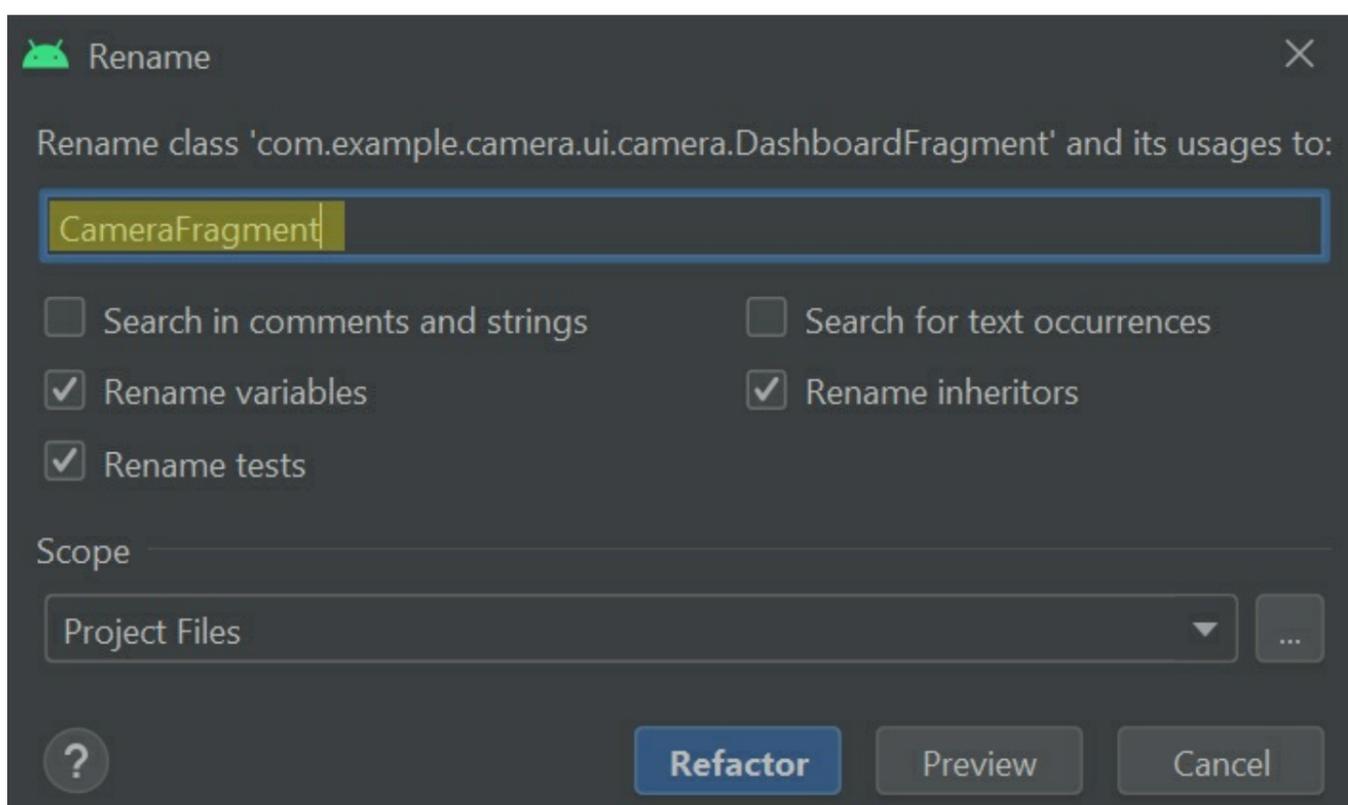
For this app, we will only require two fragments, so right-click one of the fragment folders (e.g. notifications) and press Delete. We will also not require the view model's for the two remaining fragments, so right-click the **DashboardViewModel.kt** and **HomeViewModel.kt** files and press Delete. Finally, we must rename the remaining fragment files. Android Studio provides a refactor option that allows you to rename an item (e.g. a file, variable or class) and automatically update all other areas in the app which refer to the item. In this way, you can change the name of application components without creating errors in your code. To refactor the Dashboard package, right-click the dashboard folder then press **Refactor > Rename**.



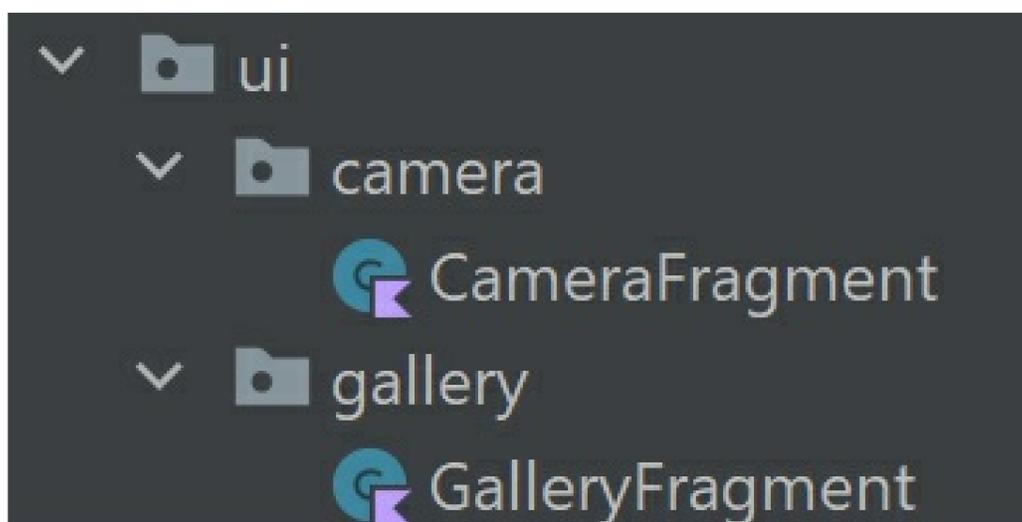
Set the name to camera then press Refactor.



Next, right-click the DashboardFragment file and again select **Refactor** > **Rename**. Set the new name to CameraFragment then press Refactor.

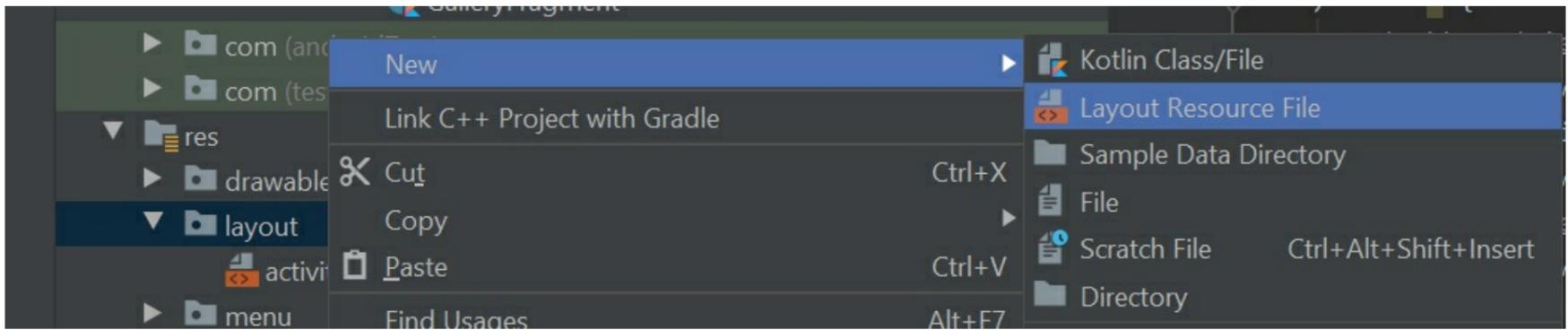


The camera fragment and directory have now been successfully refactored. At this point, it may be a good opportunity to also prepare the image gallery package and fragment. To do this, refactor the **home** directory to **gallery** and refactor the HomeFragment file to GalleryFragment.



The camera fragment will require a layout file to display a live feed of the camera's input and allow the user to capture photos. To implement this, locate the **layout** folder by navigating through **Project** > **app** > **res**. The readymade **fragment\_dashboard.xml**, **fragment\_home.xml** and **fragment\_dashboard.xml** files can be deleted because we will not use them. Next, create a new layout file by right-clicking the **layout** folder and selecting **New** >

## Layout Resource File.



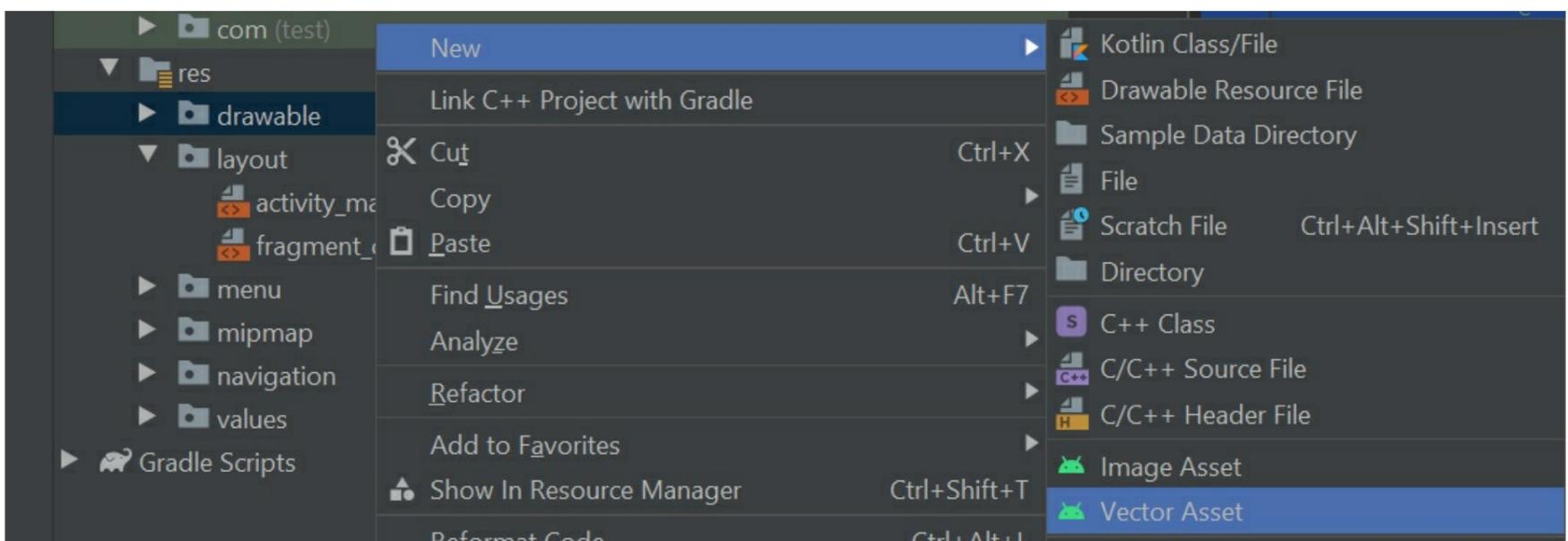
Name the file `fragment_camera` then press OK. Once the `fragment_camera.xml` file opens in the editor, switch the layout to Code view and edit its contents as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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">

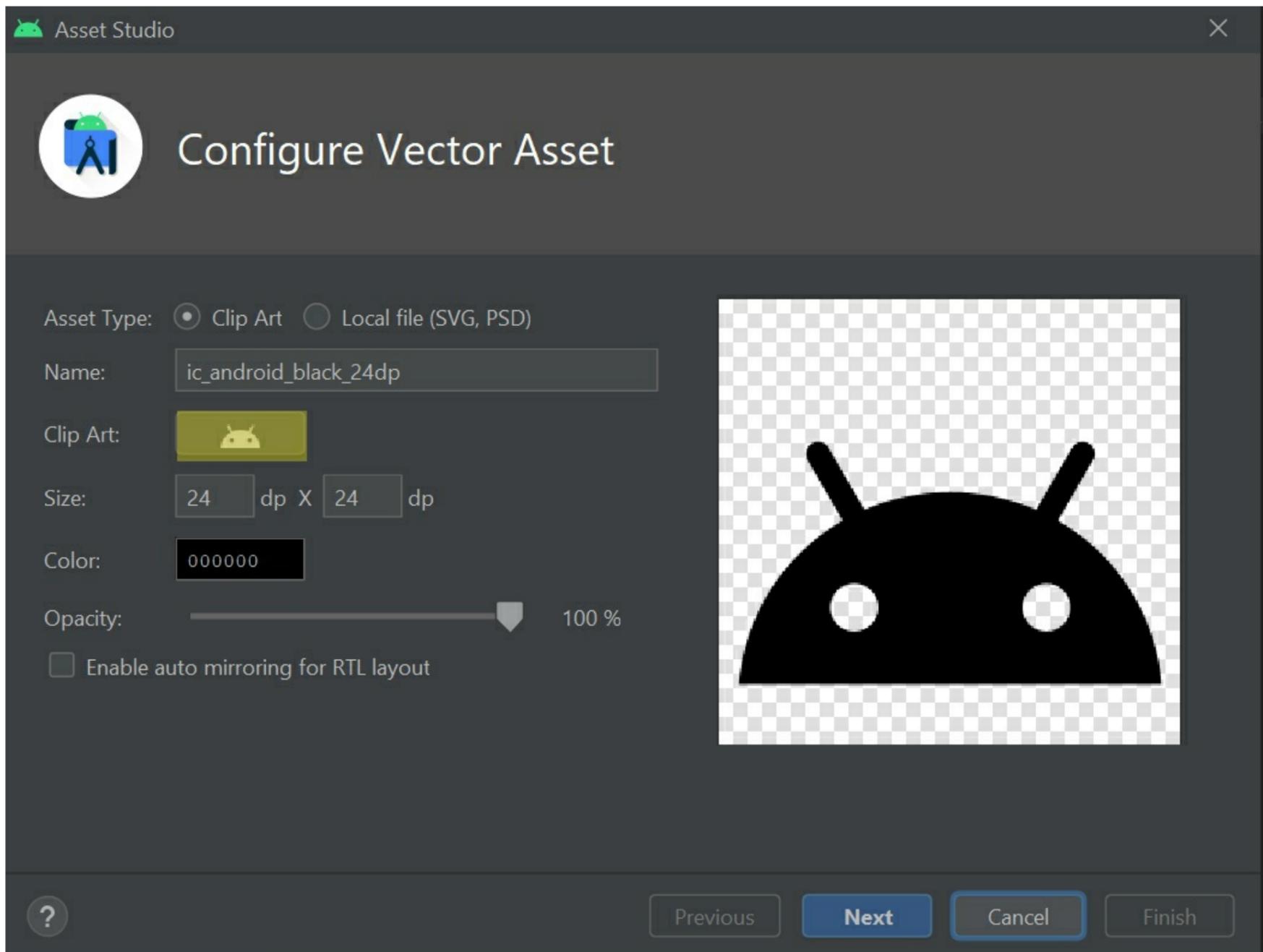
    <androidx.camera.view.PreviewView
        android:id="@+id/camera_feed"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/fab_take_photo"
        android:layout_width="56dp"
        android:layout_height="56dp"
        android:layout_marginBottom="24dp"
        android:src="@drawable/ic_camera"
        android:contentDescription="@string/capture_photo"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintBottom_toBottomOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

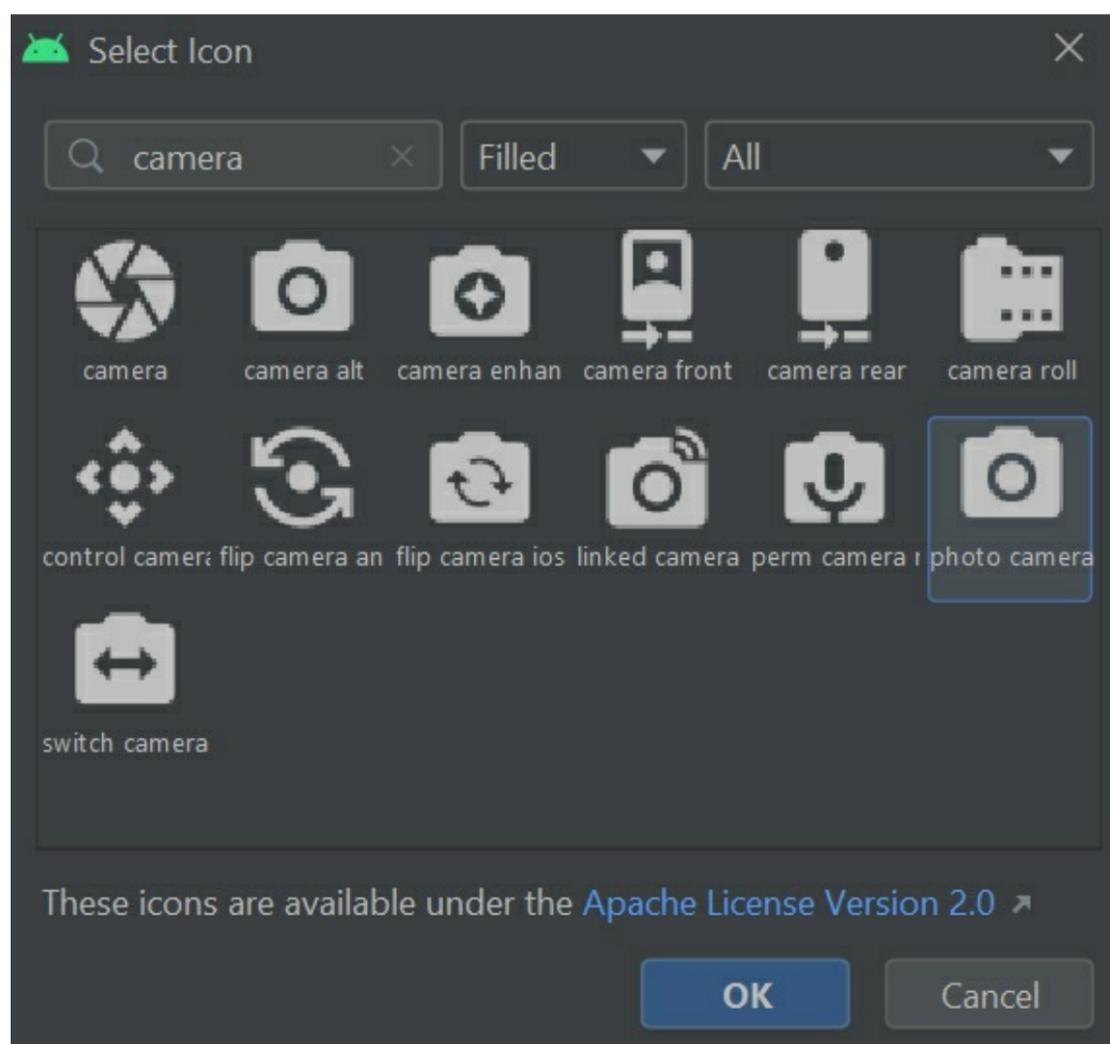
The above code begins by adding a `PreviewView` widget to the layout. The `PreviewView` widget is specially designed for displaying camera feeds using the `CameraX` library. It provides a surface for the camera feed to be drawn on and automatically adjusts to different camera properties such as the aspect ratio, scale and rotation. The layout also contains a `FloatingActionButton` widget, which will allow the user to capture photos. The `FloatingActionButton` will display a camera icon to indicate the role of the button. To create the camera icon, right-click the **drawable** folder (which is above the **layout** folder) then select **New > Vector Asset**.



In the Asset Studio window, click the image of the Android next to the phrase `Clip Art` to open a window called `Select Icon`.



In the Select Icon window, search for and select the photo camera icon then press OK.



When you return to the Asset Studio window, set the name to `ic_camera` then press Next followed by Finish to save the icon. The layout for the camera fragment is now complete and ready for integration with the CameraFragment class. To do this, open the **CameraFragment.kt** file (**Project > app > java > name of the project > ui > camera**)

and edit the `_binding` variable definition so it reads as follows:

```
private var _binding: FragmentCameraBinding? = null
```

Also, delete the `dashboardViewModel` variable shown below. This variable was generated by Android Studio but will not be necessary for the Camera project.

```
private lateinit var dashboardViewModel: DashboardViewModel
```

To initialise the `_binding` variable, edit the `onCreateView` method so it reads as follows:

```
override fun onCreateView(  
    inflater: LayoutInflater,  
    container: ViewGroup?,  
    savedInstanceState: Bundle?  
): View {  
    _binding = FragmentCameraBinding.inflate(inflater, container, false)  
  
    return binding.root  
}
```

The above code initialises the `fragment_camera.xml` layout's binding class and allows the fragment to interact with the layout and its widgets.

## Connecting to a camera and displaying a live feed

In this section, we'll configure the camera fragment to capture photos using Android's CameraX library. First, add the following variable below the binding variables at the top of the `CameraFragment` class:

```
private lateinit var cameraExecutor: ExecutorService
```

The `cameraExecutor` variable defined above will provide access to an instance of the `ExecutorService` class once initialised. The `ExecutorService` class is used to coordinate tasks. We will use the `ExecutorService` instance to manage actions relating to the camera, such as capturing photos. To initialise the `cameraExecutor` variable and load the camera feed, add the following code below the `onCreateView` method:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
    super.onViewCreated(view, savedInstanceState)  
  
    cameraExecutor = Executors.newSingleThreadExecutor()  
    openCamera()  
}
```

Note you may need to import the `ExecutorService` and `Executors` classes manually by adding the following import statements to the top of the file:

```
import java.util.concurrent.ExecutorService  
import java.util.concurrent.Executors
```

Moving on, we'll now write a method called `openCamera`. The `openCamera` method will establish a connection with the camera and display a live camera feed. To define the method, add the following code below the `onViewCreated` method:

```
private fun openCamera() {  
    if (MainActivity.CameraPermissionHelper.hasCameraPermission(requireActivity()) &&  
        MainActivity.CameraPermissionHelper.hasStoragePermission(requireActivity())) {  
        val cameraProviderFuture = ProcessCameraProvider.getInstance(requireActivity())  
  
        cameraProviderFuture.addListener({  
            val cameraProvider = cameraProviderFuture.get()  
            val preview = Preview.Builder()  
                .build()  
                .also {  
                    it.setSurfaceProvider(binding.cameraFeed.surfaceProvider)  
                }  
  
            val cameraSelector = CameraSelector.DEFAULT_BACK_CAMERA
```

```
// TODO: Initialise the ImageCapture instance here
```

```
try {  
    cameraProvider.unbindAll()  
    cameraProvider.bindToLifecycle(this, cameraSelector, preview)  
} catch (e: IllegalStateException) {  
    Toast.makeText(requireActivity(), resources.getString(R.string.error_connecting_camera),  
Toast.LENGTH_LONG).show()  
}  
}, ContextCompat.getMainExecutor(requireActivity()))  
} else MainActivity.CameraPermissionHelper.requestPermissions(requireActivity())  
}
```

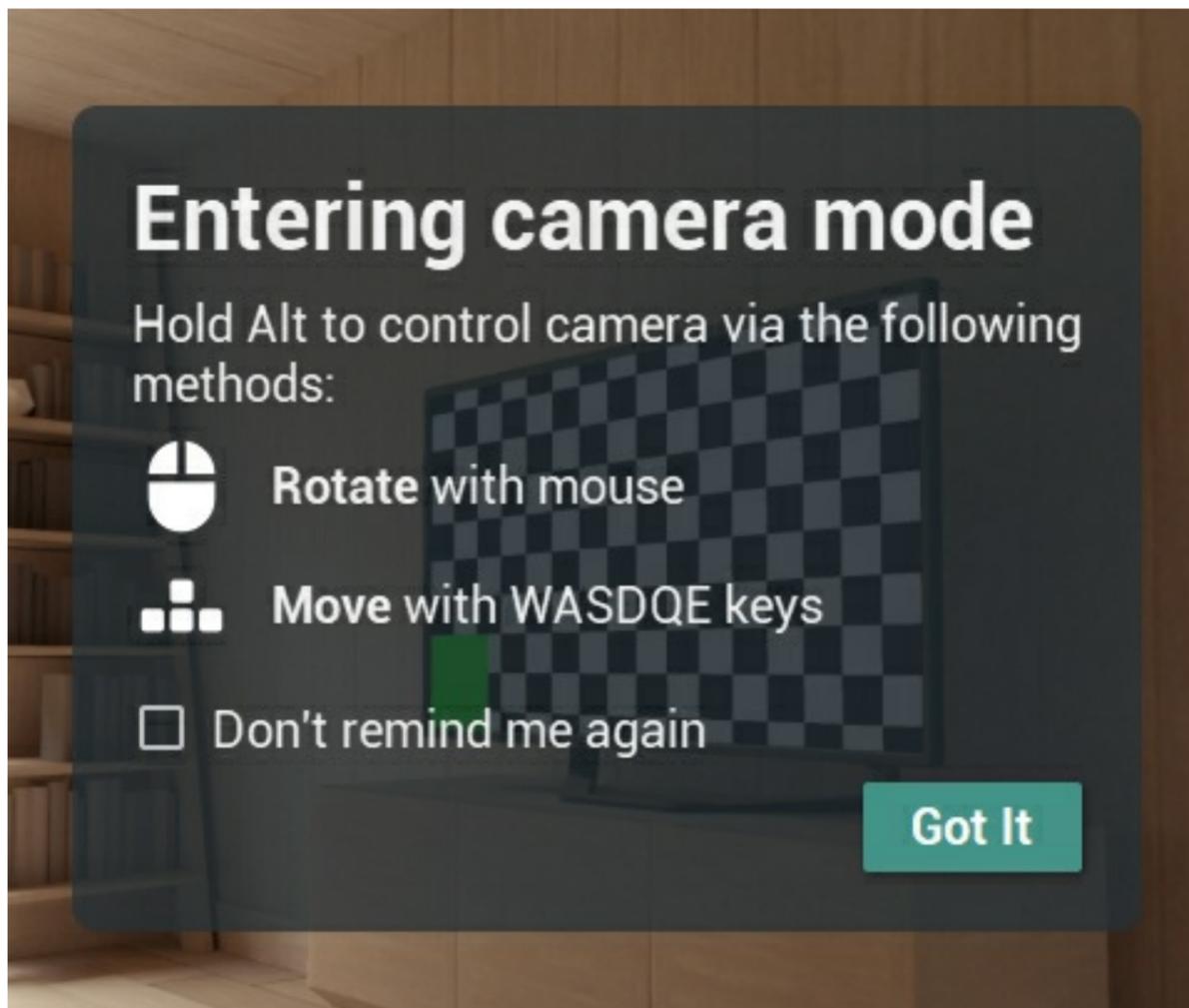
The openCamera method begins by checking whether the app has permission to access the camera(s) and storage on the user's device. If the app does not have permission, then permission is requested using the CameraPermissionHelper object found in the MainActivity class. Otherwise, the method proceeds to create an instance of the ProcessCameraProvider class. The ProcessCameraProvider instance will bind the device's cameras with the app so that the app can interact with the cameras.

Once the ProcessCameraProvider instance is ready, the listener we register above will be notified and begin to execute tasks with the assistance of the ExecutorService class. First, we initialise the PreviewView widget's surface provider. The surface provider will prepare a Surface interface for displaying a live preview feed from the camera. Next, we specify the camera that we would like to use. By default, the app will use the rear-facing camera on the back of the user's device. You could use the front-facing camera instead by replacing DEFAULT\_BACK\_CAMERA with DEFAULT\_FRONT\_CAMERA.

Finally, any currently active camera sessions are closed using the ProcessCameraProvider class's unbindAll command. This frees up the camera for our app to initiate a new binding session and begin displaying a live feed from the rear-facing camera to the PreviewView widget. These latter two operations are wrapped in a try/catch block to intercept any illegal state exceptions that may occur. For example, the bindToLifecycle method can throw an illegal state exception if the camera is already bound to another app or process. You can find all possible exceptions associated with the different methods by referring to the official Android documentation. For instance, the documentation for the bindToLifecycle method can be found here: [https://developer.android.com/reference/androidx/camera/lifecycle/ProcessCameraProvider#public-methods\\_1](https://developer.android.com/reference/androidx/camera/lifecycle/ProcessCameraProvider#public-methods_1). If an illegal state exception is thrown, then the above code will display a toast notification advising that there was an error connecting to the camera.



You can test the camera functionality on your computer using Android Studio's virtual device emulator. For instructions on how to set up a virtual device, refer to the 'Testing an application on a virtual device' section at the beginning of the book. To control the virtual device's camera, hold the Alt key on your computer keyboard and move your mouse to look around the virtual room. You can also move around the virtual room by pressing the WASDQE keyboard keys. To capture a photo, click the floating action button that we added to the bottom of the `fragment_camera.xml` layout.



## Capturing photos

In this section, we will explore how to use the camera to capture photos. The application will capture a photo when the user clicks the `FloatingActionButton` widget from the `fragment_camera.xml` layout. To configure this feature, add the following variable to the list of variables at the top of the `CameraFragment` class:

```
private lateinit var imageCapture: ImageCapture
```

The variable defined above will store an instance of the `ImageCapture` class. The `ImageCapture` class handles actions relating to capturing photos using the `CameraX` library. We will want to initialise the `ImageCapture` instance once the app has successfully connected to a camera. To do this, locate the `openCamera` method and replace the `TODO` comment with the following code:

```
imageCapture = ImageCapture.Builder()  
    .setCaptureMode(ImageCapture.CAPTURE_MODE_MINIMIZE_LATENCY)  
    .build()
```

The above code builds an instance of the `ImageCapture` class and sets the capture mode of the instance to `CAPTURE_MODE_MINIMIZE_LATENCY`. The `CAPTURE_MODE_MINIMIZE_LATENCY` capture mode will minimise the capture latency and ensure photos are taken as quickly as possible. If you would prefer to prioritise image quality instead of capture speed, then you could use the capture mode `CAPTURE_MODE_MAXIMIZE_QUALITY` instead.

Similar to the camera feed preview, the image capture feature is another `CameraX` use case. For this reason, we must bind the `ImageCapture` instance to the `CameraX` lifecycle so it can be linked with the camera state. To do this, locate the section of the `openCamera` method where we run the `bindToLifecycle` method. Edit that line of code so it reads as follows to bind the `ImageCapture` instance to the `CameraX` lifecycle:

```
cameraProvider.bindToLifecycle(this, cameraSelector, preview, imageCapture)
```

To initiate capture requests when the floating action button is clicked, add the following code to the `onViewCreated` method:

```
binding.fabTakePhoto.setOnClickListener {  
    capturePhoto()  
}
```

The above code registers an `onClick` listener to the `FloatingActionButton` widget, which runs a method called `capturePhoto` whenever the button is clicked. The `capturePhoto` method will capture a freeze-frame of the camera

feed and save the resulting image to the user's device. To define the capturePhoto method, add the following code below the openCamera method:

```
private fun capturePhoto() {
    if (!this::imageCapture.isInitialized) {
        Toast.makeText(requireActivity(), resources.getString(R.string.error_saving_photo),
            Toast.LENGTH_LONG).show()
        return
    }

    val contentValues = (activity as MainActivity).prepareContentValues()

    val outputFileOptions = ImageCapture.OutputFileOptions.Builder(
        requireActivity().applicationContext.contentResolver,
        MediaStore.Images.Media.EXTERNAL_CONTENT_URI,
        contentValues).build()

    imageCapture.takePicture(
        outputFileOptions, ContextCompat.getMainExecutor(requireActivity()), object :
        ImageCapture.OnImageSavedCallback {
            override fun onError(exc: ImageCaptureException) {
                Toast.makeText(requireActivity(), resources.getString(R.string.error_saving_photo),
                    Toast.LENGTH_LONG).show()
            }

            override fun onImageSaved(output: ImageCapture.OutputFileResults) {
                Toast.makeText(requireActivity(), resources.getString(R.string.photo_saved),
                    Toast.LENGTH_LONG).show()
            }
        })
}
```

The capturePhoto method begins by confirming that the imageCapture variable has been initialised. Remember, the imageCapture variable is initialised by the openCamera method once it successfully connects to the camera. If the user clicks the floating action button before a connection to the camera has been established, then the imageCapture variable will not be initialised and attempts to use it could cause the app to crash. For this reason, if the imageCapture variable is not initialised, then the method will display a toast notification advising that the app was unable to capture a photo. The return command will then exit the method without executing any further code.

Providing the imageCapture variable is initialised, the method will save the captured photo as an image file. The image file will require content values, which define data such as the file's name and extension. The content values will be prepared by a MainActivity method that we will write shortly called prepareContentValues. The methodology for defining the content values is delegated to the MainActivity class so it can also be accessed by another fragment that we will create later. In defining the method in a place that can be accessed by both fragments, we help avoid the repetition of code (also known as boilerplate code), which is good software development practice.

Once all the content values have been prepared, they are packaged into an instance of the OutputFileOptions class. The ImageCapture instance will use the specifications detailed in the OutputFileOptions object to write the image file to the device's storage and media store. The media store is a collection of tables detailing the different collections of media on the user's device. For example, there are separate mediastore tables for music, images and videos. Each entry in the mediastore table is identified by a URI, which serves as a reference to the media item's location on the device.

Finally, an image is captured from the camera using the ImageCapture class's takePicture method and saved to the user's device. The takePhoto method has two callback methods called onError and onImageSaved. The onError method will run if the app is unable to save the image, while the onImageSaved method will run if the image is saved successfully. In either case, the above code will display a toast notification advising the user accordingly. You can edit the above code if you would rather respond to successful and failed image capture attempts in a different way.

The image will be saved in PNG format to the DCIM folder on the user's device, as specified in the image file's content values. To assign content values to an image, we will write a method in the MainActivity class called prepareContentValues. Open the **MainActivity.kt** file (**Project > app > java > name of the project**) and add the following code below the onRequestPermissionsResult function:

```

fun prepareContentValues(): ContentValues {
    val timeStamp = SimpleDateFormat("yyyyMMdd_HHmmss", Locale.getDefault()).format(Date())
    val imageFileName = "image_ $timeStamp"

    return ContentValues().apply {
        put(MediaStore.MediaColumns.DISPLAY_NAME, imageFileName)
        put(MediaStore.MediaColumns.MIME_TYPE, "image/png")
        put(MediaStore.MediaColumns.RELATIVE_PATH, "DCIM")
    }
}

```

Note you may need to add the following import statement to the top of the file:

```
import java.text.SimpleDateFormat
```

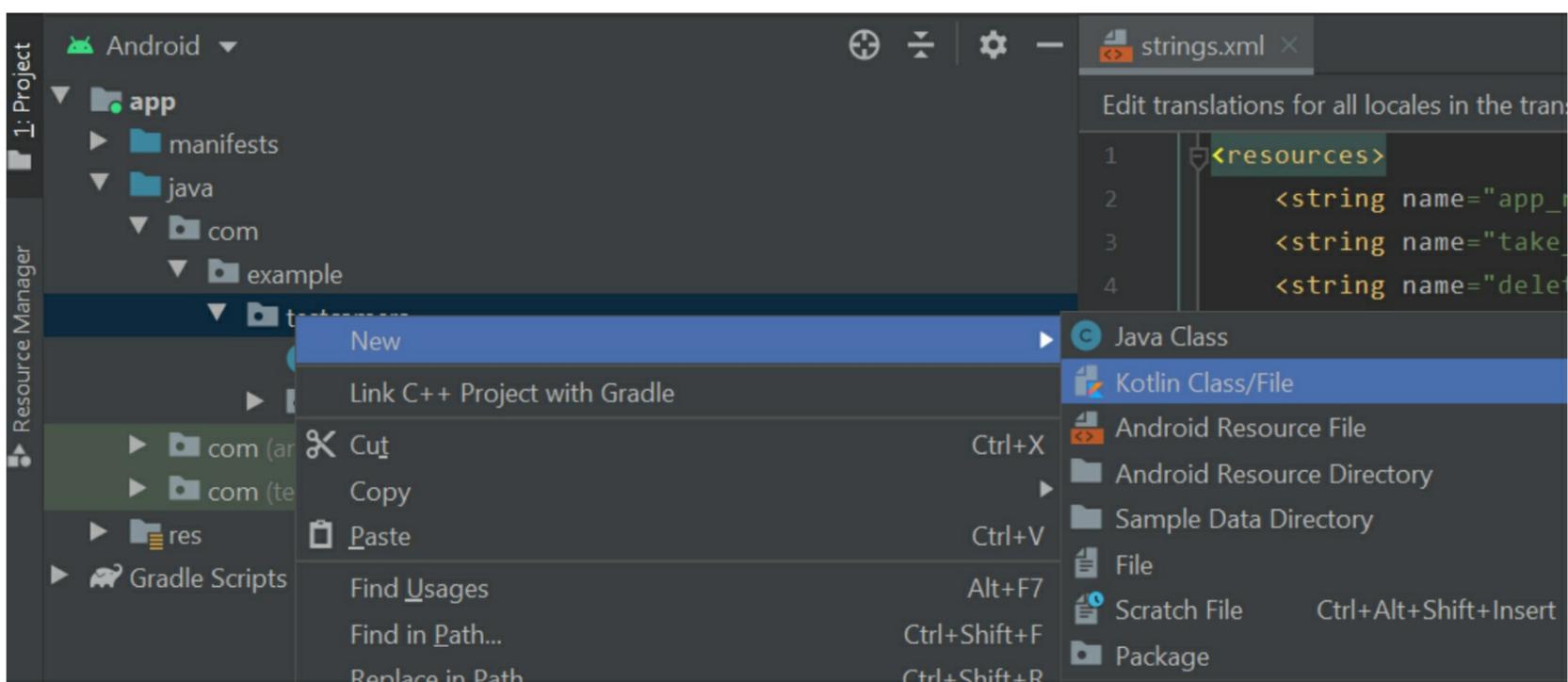
In the above code, content values are defined for the image’s display name, MIME type (the file format) and relative file path (DCIM directory). The file name is set to “image\_” followed by a timestamp of when the photo was taken. The timestamp is determined using the SimpleDateFormat class, which uses the Date class to retrieve the current date and time (to the nearest millisecond) and formats the timestamp based on the structure “yyyyMMdd\_HHmmss”. For example, if the current time was 20 seconds past 11:34 AM on January 21st 2022 then the SimpleDateFormat class will output “20220121\_113420”. The imageFileName variable incorporates the timestamp into the complete filename e.g. “image\_20220121\_113420”.

If you wish to provide further information about the image then you can use put commands as shown above. The put command requires you to specify a key, which identifies the content value, and the data associated with that key. For a list of possible keys then you should refer to the official Android documentation:

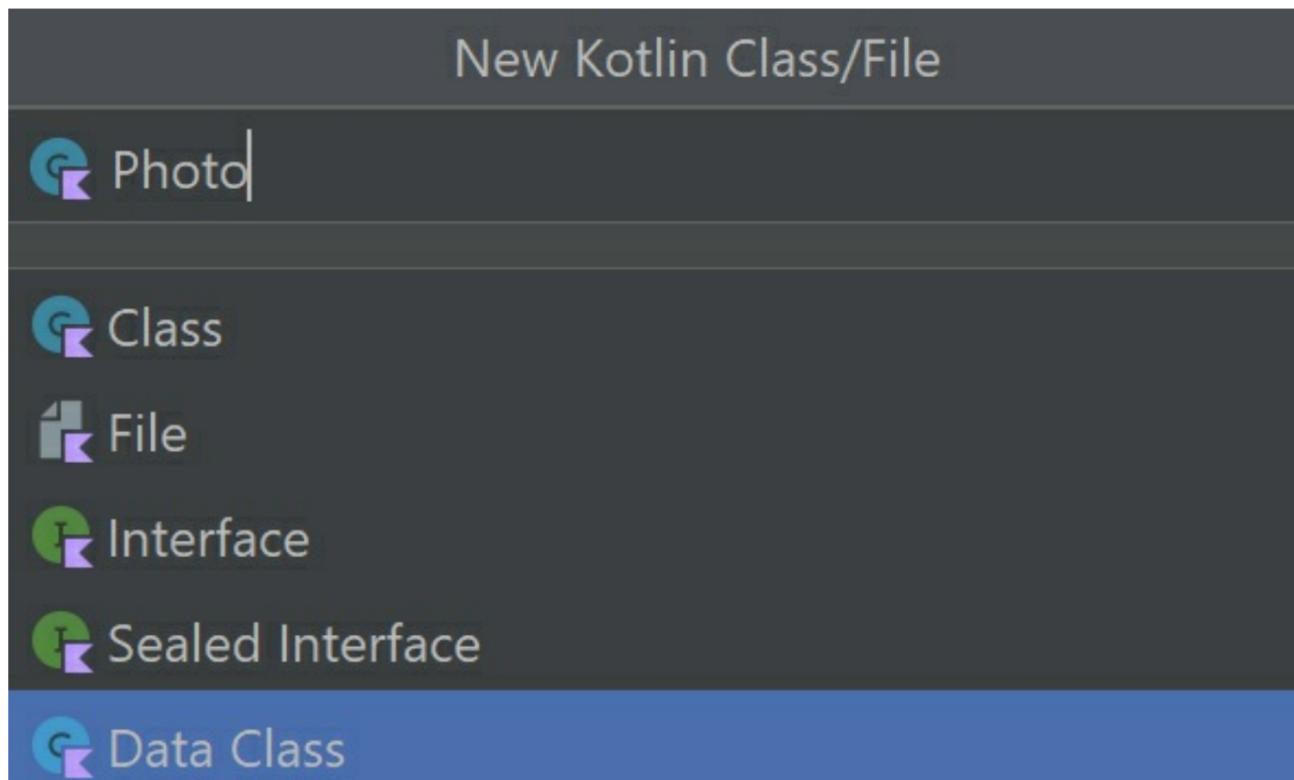
<https://developer.android.com/reference/android/provider/MediaStore.MediaColumns>

## Creating the Photo data class

Besides capturing photos, the Camera app will also allow the user to view all the images on their device in a gallery. For processing purposes, information about each image will be packaged in a Kotlin data class. To create the data class, navigate through **Project > app > java** and right-click the folder with the project’s name. Next, select **New > Kotlin File/Class**.



Name the file Photo and select Data Class from the list of options.



A file called **Photo.kt** should then open in the editor. Modify the Photo class code so it reads as follows:

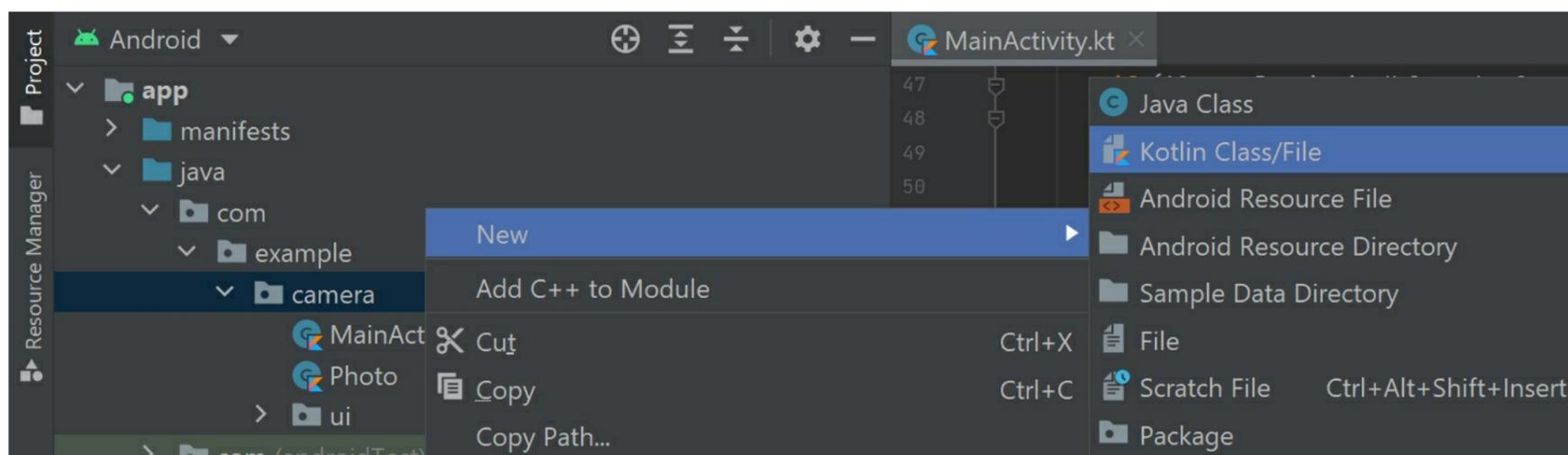
```
import android.net.Uri
import android.os.Parcelable
import kotlinx.parcelize.Parcelize

@Parcelize
data class Photo(val id: Long,
                val uri: Uri) : Parcelable
```

The Photo class defined above is labelled with the `@Parcelize` annotation and extends the Parcelable interface. Labelling the data class as parcelable means it can be packaged in a Parcel object for transportation between different areas of the app. The primary constructor of the data class contains two parameters, each storing a different piece of information. The id variable contains a Long numerical value which is unique to each Photo object, while the uri variable will contain a URI object that details the location of the image file.

## Loading images from the user's device

The image gallery will be coordinated by a view model. View models provide a way for applications to compartmentalise backend processes. For example, the view model we create here will load images from the device and handle requests to delete images. To create the class that will power the view model, right-click the folder with the name of the project (**Project** > **app** > **java**) then press **New** > **Kotlin Class/File**.



Name the class GalleryViewModel and select Class from the list of options. Once the **GalleryViewModel.kt** file opens in the editor, modify the contents of the class so it reads as follows:

```
class GalleryViewModel(application: Application) : AndroidViewModel(application) {

    private val appContext: Application = application
    private var contentObserver: ContentObserver
    val photos = MutableLiveData<List<Photo>>()
```

```
}
```

The GalleryViewModel class will inherit all the data and methods from the AndroidViewModel class. The AndroidViewModel class is a subclass of the ViewModel class; it incorporates all the functionality of the ViewModel class while also providing access to the application context. In the above code, the application context is stored in a variable called appContext so it can be used to access content on the device such as images. There is also a variable called contentObserver, which will monitor changes to the device's content, and a variable called photos, which will contain a list of Photo objects representing the images stored on the user's device. The list of Photo objects is classified as MutableLiveData. MutableLiveData can be observed by other areas of the app, which means those areas will be notified whenever the list of photos changes.

Let's now make the view model operational. To do this, add the following code below the list of variables:

```
fun loadPhotos() = viewModelScope.launch(Dispatchers.IO) {
    val projection = arrayOf(MediaStore.Images.Media._ID)
    val selection = MediaStore.Images.Media.DATE_ADDED
    val sortOrder = "${MediaStore.Images.Media.DATE_ADDED} DESC"

    appContext.contentResolver.query(
        MediaStore.Images.Media.EXTERNAL_CONTENT_URI,
        projection,
        selection,
        null,
        sortOrder
   )?.use { cursor ->
        photos.postValue(addPhotosFromCursor(cursor))
    }
}
```

Note you may need to add the following import statements to the top of the file:

```
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch
```

The loadPhotos method will scan the user's device for photos. This could be an energy-intensive and lengthy process and so we must consider what the optimum method for completing the task is. To complete complex tasks in a resource-efficient manner, we can use a Kotlin feature called coroutines. Coroutines provide a mechanism for completing complex tasks behind the scenes by allocating tasks to alternative worker threads. By default, most tasks will be completed on the main thread of the app; however, if too much work is performed on the main thread it can cause the application to slow down and freeze. For this reason, it is advantageous to use coroutines to complete complex tasks without compromising performance.

The loadPhotos method is launched using a view model scope, which is a custom coroutine scope owned by the view model. If the view model is closed, then any outstanding tasks launched via the view model scope will be cancelled. The coroutine defined above also utilises the IO dispatcher. The IO dispatcher directs tasks to a pool of threads geared towards handling data input and output, which is perfect for the loadPhotos method because it will be processing image files.

The loadPhotos method will locate all the images on the user's device via a content query. The content query comprises multiple components. First, we must set the projection, which defines the columns that should be included in the table of results. In the above code, the projection will return only the ID of each image because this is all we need to locate the image's file. You could request additional data such as the display name of the image and the date it was taken. For more information on this then refer to the Android documentation to find the column names <https://developer.android.com/reference/kotlin/android/provider/MediaStore.MediaColumns>. The content query also includes a selection argument, which will filter the results. In the above code, the selection criterion will sort the images based on when they were added to the device. This will allow us to sort the results from newest to oldest so the user will see their most recent photos first.

The results of the query are returned as a Cursor interface table, which the app can iterate over row-by-row to find the details of each image on the user's device. This processing will be handled by a method called addPhotosFromCursor that will use the results in the Cursor to create a corresponding list of Photo objects. To define the addPhotosFromCursor method, add the following code below the loadPhotos method:

```
private fun addPhotosFromCursor(cursor: Cursor): List<Photo> {
    val photoList = mutableListOf<Photo>()
```

```

while (cursor.moveToNext()) {
    val id = cursor.getLong(0)
    val contentUri = ContentUris.withAppendedId(MediaStore.Images.Media.EXTERNAL_CONTENT_URI, id)

    val photo = Photo(id, contentUri)
    photoList += photo
}

return photoList
}

```

The `addPhotosFromCursor` method will use a mutable list called `photoList` to store the `Photo` objects that are generated based on the data in the `Cursor`. The results are iterated over using the `Cursor` class's `moveToNext` method. For each result, we extract the image's ID using the `getLong` command because the ID is stored in `Long` format. The `Cursor` only contains one column (as specified in the projection in the `loadPhotos` method) so we know the ID column index will be 0. Next, the ID of each image is used to build a content URI that identifies the image in the device's media store. For example, image content URIs often take the format `content://media/external/images/media/ID`, where 'ID' represents the image's ID number. The image ID and content URI are then packaged into a `Photo` object and added to the `photoList` variable. Once all the results in the `Cursor` have been processed, the `addPhotosFromCursor` method returns the full list of `Photo` objects to the `loadPhotos` method so it can be made available to the wider app.

The app is now capable of generating a list of `Photo` objects representing the images on the user's device. However, it is important to consider that the collection of images may change as old images are deleted and new ones are created. To detect and respond to changes we must register a content observer. The content observer will monitor and respond to changes to the collection of images on the user's device. To implement the content observer when the view model is initialised, add the following code below the list of variables at the top of the `GalleryViewModel` class:

```

init {
    contentObserver = getApplication<Application>
().contentResolver.registerObserver(MediaStore.Images.Media.EXTERNAL_CONTENT_URI) {
    loadPhotos()
}
}

```

Next, add the following code below the `addPhotosFromCursor` method to direct the content observer to monitor the images on the user's device:

```

private fun ContentResolver.registerObserver(
    uri: Uri,
    observer: (selfChange: Boolean) -> Unit
): ContentObserver {
    val contentObserver = object : ContentObserver(Handler(Looper.getMainLooper())) {
        override fun onChange(selfChange: Boolean) {
            observer(selfChange)
        }
    }
    registerContentObserver(uri, true, contentObserver)
    return contentObserver
}

```

Note you may need to add the following import statement to the top of the file:

```
import android.os.Handler
```

The `registerObserver` method will direct the content observer to monitor changes to files that feature an `Images MediaStore` content URI. If a change occurs (e.g. an image is added, updated or deleted) then the `loadPhotos` method will run. In this way, the list of `Photo` objects held by the app will update to reflect the changes.

When the view model is closed, we should unregister the content observer to prevent memory leaks and any unnecessary use of the device's computational resources. To unregister the content observer, add the following code below the `addPhotosFromCursor` method:

```

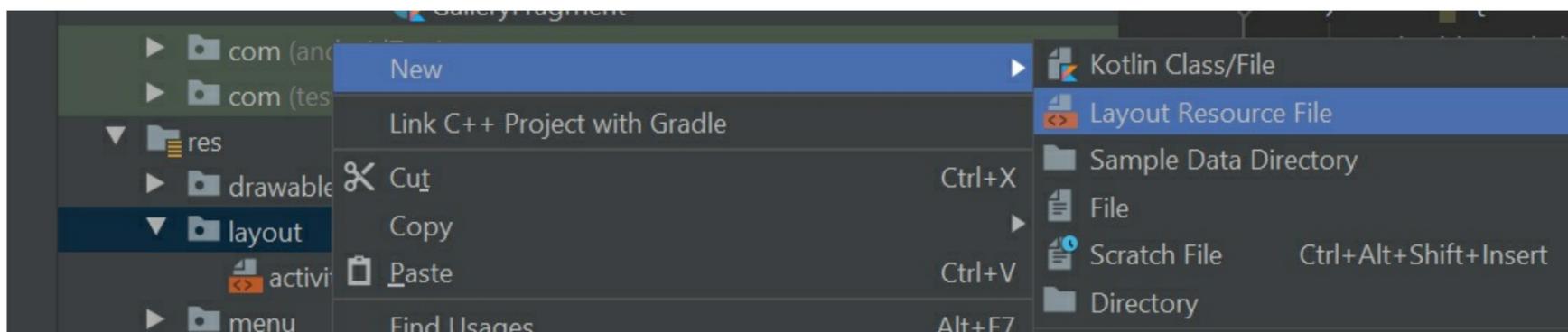
override fun onCleared() {
    applicationContext.contentResolver.unregisterContentObserver(contentObserver)
}

```

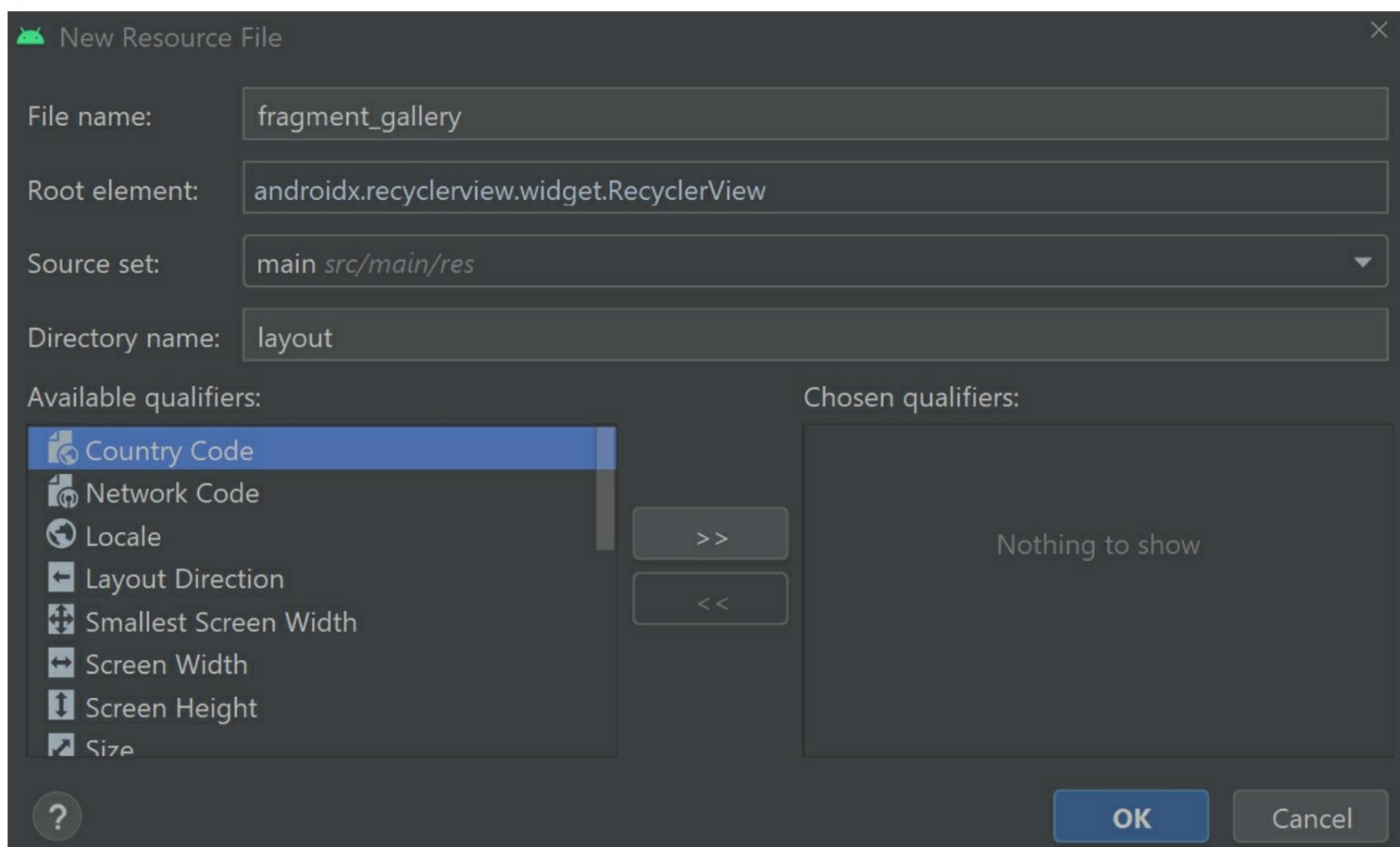
The onCleared method defined above refers to a stage of the ViewModel class lifecycle that runs when the view model is no longer being used and is in the process of shutting down. In this event, the above code will unregister the content observer and prevent it from continuing to monitor the images on the user's device.

## Setting up the Gallery fragment and layout

The image gallery will be displayed using a dedicated fragment, which will require a layout. To create the layout navigate through **Project > app > res** then right-click the **layout** folder and select **New > Layout Resource File**.



Name the file `fragment_gallery`, set the root element to `androidx.recyclerview.widget.RecyclerView`, then press OK.



Android Studio should then create a layout that contains a solitary RecyclerView widget. The RecyclerView widget will occupy the entire layout and display previews of every image stored on the user's device. To load the image previews, we must first initialise the RecyclerView widget. To do this, locate and open the **GalleryFragment.kt** file by navigating through **Project > app > java > name of the project > ui > gallery**. If Android Studio has already added a view model variable at the top of the class (likely called `homeViewModel`) then delete it. Next, amend the `_binding` variable so it reads as follows and imports the gallery fragment layout binding class:

```
private var _binding: FragmentGalleryBinding? = null
```

Also, add the following variable to provide access to the GalleryViewModel class:

```
private lateinit var viewModel: GalleryViewModel
```

To initialise the `fragment_gallery` layout's binding class, edit the `onCreateView` method so it reads as follows:

```
override fun onCreateView(  
    inflater: LayoutInflater,  
    container: ViewGroup?,  
    savedInstanceState: Bundle?  
): View {  
    _binding = FragmentGalleryBinding.inflate(inflater, container, false)  
    return binding.root  
}
```

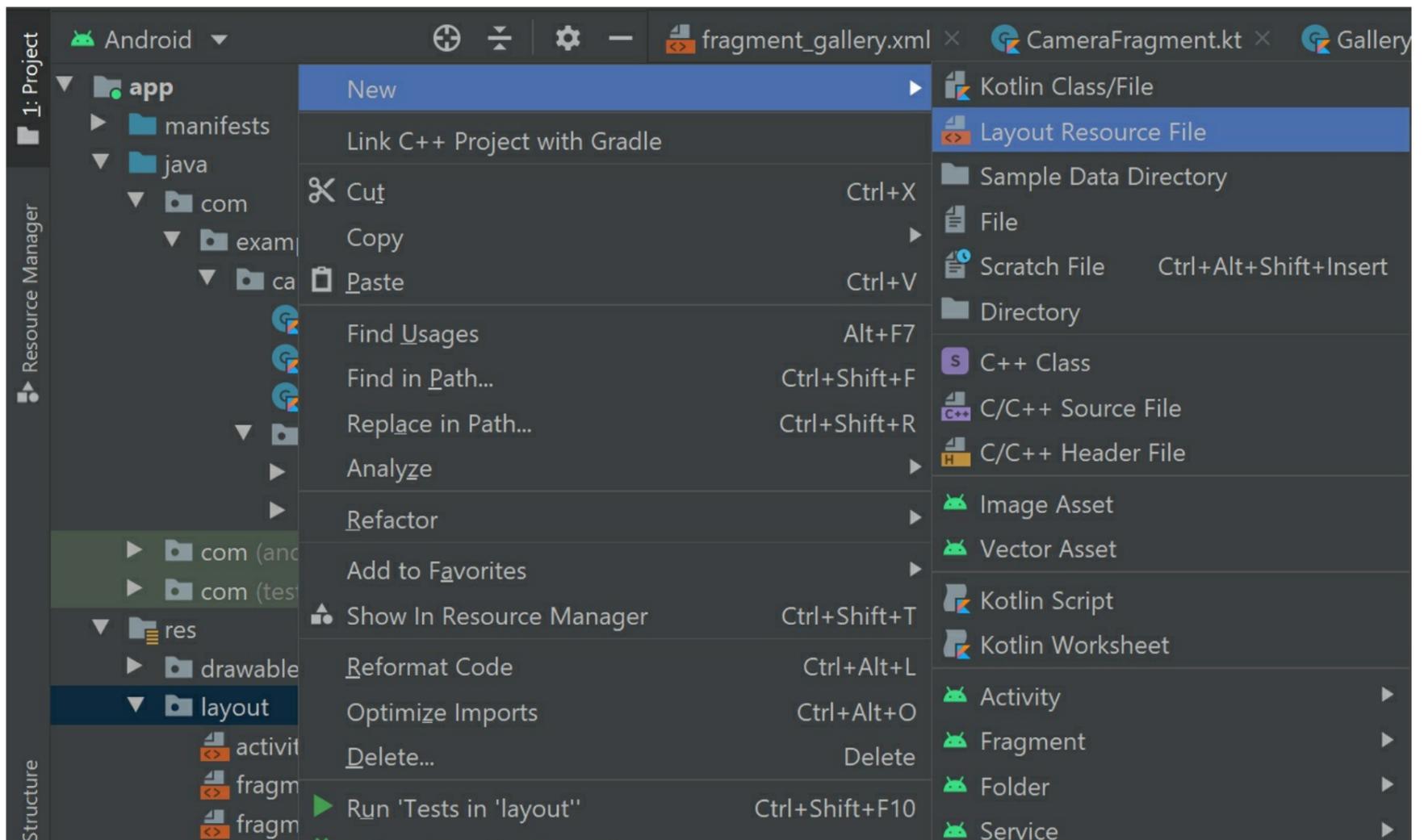
Once the `fragment_gallery` layout is available, we should initialise the `GalleryViewModel` view model. To arrange this, add the following code below the `onCreateView` method:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
    super.onViewCreated(view, savedInstanceState)  
  
    // TODO: Apply the adapter to the RecyclerView  
  
    viewModel = ViewModelProvider(this)[GalleryViewModel::class.java]  
    viewModel.photos.observe(viewLifecycleOwner, { photos ->  
        photos?.let {  
            // TODO: Load the photo previews here  
        }  
    })  
  
    if (MainActivity.CameraPermissionHelper.hasStoragePermission(requireActivity())) viewModel.loadPhotos()  
    else MainActivity.CameraPermissionHelper.requestPermissions(requireActivity())  
}
```

The above code initialises the `GalleryViewModel` view model and registers an observer on the view model's `photos` variable. The observer allows the fragment to monitor the contents of the variable and respond to changes in real-time. The remainder of the above code uses the `CameraPermissionHelper` object in the `MainActivity` class to check whether the app has permission to access the device's storage. If permission has been granted, then the `GalleryViewModel` view model's `loadPhotos` method will retrieve the list of images on the user's device. Otherwise, the `CameraPermissionHelper` object will request the necessary permissions from the user.

## Displaying image previews

In this section, we will design and implement an adapter that will load a preview of each image on the user's device into the `RecyclerView` widget found in the `fragment_gallery.xml` layout. To facilitate this, we first need to create a layout file that will display each image preview. Create a new layout resource file in the usual way, by right-clicking the `layout` directory (found by navigating **Project** > **app** > **res**) then selecting **New** > **Layout Resource File**.



Name the layout image\_preview then press OK. Once the layout opens in the editor, switch to Code view and edit the contents of the file so it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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="wrap_content">

    <ImageView
        android:id="@+id/image"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:foreground="?attr/selectableItemBackground"
        android:contentDescription="@string/image"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintDimensionRatio="1:1" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

The above code defines an `ImageView` widget that will display an image from the user's device. The `ImageView` widget contains a foreground attribute set to `selectableItemBackground`, which means a ripple effect will appear when the widget is pressed. The ripple effect will show the user which image they are selecting. The `ImageView` also contains a constraint dimension ratio of 1:1, a width set to occupy the maximum available space and a height set to 0dp. Altogether, these attributes mean the `ImageView` widget will occupy the maximum available width and the height will be the same length as the width, thereby ensuring the `ImageView` is square-shaped.

Moving on, let's create the adapter class that will coordinate the list of images loaded into the `RecyclerView` widget. Right-click the **gallery** directory then select **New > Kotlin Class/File**. Name the file `GalleryAdapter` and select `Class` from the list of options. Once the **GalleryAdapter.kt** file opens in the editor modify its contents so it reads as follows:

```
class GalleryAdapter(private val activity: MainActivity, private val fragment: GalleryFragment):
    RecyclerView.Adapter<RecyclerView.ViewHolder>() {

    var photos = listOf<Photo>()

    inner class GalleryViewHolder(itemView: View) :
        RecyclerView.ViewHolder(itemView),
```

```
View.OnClickListener {
```

```
    internal var mImage = itemView.findViewById<View>(R.id.image) as ImageView
```

```
    init {
```

```
        itemView.isClickable = true
```

```
        itemView.setOnClickListener(this)
```

```
        itemView.setOnLongClickListener {
```

```
            // TODO: Open the popup menu
```

```
            return@setOnLongClickListener true
```

```
        }
```

```
    override fun onClick(view: View) {
```

```
        // TODO: Navigate to the photo filter fragment
```

```
    }
```

```
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): RecyclerView.ViewHolder {
```

```
        return GalleryViewHolder(LayoutInflater.from(parent.context).inflate(R.layout.image_preview, parent, false))
```

```
    }
```

```
    override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position: Int) {
```

```
        holder as GalleryViewHolder
```

```
        val current = photos[position]
```

```
        Glide.with(activity)
```

```
            .load(current.uri)
```

```
            .centerCrop()
```

```
            .into(holder.mImage)
```

```
    }
```

```
    override fun getItemCount(): Int {
```

```
        return photos.size
```

```
    }
```

Note you may need to add the following import statement to the top of the class to import the Photo data class we created earlier (edit 'com.example.camera' to reflect the package name of your project as stated on line 1 of the file):

```
import com.example.camera.Photo
```

The GalleryAdapter class's primary constructor contains parameters called activity and fragment. The parameter values will link to the MainActivity and GalleryFragment classes and allow the adapter to access their data and methods. In the body of the adapter, a variable called photos will hold the list of Photo objects generated by the GalleryViewModel view model. There is also an inner class called GalleryViewHolder. The GalleryViewHolder inner class will initialise the components of the **image\_preview.xml** layout and handle user interactions. The adapter knows to use the **image\_preview.xml** layout for displaying items in the RecyclerView because this is the layout returned by the onCreateViewHolder method.

The next method in the adapter is called onBindViewHolder and determines how data is displayed at each position in the RecyclerView. It does this by finding the corresponding Photo object in the photos list for the current position in the RecyclerView then assigning the Photo object to a variable called current. Next, an image loading framework called Glide (<https://github.com/bumptech/glide>) retrieves the image file based on the content URI stored in the Photo object and loads it into the ImageView from the **image\_preview.xml** layout. In the above code, we also direct Glide to crop the image if necessary to make it fit in the ImageView widget.

Once the adapter is set up, we can integrate it with the gallery fragment and apply it to the RecyclerView widget. To do this, open the **GalleryFragment.kt** file and add a variable for the GalleryAdapter class to the list of variables at the top of the class:

```
private lateinit var galleryAdapter: GalleryAdapter
```

Next, locate the TODO comment in the onCreateView method that says to apply the adapter to the RecyclerView widget and replace the comment with the following code:

```
galleryAdapter = GalleryAdapter(activity as MainActivity, this)
binding.root.layoutManager = GridLayoutManager(context, 3)
binding.root.adapter = galleryAdapter
```

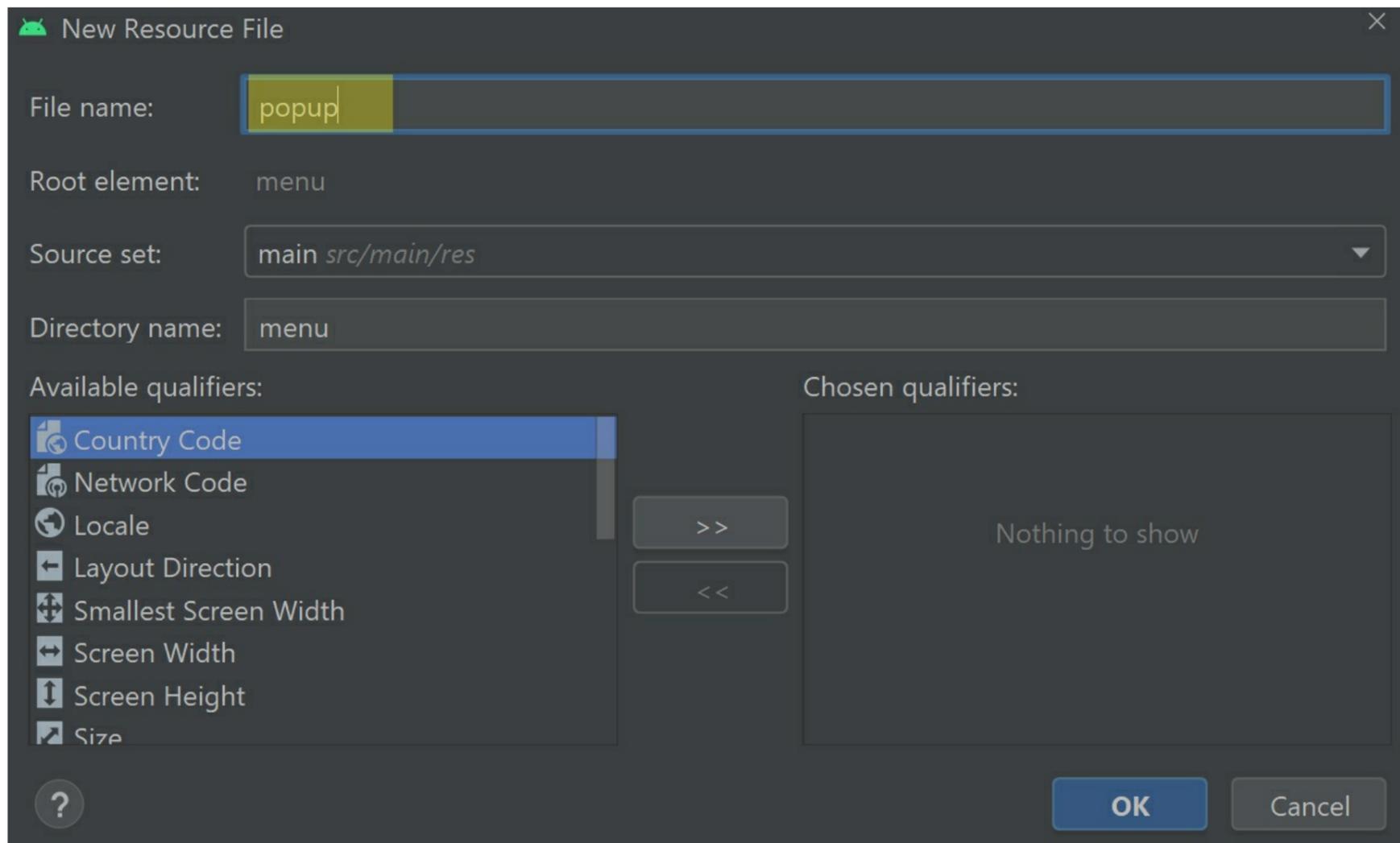
The above code initialises the GalleryAdapter adapter and binds the adapter to the RecyclerView so the adapter can interact with the RecyclerView and update its contents. A GridLayoutManager instance with a span count of 3 is also applied to the RecyclerView. The GridLayoutManager will direct the RecyclerView to display items in a grid containing three columns. To load the list of Photo objects into the adapter, locate the section of code where the observer is registered on the gallery view model's photos variable and replace the TODO comment with the following code:

```
galleryAdapter.notifyItemRangeRemoved(0, galleryAdapter.itemCount)
galleryAdapter.photos = it
galleryAdapter.notifyItemRangeInserted(0, it.size)
```

The above code directs the observer to send the list of Photo objects from the gallery view model to the adapter's photos variable. The observer will do this whenever there is a change to the contents of the gallery view model's photos variable, which means the adapter will be notified whenever photos are added or deleted. The observer uses the adapter's notifyItemRangeRemoved and notifyItemRangeInserted methods to refresh the image gallery with the new set of images.

## Deleting an image from the MediaStore

The Camera app will allow the user to delete images if they wish. To do this, the user simply needs to long click an image preview and a popup menu will appear inviting the user to delete the image. To create the popup menu, navigate through **Project > app > res** and right-click the **menu** directory. Next, select **New > Menu Resource File**, name the file popup and press OK.



Switch the file to Code view and add the following item between the opening and closing menu tags:

```
<item android:id="@+id/popup_delete"
    android:title="@string/delete_image" />
```

This menu item has an ID set to popup\_delete and will display the text "Delete image". When the user clicks the menu item, it will begin the process of deleting the selected image from the user's device.



## Delete image



The popup menu will open when the user long presses an image preview in the gallery fragment's RecyclerView. To enable this functionality, open the **GalleryAdapter.kt** file and replace the TODO comment inside the `onLongClickListener` of the `GalleryViewHolder` inner class with the following code:

```
fragment.showPopup(it, photos[layoutPosition])
```

The above code will run a `GalleryFragment` method called `showPopup`. The `showPopup` method will display the popup menu and invite the user to delete the image. To define the `showPopup` method, open the **GalleryFragment.kt** file and add the following code below the `onViewCreated` method:

```
fun showPopup(view: View, photo: Photo) {  
    val popup = PopupMenu(requireActivity(), view)  
    popup.inflate(R.menu.popup)  
  
    popup.setOnMenuItemClickListener {  
        if (it.itemId == R.id.popup_delete) {  
            viewModel.photoToDelete = photo  
            deletePhoto()  
        }  
        true  
    }  
  
    popup.show()  
}
```

You may need to manually add the following import statements to the top of the file:

```
import androidx.appcompat.widget.PopupMenu  
import com.example.camera.Photo
```

The `showPopup` method uses the `PopupMenu` class to create a popup menu that appears over a given `View`. In this case, the `View` will be the `ImageView` widget that the user has selected. The actions that occur when a menu item is clicked are specified in the `OnMenuItemClickListener` callback function. In this case, there is only one menu item and it is called `popup_delete`. If the `popup_delete` menu item is clicked then the `Photo` object associated with the user's selected image will be assigned to a `GalleryViewModel` variable called `photoToDelete`. Next, a method called `deletePhoto` will attempt to delete the image. The `Photo` object is assigned to a `GalleryViewModel` variable because the app will need to refer to the object again if the first deletion attempt is unsuccessful. For example, if the user attempts to delete an image that the app did not create, then the app will have to request permission from the user and reattempt the deletion. To define the `photoToDelete` variable, add the following code to the list of variables at the top of the `GalleryViewModel` class:

```
var photoToDelete: Photo? = null
```

Next, return to the `GalleryFragment` class and add the following code below the `showPopup` method to define the `deletePhoto` method:

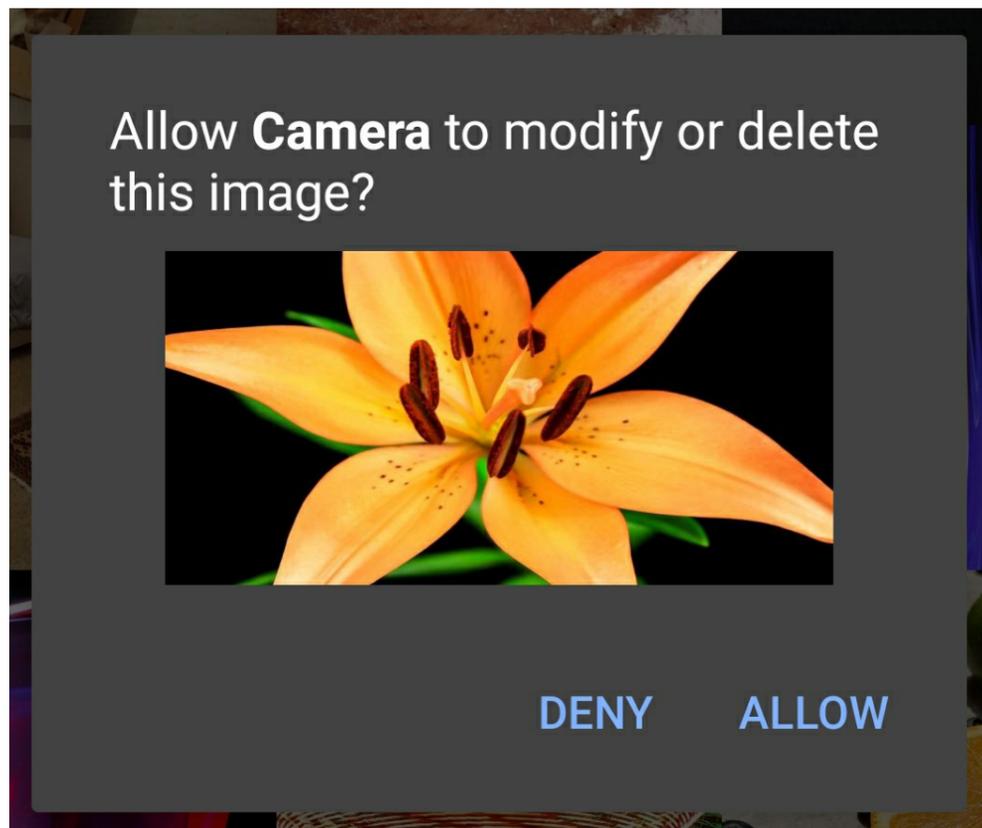
```
private fun deletePhoto() {  
    try {  
        val photo = viewModel.photoToDelete ?: return  
        val rowsDeleted = requireActivity().applicationContext.contentResolver.delete(photo.uri, null)
```

```

    if (rowsDeleted == 1) viewModel.photoToDelete = null
  } catch (recoverableSecurityException: RecoverableSecurityException) {
    val intentSender = recoverableSecurityException.userAction.actionIntent.intentSender
    val intentSenderRequest = IntentSenderRequest.Builder(intentSender).build()
    registerResult.launch(intentSenderRequest)
  }
}

```

The deletePhoto method retrieves the value assigned to the GalleryViewModel class's photoToDelete variable. If the variable is null, then no further processing will occur. Meanwhile, if the Photo object is successfully retrieved, then the method uses the app's content resolver to delete the associated image from the MediaStore based on its ID. This operation is enclosed inside a try/catch block because from Android 10.0 (API 29) and up all applications must adhere to scoped storage guidelines. Scoped storage is a framework that restricts an application's access to files it did not create and is designed to protect the user's privacy. For this reason, requests to delete images may throw a recoverable security exception that requires the user to grant permission for the action to proceed. If a recoverable security exception is thrown, then an IntentSenderRequest object is built to prompt the device to ask the user whether they wish to modify (or delete) the image file.



The IntentSenderRequest object will be initiated and monitored by an activity request launcher. To define the activity request launcher, add the following variable to the list of variables at the top of the fragment:

```

private val registerResult = registerForActivityResult(ActivityResultContracts.StartIntentSenderForResult()) {
    result: ActivityResult ->
        // Result code of 0 means the user declined permission to delete the photo
        if (result.resultCode != 0) deletePhoto()
}

```

Note you may need to add the following import statement to the top of the file:

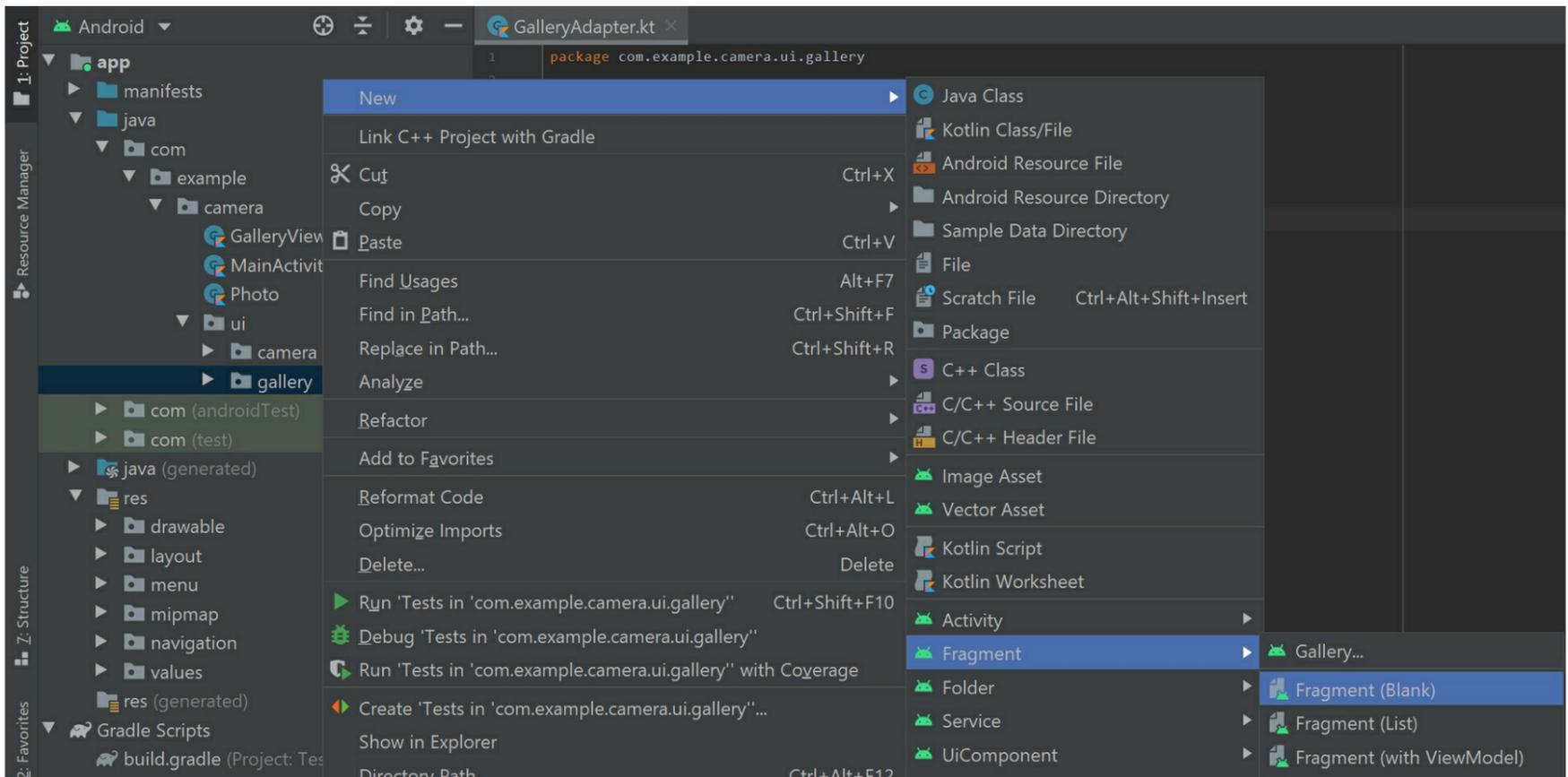
```
import androidx.activity.result.ActivityResult
```

The registerResult variable defined above will create the IntentSenderRequest object that prompts the user for whether they wish to delete the photo. If the user denies permission, then a result code of 0 will be returned. In which case, no further processing will occur. Meanwhile, if the user approves permission, then the deletePhoto method will run again. On which occasion, the app should have permission to delete the image file and the operation should complete successfully. If the content resolver confirms that 1 row has been removed from the media store, then this means the image has been successfully deleted and so the deletePhoto method sets the GalleryViewModel view model's photoToDelete variable back to null.

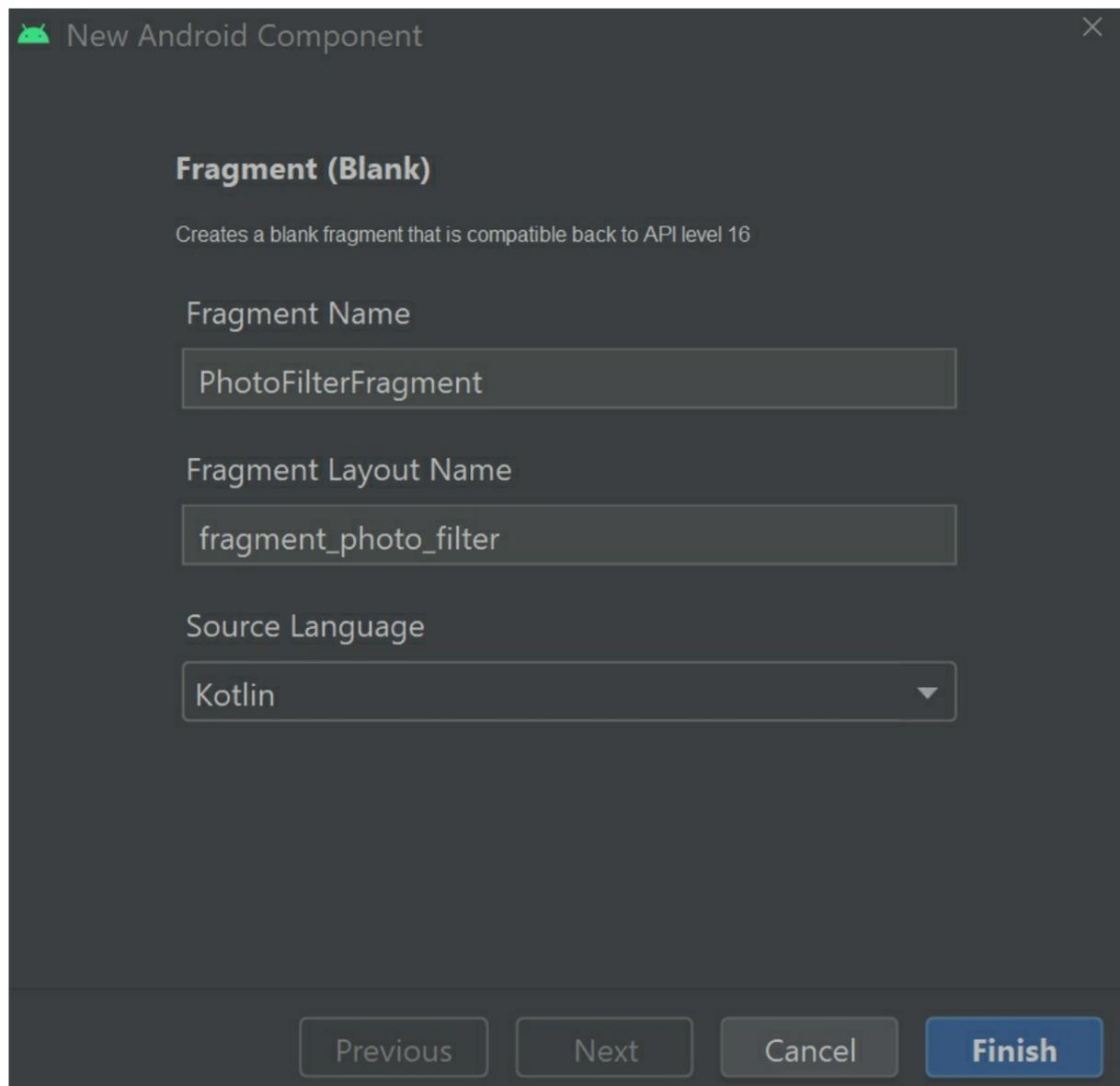
## Setting up the Photo Filter fragment

The Camera application will have a feature that enables the user to apply a readymade set of filters to their images. This functionality will be handled by a dedicated fragment. To create the fragment, right-click the **gallery ui**

directory then select **New > Fragment > Fragment (Blank)**.



In the New Android Component window that opens, set the fragment name to `PhotoFilterFragment` and use `fragment_photo_filter` as the layout name. Next, press **Finish** and Android Studio should then create a Kotlin class and XML layout file for the new fragment.



Let's design the layout first. Open the `fragment_photo_filter.xml` layout in Code view and edit its contents so it reads as follows:

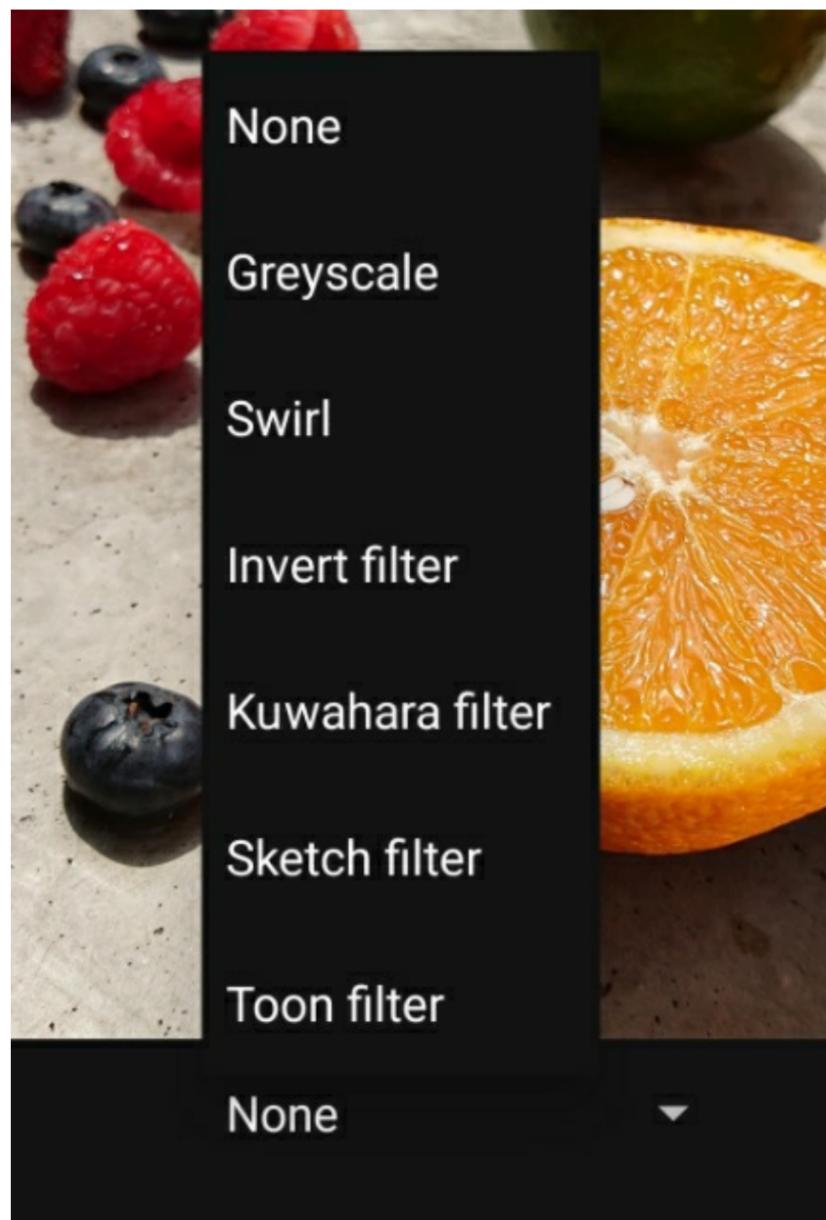
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
```

```
android:layout_height="match_parent">
```

```
<ImageView  
    android:id="@+id/selectedImage"  
    android:layout_width="match_parent"  
    android:layout_height="0dp"  
    android:contentDescription="@string/image"  
    app:layout_constraintDimensionRatio="1:1"  
    app:layout_constraintTop_toTopOf="parent" />
```

```
<Spinner  
    android:id="@+id/filterSpinner"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginTop="12dp"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintTop_toBottomOf="@id/selectedImage" />  
</androidx.constraintlayout.widget.ConstraintLayout>
```

The **fragment\_photo\_filter.xml** layout contains two widgets. The first widget is an `ImageView` that will display the user's selected image. The `ImageView` will occupy the maximum available width. Its height will equal the width because the constraint dimension ratio used is 1:1. In this way, we ensure the image will appear as a square. The second widget is a `Spinner`. The `Spinner` will display the list of available filters.



With the layout now in place, let's turn our attention to the `PhotoFilterFragment` class. Android Studio will likely have automatically generated a lot of code when the fragment was created. Much of this code is not applicable for our purposes so delete all the code except the package declaration on line 1. Next, add the following code to the file to define the class and initialise the `fragment_photo_filter` layout's binding class:

```
import androidx.fragment.app.Fragment  
import com.example.camera.Photo  
  
class PhotoFilterFragment : Fragment() {
```

```

private var _binding: FragmentPhotoFilterBinding? = null
private val binding get() = _binding!!
private var photo: Photo? = null

override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
): View {
    _binding = FragmentPhotoFilterBinding.inflate(inflater, container, false)
    return binding.root
}

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    setHasOptionsMenu(true)

    loadImage(null)
}

override fun onDestroyView() {
    super.onDestroyView()
    _binding = null
}
}

```

The photo filter fragment also contains a variable called photo, which will store the Photo object associated with the user's selected image. The image will be displayed using a method called loadImage. To define the loadImage method, add the following code below the onViewCreated method:

```

private fun loadImage(glideFilter: Transformation<Bitmap>?) {
    when {
        photo != null && glideFilter != null -> {
            Glide.with(this)
                .load(photo!!.uri)
                .transform(
                    CenterCrop(),
                    glideFilter
                )
                .diskCacheStrategy(DiskCacheStrategy.NONE)
                .into(binding.selectedImage)
        }
        photo != null -> {
            Glide.with(this)
                .load(photo!!.uri)
                .centerCrop()
                .diskCacheStrategy(DiskCacheStrategy.NONE)
                .into(binding.selectedImage)
        }
    }
}
}

```

Note you may need to add the following import statement to the top of the file:

```
import com.bumptech.glide.load.Transformation
```

The loadImage method has an argument called glideFilter that will accept a Transformation object. The Transformation object details a filter that should be applied to the image. If the image should be displayed without a filter, then a null value should be supplied for the glideFilter argument. The loadImage method uses a when block to respond to different use-case scenarios. First, if both the Photo and Transformation objects are not null then Glide will load the image and use its transform command to apply the user's selected filter. The filters are provided by an external library called Glide Transformations, which you can read about on Github:

<https://github.com/wasabeef/glide-transformations>. Meanwhile, the second scenario in the when block will run

when a Photo object is provided but the glideFilter parameter is null. In which case, Glide is directed to load the image without applying a transformation, which will restore an unfiltered version of the image.

It is noteworthy that we instruct Glide to implement a DiskCacheStrategy of NONE. Typically, Glide will attempt to store a cache of images in the device's memory to make it easier to load those images again in future. In this instance, we disable the cache feature because each image must be reloaded from scratch whenever a filter is applied or removed. Otherwise, Glide may load a previous version of the image rather than one with the filter applied.

## Navigating to the photo filter fragment

For the photo filter fragment to load the user's selected image, we must communicate their selection from the gallery fragment to the photo filter fragment. To send data between fragments, we will use the Safe Args plugin. The Safe Args plugin creates a custom class for each destination that can transport data during navigation events.

Each item of data that is transmitted during navigation must be defined in the app's navigation graph. The navigation graph demonstrates the network of destinations throughout the app. To locate the navigation graph, navigate through **Project > app > res > navigation** and open the file called **mobile\_navigation.xml**. To define the destinations that the Camera app will use, switch the navigation graph to Code view and replace the code between the opening and closing navigation tags with the following:

```
<fragment
  android:id="@+id/nav_camera"
  android:name="com.example.camera.ui.camera.CameraFragment"
  android:label="@string/camera"
  tools:layout="@layout/fragment_camera" />
```

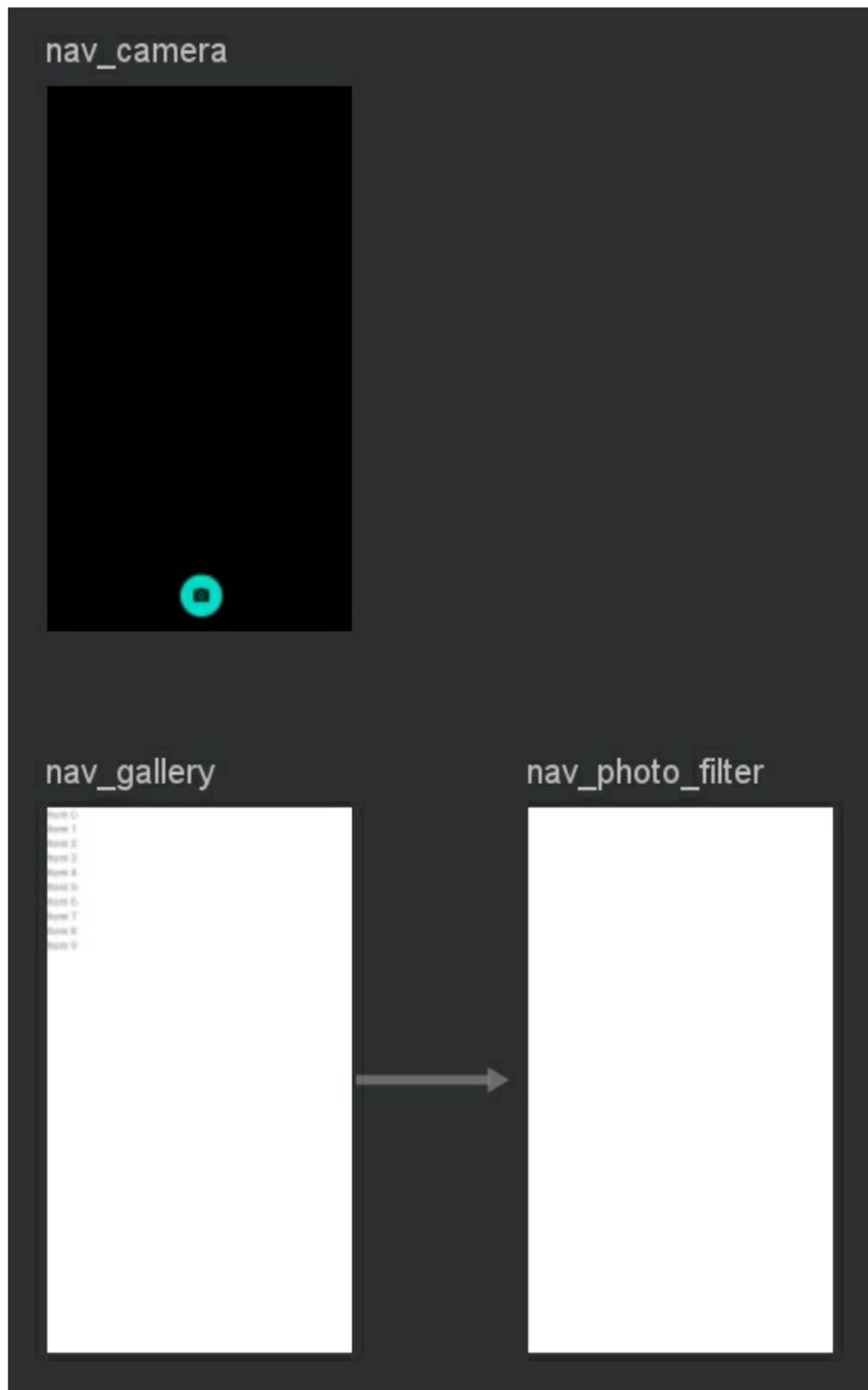
```
<fragment
  android:id="@+id/nav_gallery"
  android:name="com.example.camera.ui.gallery.GalleryFragment"
  android:label="@string/gallery"
  tools:layout="@layout/fragment_gallery">
```

```
<action
  android:id="@+id/actionPhotoFilter"
  app:destination="@id/nav_photo_filter" />
</fragment>
```

```
<fragment
  android:id="@+id/nav_photo_filter"
  android:name="com.example.camera.ui.gallery.PhotoFilterFragment"
  android:label="Edit photo"
  tools:layout="@layout/fragment_photo_filter" >
  <argument
    android:name="photo"
    android:defaultValue="@null"
    app:argType="com.example.camera.Photo"
    app:nullable="true" />
</fragment>
```

The above code defines separate destinations for the camera, gallery and photo filter fragments. Each destination contains an ID attribute, which identifies the destination in the navigation graph; a name attribute that specifies the location of the destination; a label that will be displayed in the app toolbar when the destination is active; and the layout file which will be used for that destination.

Destinations are linked through actions. For example, the gallery fragment has an action that leads to the photo filter fragment because the user can navigate from the gallery fragment to the photo filter fragment by clicking one of the image previews. Each action must include an ID, which is used to reference and initiate the action, and a destination attribute that contains the ID of the target destination. To see a graphical representation of the navigation network, switch the navigation graph to Design view and notice the arrow from the gallery fragment to the photo filter fragment. The arrow represents that there is an action leading from one fragment to the other.



Data can be transmitted from one destination to another using the Safe Args plugin. The data type that can be transmitted is defined as an argument for the target destination. For example, the photo filter fragment accepts a Photo object as an argument. The Photo object will be supplied by the gallery fragment when the user selects an image preview. When defining an argument, you must assign it a name, a default value and specify the data type (argType). You can also state whether the argument is nullable. If an argument is nullable, then the argument value can be null rather than a value of the corresponding data type.

We're almost finished with the navigation graph. The last change we need to make is to set the start destination. The start destination is the origin of the navigation graph and the first destination the user will see when the app loads. To set the origin destination to the camera fragment, edit the startDestination attribute of the opening navigation tag so it reads as follows:

```
app:startDestination="@+id/nav_camera"
```

Moving on, let's implement the action that transports the user from the gallery fragment to the photo filter fragment. To do this, open the **GalleryAdapter.kt** file (**Project > Java > Name of project > ui > gallery**) and locate the onClick function in the GalleryViewHolder inner class. The onClick function should contain a TODO comment. Replace the comment with the following code:

```
val action = GalleryFragmentDirections.actionPhotoFilter(photos[layoutPosition])
view.findNavController().navigate(action)
```

Note you may need to add the following import statement to the top of the file:

```
import androidx.navigation.findNavController
```

The above code uses the `GalleryFragmentDirections` class (that was generated by the Safe Args plugin) to access the actions that originate from the gallery fragment destination. In this instance, we will use the `actionPhotoFilter` action that leads to the photo filter fragment. The photo filter fragment destination accepts a `Photo` object as an argument, and so the `Photo` object associated with the user's selection is included with the action. Once the action has been prepared, we instruct the navigation controller to initiate the `actionPhotoFilter` action and transport the user to the photo filter fragment along with the corresponding `Photo` object for their selected image.

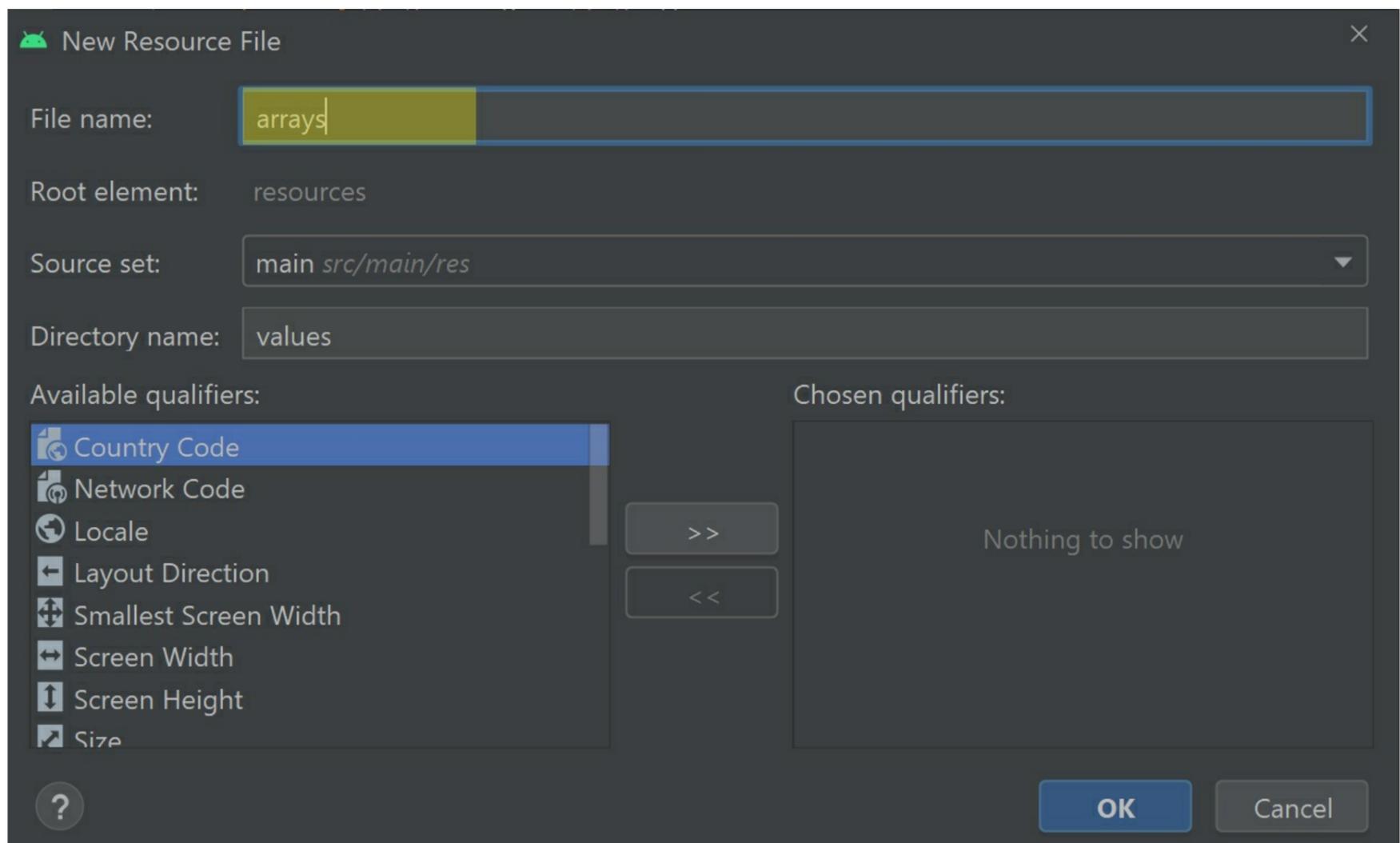
Finally, let's prepare the photo filter fragment to receive the incoming `Photo` object. Open the **PhotoFilterFragment.kt** file (**Project** > **Java** > **Name of project** > **ui** > **gallery**) and add the following code to the beginning of the `onCreateView` method:

```
arguments?.let {  
    val safeArgs = PhotoFilterFragmentArgs.fromBundle(it)  
    photo = safeArgs.photo  
}
```

The above code retrieves the arguments that are supplied when the photo filter fragment is created. To extract an argument's value, simply search the fragment's Safe Args class for a property that shares the same name as the argument. Referring back to the navigation graph, we can see the photo filter fragment's argument is called `photo`. In the above code, the `Photo` object associated with the `photo` argument is transferred to a variable so it can be used elsewhere in the fragment.

## Applying filters to a photo

The photo filter fragment will allow the user to apply filters to their selected image. The range of available filters will be listed in a spinner widget, which will draw the list of options from an array resource. To create the array resource, right-click the `values` directory (**Project** > **app** > **res**) then select **New** > **Values Resource File**. Name the file `arrays` then press OK to create a resource file called **arrays.xml**.



Add the following code between the opening and closing resource tags to define an array of strings called `filters_array`. Each item in the array contains the name of a different filter the user can select.

```
<string-array name="filters_array">  
    <item>None</item>  
    <item>Greyscale</item>  
    <item>Swirl</item>  
    <item>Invert filter</item>
```

```

<item>Kuwahara filter</item>
<item>Sketch filter</item>
<item>Toon filter</item>
</string-array>

```

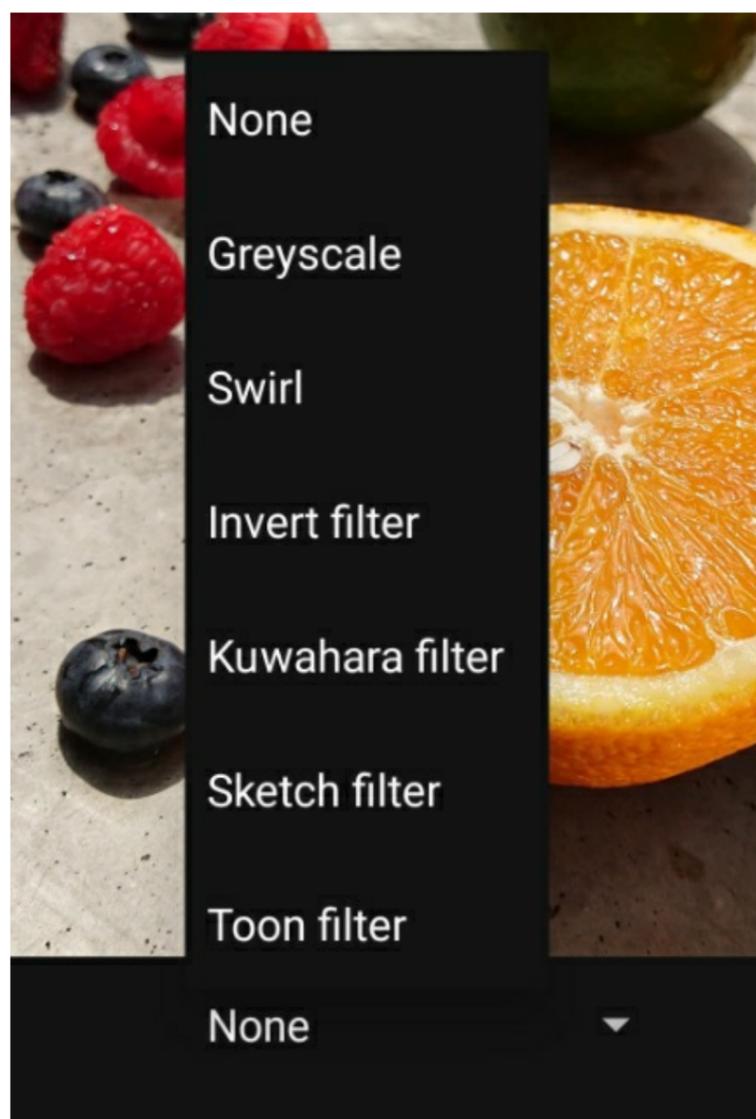
To apply the string array to the spinner, return to the **PhotoFilterFragment.kt** file (**Project > Java > Name of project > ui > gallery**) and add the following code to the bottom of the `onViewCreated` method:

```

// Create an ArrayAdapter using the string array and a default spinner layout
ArrayAdapter.createFromResource(
    requireActivity(),
    R.array.filters_array,
    android.R.layout.simple_spinner_item
).also { adapter ->
    // Specify the layout to use when the list of choices appears
    adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item)
    // Apply the adapter to the spinner
    binding.filterSpinner.adapter = adapter
}

```

The above code creates an instance of the `ArrayAdapter` class, which is a class that displays and presents data in widgets such as `ListViews` and `Spinners`. In this case, we use the `ArrayAdapter` class to load the filters array and present its contents using Android's default layout for `Spinner` items. The `ArrayAdapter` instance is then applied to the `Spinner` widget from the `fragment_photo_filter.xml` layout so the user can select their preferred filter.



Next, let's define what actions will happen when an item is selected. To do this, add the following code below the `ArrayAdapter` instance:

```

binding.filterSpinner.onItemSelectedListener = object : AdapterView.OnItemSelectedListener {
    override fun onItemSelected(
        parent: AdapterView<*>?,
        view: View?,
        position: Int,
        id: Long
    ) {
        val filter = parent?.getItemAtPosition(position).toString()
    }
}

```

```

    applyFilter(filter)
}

override fun onNothingSelected(parent: AdapterView<*>?) { }
}

```

The above code applies an `onItemSelected` listener to the Spinner. When the user selects an item, the listener's `onItemSelected` callback function will retrieve the name of the filter based on the selected item's position in the filters array. Next, a method called `applyFilter` will apply the selected filter to the image. `onItemSelected` listeners also require you to define an `onNothingSelected` callback function that runs when the user unselects an item or the selected item disappears from the adapter. In this app, we will not utilise the `onNothingSelected` function so it is left empty.

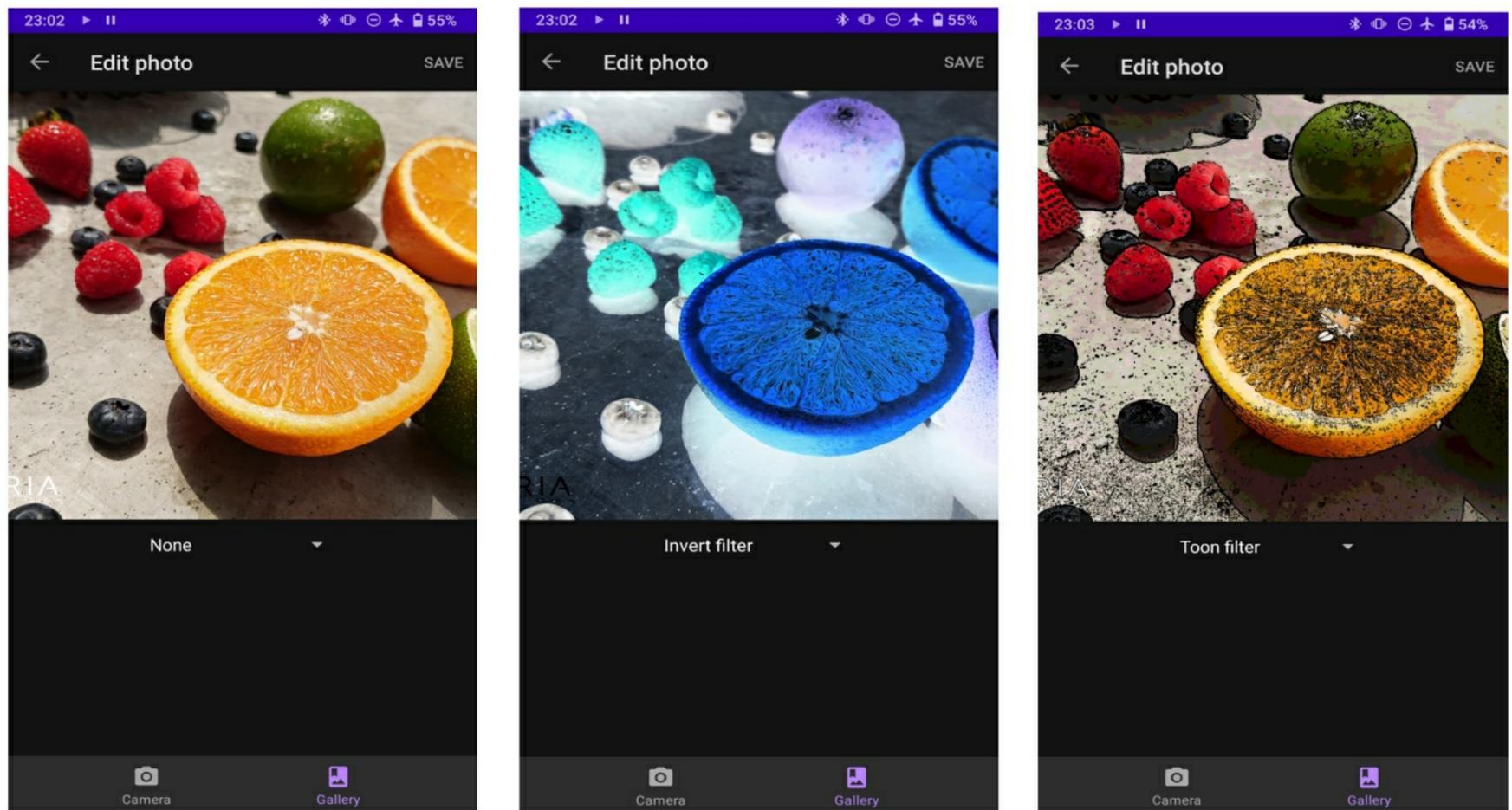
Moving on, let's define the `applyFilter` method. The `applyFilter` method will transform the image based on the user's filter selection. To define the `applyFilter` method, add the following code below the `onViewCreated` method:

```

private fun applyFilter(filter: String?) {
    when (filter) {
        "None" -> loadImage(null)
        "Greyscale" -> loadImage(GrayscaleTransformation())
        "Swirl" -> loadImage(SwirlFilterTransformation(0.5f, 1.0f, PointF(0.5f, 0.5f)))
        "Invert filter" -> loadImage(InvertFilterTransformation())
        "Kuwahara filter" -> loadImage(KuwaharaFilterTransformation(25))
        "Sketch filter" -> loadImage(SketchFilterTransformation())
        "Toon filter" -> loadImage(ToonFilterTransformation())
    }
}
}

```

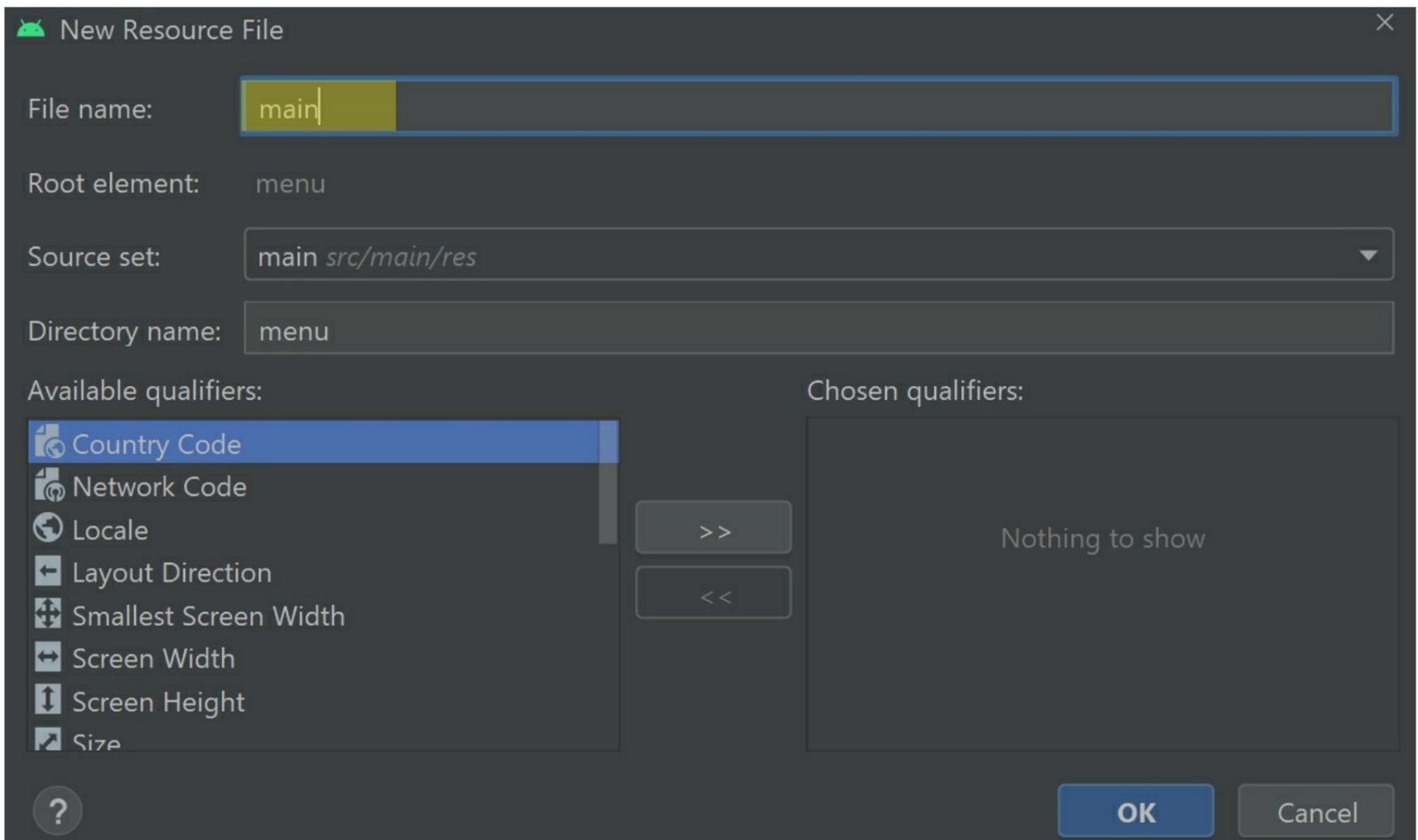
The `applyFilter` method uses a `when` block to apply the selected filter to the image. The classes which generate the filter effects are retrieved from a transformation library called `Glide Transformations` made by Daichi Furiya. You can read more about this library on the project's Github page: <https://github.com/wasabeef/glide-transformations>. The corresponding filter class for the user's selection is sent to the `loadImage` method, which applies the filter to the image as described in the previous section called 'Setting up the Photo Filter fragment'. If the user selects the "None" item from the Spinner, then the `loadImage` method will display the original unfiltered image.



### Saving the image once a filter has been applied

In this section, we will explore how to save the content of an `ImageView` widget to the user's device as an image file. For example, there will be a "save" menu item in the toolbar of the photo filter fragment that allows the user to

save the filtered image. To build the menu, navigate through **Project** > **app** > **res** and right-click the **menu** directory. Select **New** > **Menu Resource File**, name the file **main** and press OK.



The **main.xml** menu file will then open in the editor. Switch the file to Code view and edit its contents so it reads as follows:

```
<?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/save"
        android:title="@string/save"
        android:visible="false"
        app:showAsAction="ifRoom" />
</menu>
```

The above code defines a menu item called **save**, which by default will not be visible because the menu item should only appear in the photo filter fragment. The menu item's **showAsAction** attribute is set to **ifRoom**, which means the full menu item will appear in the app toolbar, providing there is enough space. Otherwise, the menu item will be accessible through the overflow menu.

To make the menu operational, return to the **PhotoFilterFragment.kt** file and add the following code below the **onViewCreated** method:

```
override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.main, menu)
    menu.findItem(R.id.save).isVisible = true

    super.onCreateOptionsMenu(menu, inflater)
}
```

The **onCreateOptionsMenu** method defined above uses the **MenuInflater** class to initialise the **main.xml** menu resource and interact with its constituent items. Also, the visibility attribute of the **save** menu item is set to **true** to reveal the menu item to the user. To handle clicks on the menu item, add the following code method below the **onCreateOptionsMenu** method:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        android.R.id.home -> findNavController().popBackStack()
    }
}
```

```

R.id.save -> {
    val image = getBitmapFromView(binding.selectedImage)
    if (image != null) (activity as MainActivity).saveImage(image)
    true
}
else -> super.onOptionsItemSelected(item)
}
}

```

Note you may need to add the following import statement to the top of the file:

```
import androidx.navigation.fragment.findNavController
```

If the save menu item is clicked, then a method called `getBitmapFromView` will capture the image currently being displayed, including any filters and modifications that have been applied. The resulting image is then saved to the user's device using a `MainActivity` class method called `saveImage`.

To define the `getBitmapFromView` method, add the following code below the `loadImage` method:

```

private fun getBitmapFromView(view: View): Bitmap? {
    val bitmap = Bitmap.createBitmap(view.width, view.height, Bitmap.Config.ARGB_8888)
    val canvas = Canvas(bitmap)
    view.draw(canvas)
    return bitmap
}

```

The `getBitmapFromView` method will create a bitmap representation of the image currently being displayed in the `selectedImage` `ImageView` widget. The width and the height of the bitmap are set to the same dimensions as the `ImageView` widget so the saved image will be equal in size to the image being displayed. When creating a bitmap, you must also specify a bitmap configuration to define how its pixels will be stored. The `ARGB_8888` configuration offers good pixel precision and image quality. Finally, the `Canvas` class is used to draw the image displayed in the `ImageView` widget into the bitmap. Altogether, the method creates a bitmap representation of the filtered image that is ready for storage on the user's device.

The image will be saved to the user's device using a `MainActivity` class method called `saveImage`. To define the method, add the following code to the `MainActivity.kt` file below the `prepareContentValues` method:

```

fun saveImage(bitmap: Bitmap) {
    val resolver = applicationContext.contentResolver
    val imageUri = resolver.insert(MediaStore.Images.Media.EXTERNAL_CONTENT_URI, prepareContentValues())

    try {
        val fos = imageUri?.let { resolver.openOutputStream(it) }
        bitmap.compress(Bitmap.CompressFormat.PNG, 100, fos)
        fos?.close()
        Toast.makeText(this, resources.getString(R.string.photo_saved), Toast.LENGTH_LONG).show()
    } catch (e: FileNotFoundException) {
        Toast.makeText(this, resources.getString(R.string.error_saving_photo), Toast.LENGTH_LONG).show()
    } catch (e: IOException) {
        Toast.makeText(this, resources.getString(R.string.error_saving_photo), Toast.LENGTH_LONG).show()
    }
}
}

```

The `saveImage` method contains an argument called `bitmap`, which receives a bitmap representation of the image that should be saved. The image is added to the device's media store using the application's content resolver. The details of the image file are defined using the `prepareContentValues` method, which was also used by the camera fragment when saving images captured by the camera. Next, the image is written to the destination specified in the media store URI using the `Bitmap` class's `compress` method. The `compress` method requires you to specify the file format (PNG in this instance), quality (0 being minimum quality and 100 being the maximum) and an output stream for writing data.

Once the image file has been successfully written, the output stream is closed and a toast notification informs the user the photo has been saved. The procedure is enclosed in a try/catch block to intercept errors that may otherwise cause the app to crash. These errors include a file not found exception, which is thrown by the content resolver's `openOutputStream` method if it cannot find the location specified in the URI, and an IO exception, which is thrown

by the output stream if there is an error with data input or output. If either exception occurs, then a toast notification informs the user that there was an error saving the image.

## The BottomNavigationView widget

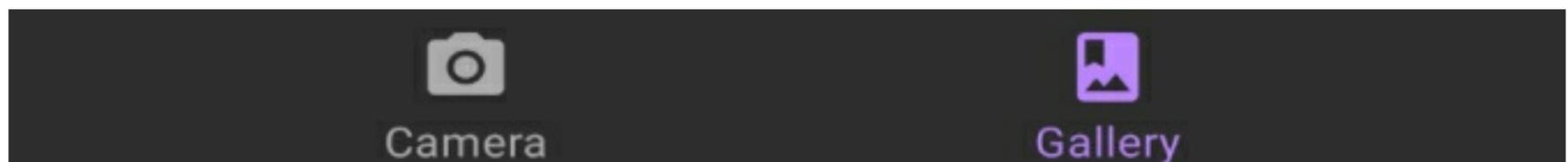
The app is almost finished. The user can capture and delete photos, and even apply filters to the images on their device. The last task is to allow the user to navigate around the app by configuring the BottomNavigationView widget. The BottomNavigationView widget was created automatically as part of the Bottom Navigation Activity project template. It will allow the user to navigate between the top-level destinations in the app, which in this case is the camera fragment and gallery fragment.

The destinations included in the BottomNavigationView widget are defined in a menu resource file. To configure the resource file, navigate through **Project > app > res > menu** and open the file called **bottom\_nav\_menu.xml**. Switch the file to Code view and replace the automatically generated menu items with the following code:

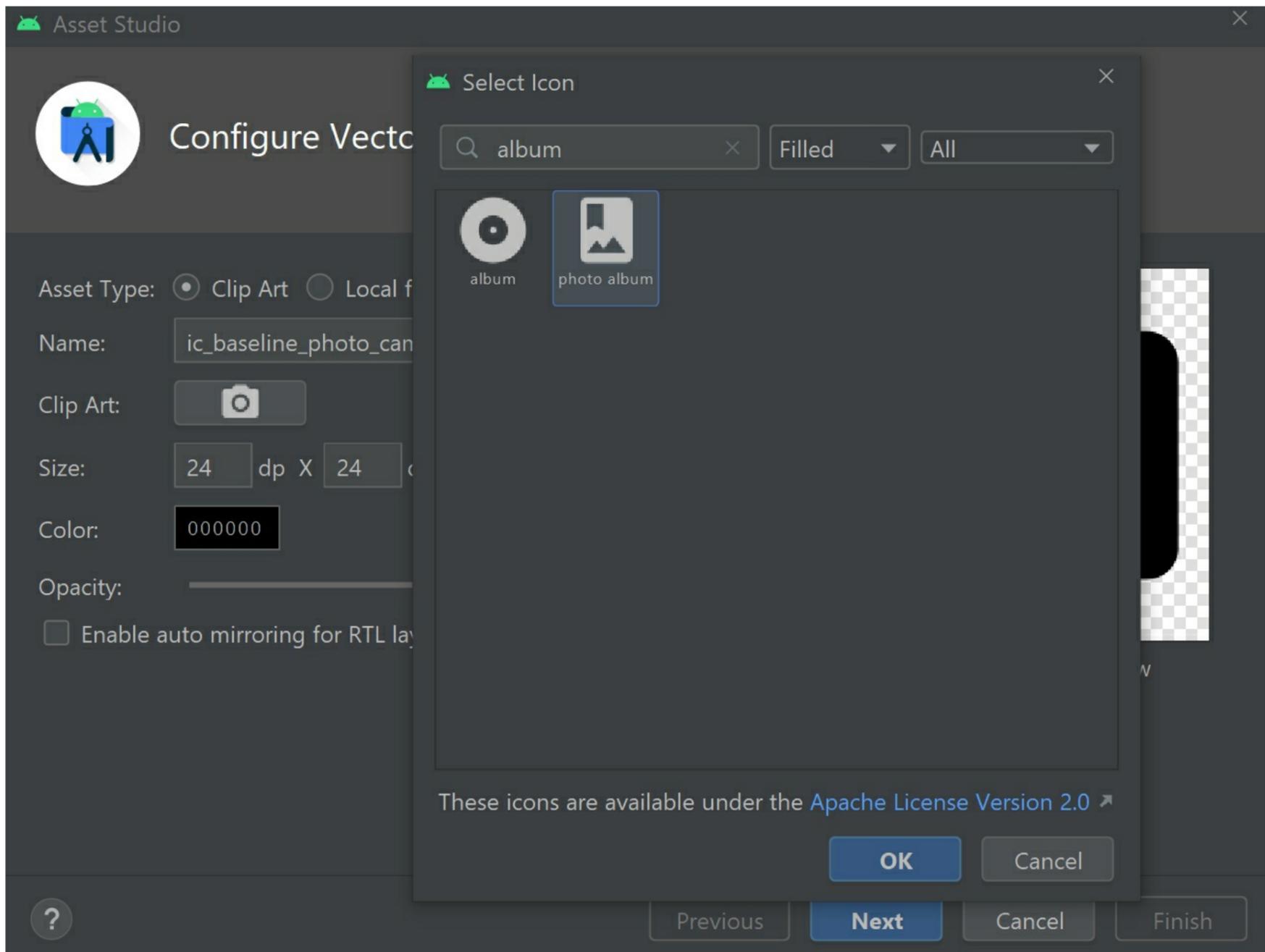
```
<item
  android:id="@+id/nav_camera"
  android:icon="@drawable/ic_camera"
  android:title="@string/camera" />
```

```
<item
  android:id="@+id/nav_gallery"
  android:icon="@drawable/ic_photo_album"
  android:title="@string/gallery" />
```

The menu items define the destinations that will appear in the BottomNavigationView. The ID of each menu item should match the ID of the corresponding navigation destination in the **mobile\_navigation.xml** navigation graph. Each menu item also contains an icon and a title, which are displayed in the BottomNavigationView widget.



We created the icon used for the nav\_camera menu item when we designed the camera fragment; however, we still need to make the photo album icon that will represent the gallery fragment. To create the icon, right-click the **drawable** directory (**Project > app > res**) then select **New > Vector Asset**. For the clip art, search for and select the icon called photo album then press OK.



Set the name of the vector asset to `ic_photo_album` then press Next and Finish to save the icon.

The `BottomNavigationView` widget can be found in the `activity_main.xml` layout (**Project** > **app** > **res**). The `activity_main` layout is the main layout of the activity and will load when the app is launched. It contains the `BottomNavigationView` widget and a fragment, which will display the content of the user's current destination in the app. It is now convention to use a `FragmentContainerView` widget rather than a regular fragment to display content, so replace the fragment with the following code:

```
<androidx.fragment.app.FragmentContainerView
    android:id="@+id/nav_host_fragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    app:defaultNavHost="true"
    app:navGraph="@navigation/mobile_navigation"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toTopOf="@id/nav_view" />
```

The above code defines a `FragmentContainerView` widget that will source its content from the `mobile_navigation.xml` navigation graph. The height of the `FragmentContainerView` is set to `0dp`, which means its height will occupy the maximum available space according to its constraints. In this case, the `FragmentContainerView` is constrained to the top of the parent layout and the `BottomNavigationView` at the bottom of the layout. These constraints mean the `FragmentContainerView` will occupy the maximum available space once it has left enough room for the `BottomNavigationView`.

Note the root `ConstraintLayout` in the `activity_main.xml` layout may contain a `paddingTop` attribute `android:paddingTop="?attr/actionBarSize"` designed to leave space for an action bar. This attribute will not be necessary for this app and can be deleted.

Moving on, let's now make the `BottomNavigationView` widget operational. To do this, return to the `MainActivity.kt` file and replace the `NavController` and `AppBarConfiguration` variables in the `onCreate` method with

the following code:

```
val navHostFragment = supportFragmentManager.findFragmentById(R.id.nav_host_fragment) as NavHostFragment
val navController = navHostFragment.navController
val appBarConfiguration = AppBarConfiguration(setOf(R.id.nav_camera, R.id.nav_gallery))
```

The `onCreate` method now initialises a `NavHostFragment` object, which will provide access to the `FragmentContainerView` widget from the `activity_main.xml` layout. The `FragmentContainerView` will allow the user to navigate between the destinations defined in the `mobile_navigation.xml` navigation graph via a `NavController` object. We also defined a variable called `appBarConfiguration`, which details the app's top-level destinations. A top-level destination is considered the origin of a navigation pathway. For example, the camera fragment and the gallery fragment are top-level destinations; however, the photo filter fragment is not a top-level destination because the user must navigate there via the gallery fragment. In this app, the `BottomNavigationView` widget will only list the top-level destinations. If the user navigates to a fragment that is not a top-level destination, then the app bar will display a back arrow. The back arrow will allow the user to return to the previous destination in the navigation graph.

## Summary

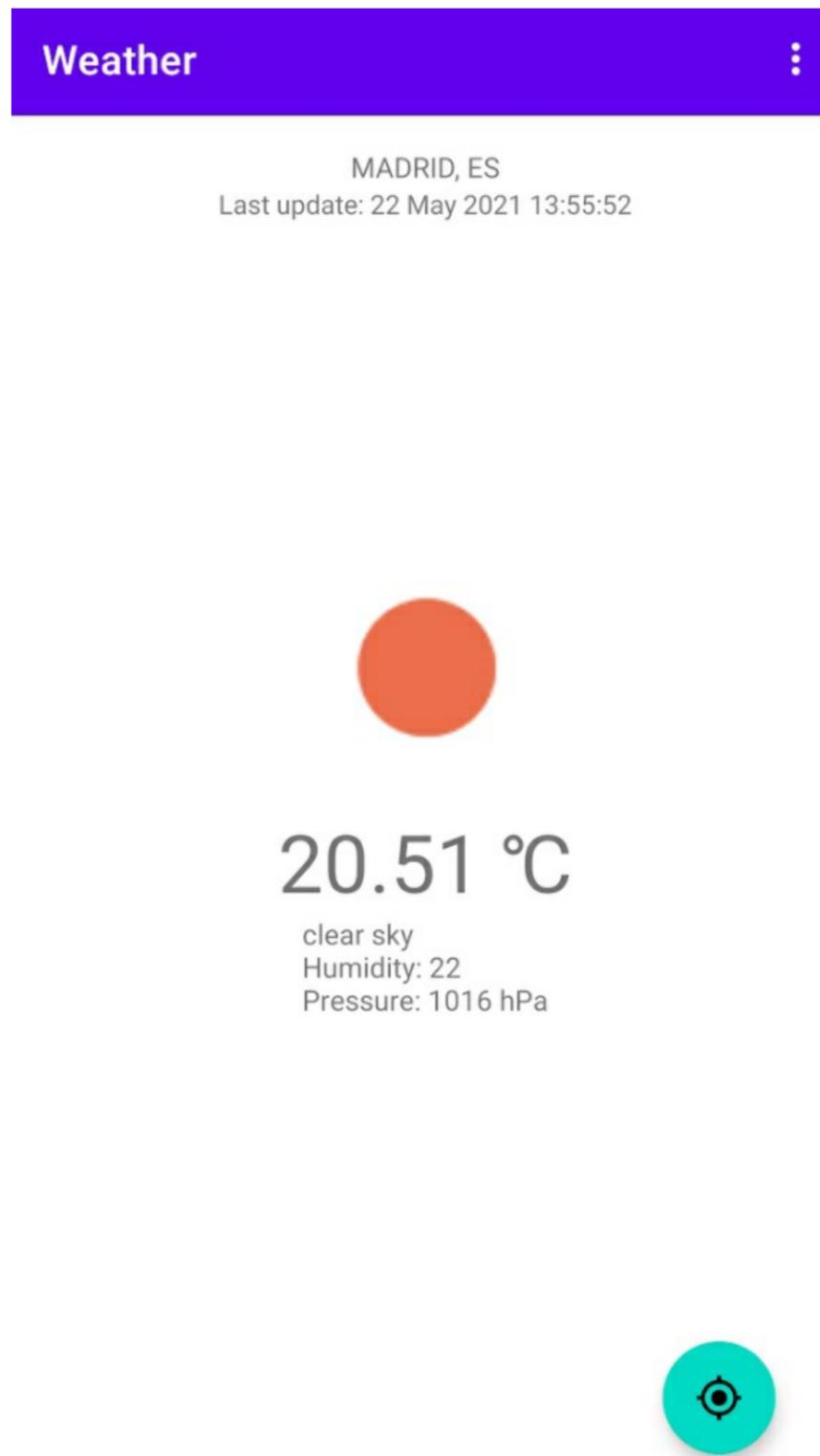
Congratulations on completing the Camera app! In creating the app, you have covered the following skills and topics:

- Create a new application using the Bottom Navigation Activity project template.
- Manage different fragments in an application.
- Use the CameraX library to display a live camera feed and capture photos.
- Access the MediaStore to find images on the user's device.
- Use the scoped storage framework to delete image files.
- Safely handle exceptions that might otherwise cause the app to crash.
- Map the app's destinations using a navigation graph.
- Create parcelable data classes, which can be transported around the app.
- Use the Safe Args plugin to send data between navigational destinations.
- Display images using a media management framework called Glide.
- Define a string array resource and load the array into a Spinner widget so the user can select an item from the array.
- Apply filters to images using an external Glide Transformations library.
- Save transformed images to the user's device.
- Design navigation graphs and specify which navigational components are top-level destinations.

# How to create a Weather application

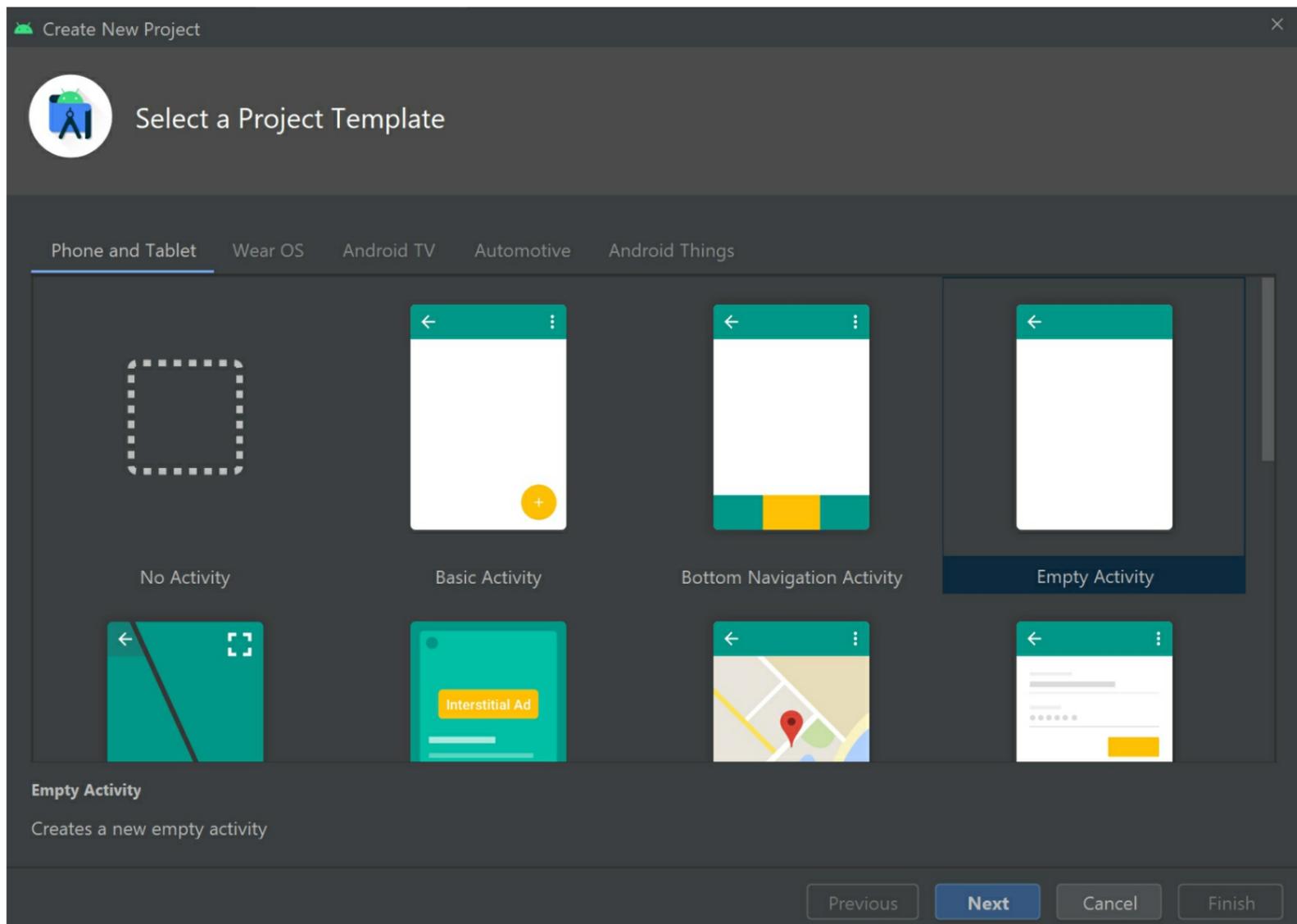
## Introduction

For this project, we will create an app that allows the user to view the weather for their current location or any other city on the planet. In creating this app, you will learn how to send and request data over the internet. You will also learn how to use Google's location services, request and process data from an online weather API, and incorporate swipe-to-refresh functionality that allows the user to refresh the weather data using a touchscreen gesture.

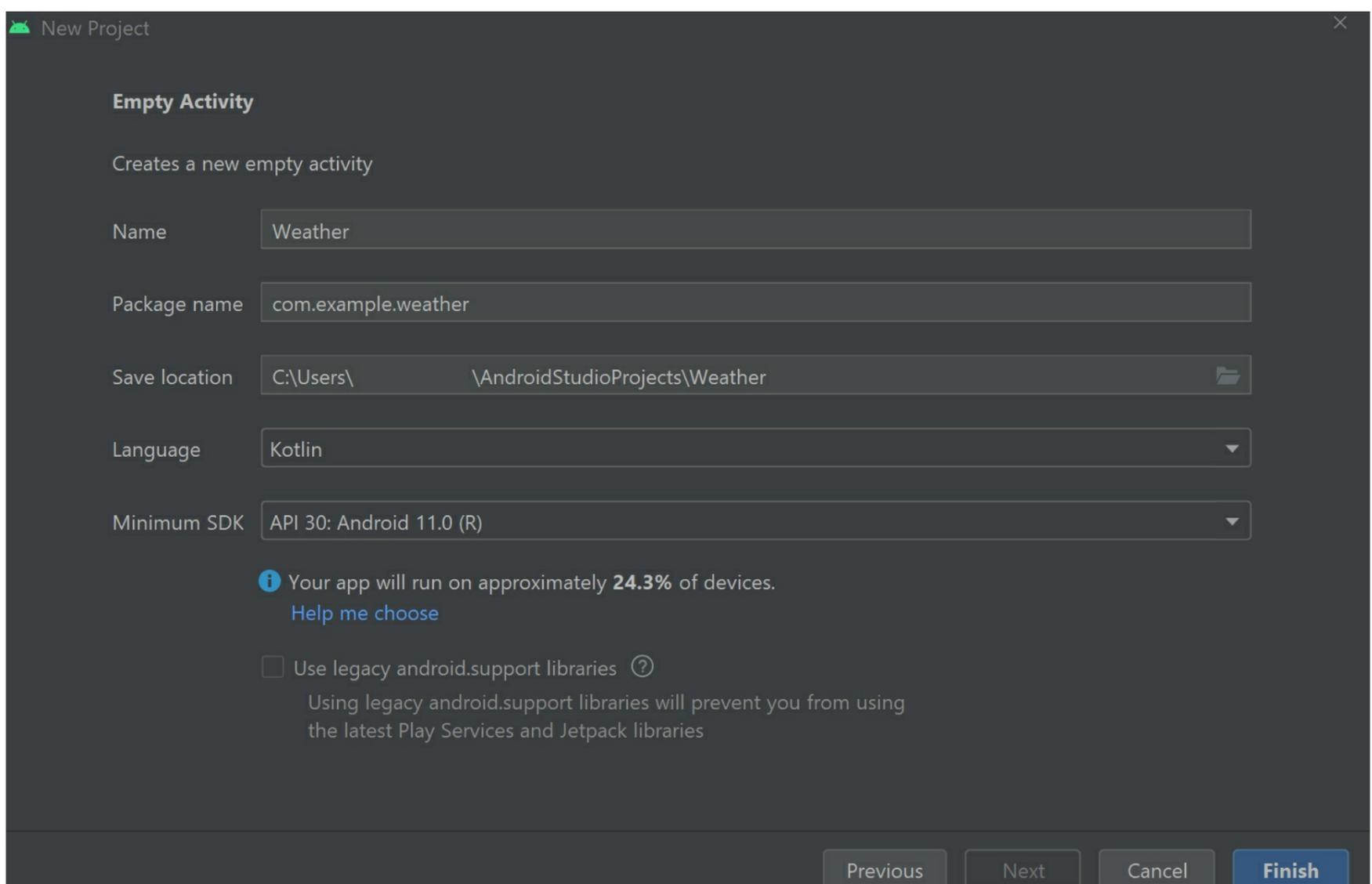


## Getting started

For this project, we will use the Empty Activity template. The Empty Activity template includes a MainActivity class and activity\_main layout but no additional features such as menus, fragments or navigation graphs. Typically, you would use the Empty Activity template for projects that do not require many readymade features and/or you expect to build most components from scratch.



In the Create New Project window that opens following the selection of the template, name the project Weather, set the language to Kotlin and choose API 30 as the minimum SDK. Next, press Finish to create the project.



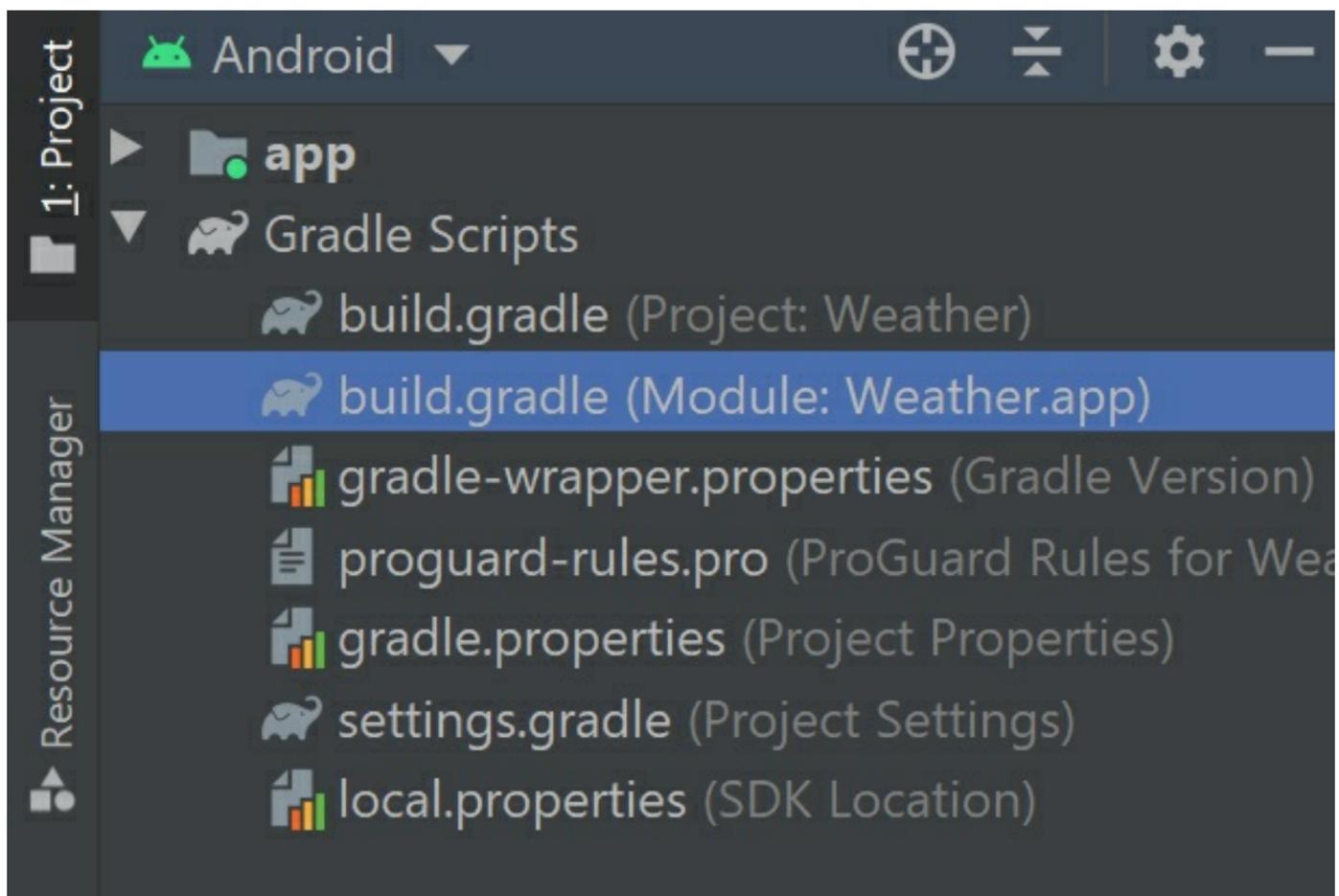
For all the projects in this book, it is recommended you enable Auto Imports to direct Android Studio to add any necessary import statements to your Kotlin files as you code. These import statements are essential for incorporating the external classes and tools required for the app to run. To enable Auto Imports, open Android Studio's Settings window by clicking **File** >

**Settings.** In the Settings window, navigate through **Editor > General > Auto Import** then select 'Add unambiguous imports on the fly' and 'Optimise imports on the fly' for both Java and Kotlin then press Apply and OK.

Android Studio should now add most of the necessary import statements to your Kotlin class files automatically. Sometimes there are multiple classes with the same name and the Auto Import feature will not work. In these instances, the requisite import statement(s) will be specified explicitly. You can also refer to the finished project code which accompanies this book to find the complete files including all import statements.

## Configuring the Gradle scripts and Manifest file

For the app to function correctly and perform all the required tasks, it must manually import several external packages using a toolkit called Gradle. To add the required packages, navigate through **Project > Gradle Scripts** and open the Module-level **build.gradle** file:



First, locate the android element and add the following code to enable view binding. View binding needs to be enabled manually for projects that use the Empty Activity template:

```
buildFeatures {  
    viewBinding true  
}
```

Next, add the following implementation statements to the dependencies element:

```
implementation 'androidx.preference:preference-ktx:1.2.0'  
implementation 'com.google.android.gms:play-services-location:19.0.1'  
implementation 'com.github.bumptech.glide:glide:4.11.0'  
implementation 'androidx.swiperefreshlayout:swiperefreshlayout:1.1.0'
```

The above implementation statements import the following tools: a shared preferences file that can store user preferences, such as their location preference for weather reports; Google Play's location services, which the app can use to locate the user's device; an image loading framework called Glide, which you may remember from the Camera app; and a swipe refresh layout widget, which will allow the user to refresh the weather data by swiping down from the top of their screen. After making the above changes, remember to re-sync the project when prompted!

Gradle files have changed since last project sync. A project sync may be necessary for the IDE ... [Sync Now](#)

Let's now turn our attention to the application's manifest file, which contains an overview of the app's activities,

services and user permissions requirements. Open the **AndroidManifest.xml** file by navigating through **Project > app > manifests** then add the following code above the application element:

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

The above code tells the device (and the Google Play store) that the app will require permission from the user to connect to the internet and access the device's location. In this app, we will request the `ACCESS_COARSE_LOCATION` permission, which grants access to the device's approximate location. The approximate location is sufficient for requesting weather data; however, if you create an app that requires the user's precise location then you should also request the `ACCESS_FINE_LOCATION` permission. Even if you request the fine location permission, you must still request the coarse location permission also because new versions of Android will give the user the option to share their approximate location only.

## Defining the string resources used in the app

Each item of text that will be displayed to the user should be stored as a string resource. A single string can be used in multiple locations across the app, and this makes it easier to edit the text when needed because you only have to change one string resource, rather than searching for each instance of the text throughout the app. Also, string resources are helpful when it comes to releasing the app in multiple languages because you can define translations for each string.

Android Studio will automatically generate a **strings.xml** resource file to store your strings when you create a new project. To locate the **strings.xml** file, navigate through **Project > app > res > values**. Open the file and edit its contents so it reads as follows:

```
<resources>
  <string name="app_name">Weather</string>
  <string name="change_city">Change city</string>
  <string name="refresh">Refresh</string>
  <string name="data_not_found">Sorry, no weather data found.</string>
  <string name="city_field">%1$s, %2$s</string>
  <string name="go">Go</string>
  <string name="details_field">%1$s
  \nHumidity: %2$s
  \nPressure: %3$s hPa</string>
  <string name="temperature_field"> %1$.2f °C </string>
  <string name="updated_field">Last update: %1$s</string>
  <string name="use_current_location">Use my current location</string>
  <string name="weather_icon">Image of current weather</string>
  <string name="permission_required">Permission required to retrieve the weather for your location.</string>
</resources>
```

Each string resource contains a name attribute that will be used to access the string from elsewhere in the project. The text that will be displayed is then input between the opening and closing string tags. You may notice that the `details_field` string includes the code `\n`. The `\n` code inserts a line break, so any subsequent text in the string will occupy a new line.

Several strings contain code in the format `'%1$s'`. These sections of code represent arguments that must be supplied when the string is created. The `'%1'` part represents the argument number, so will increase incrementally for each new argument that is added to the string e.g. `'%1'`, `'%2'`, `'%3'` etc. The second part of the argument indicates what data type is expected: `'$s'` represents a string, while `'$d'` represents a decimal integer and `'$.nf'` represents a floating number rounded to a certain number of decimal places (replace `'n'` with the number of decimal places e.g. `'$.2f'`). Returning to the `city_field` string, we can see it expects two string arguments. You can use Kotlin code to build the string and supply the argument values in the following way:

```
getString(R.string.city_field,"London", "England")
```

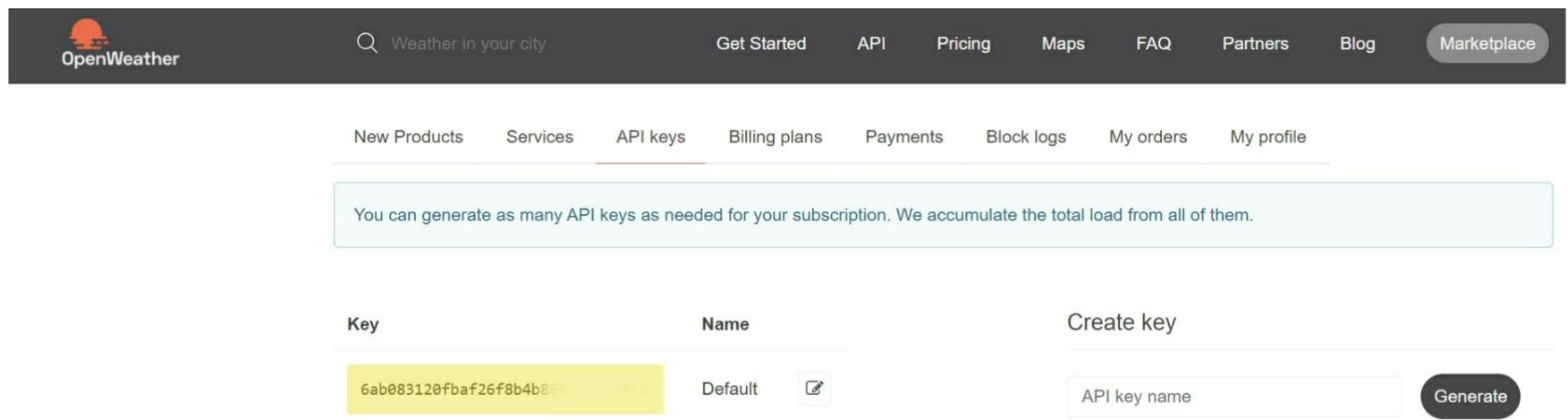
The above line imports the `city_field` string and supplies "London" as argument 1 and "England" as argument 2. The `city_field` string separates the two arguments using a comma and space, so the final output would be "London, England".

## Setting up the weather API

The app will retrieve the weather for the user's selected location from an online weather data service called OpenWeather (<https://openweathermap.org/guide>). OpenWeather offers a free weather API (Application Programming Interface) that you can request weather data from, as well as several paid subscriptions for higher volume requests. For this project, the free version is sufficient.

To request data from the OpenWeather API, you must create an account with them by visiting the following link: [https://home.openweathermap.org/api\\_keys](https://home.openweathermap.org/api_keys). Once your account is set up, you should be provided with an API access key. Make a note of your default key because you will need to copy and paste it into your application. The key will allow your app to communicate with the OpenWeather API and request weather data.

*Note: It can sometimes take several hours following the creation of your OpenWeather account for your API key to activate. You can continue with this project as normal; however, please be aware your application may not work right away and you may need to wait up to 24 hours for your API key to become operational.*



Next, return to Android Studio and open the **MainActivity.kt** class file (**Project** > **app** > **java** > **name of your project**). Add the following code above the onCreate method:

```
companion object {  
    // TODO: Get your API key here https://home.openweathermap.org/api_keys  
    // Note it can sometimes take a couple of hours following email confirmation for API keys to become active  
    private const val APP_ID = "INSERT-API-KEY-HERE"  
    private const val CITY_NAME_URL = "https://api.openweathermap.org/data/2.5/weather?q=""  
    private const val GEO_COORDINATES_URL = "https://api.openweathermap.org/data/2.5/weather?lat=""  
}
```

The above code defines a companion object, which contains several static variables that will be initialised when the activity is created. For the APP\_ID variable, replace INSERT-API-KEY-HERE with your API key from [https://home.openweathermap.org/api\\_keys](https://home.openweathermap.org/api_keys). The app will use your API key to request weather data from OpenWeather. The next two variables, CITY\_NAME\_URL and GEO\_COORDINATES\_URL, define the beginning of web URLs that will be used to request weather data for a given city or geographical location, respectively. The ending of the URLs will be supplied elsewhere in the code based on the user's input.

## Designing the activity\_main layout

In this section, we will design the layout resource file that will display weather data to the user. For this purpose, we will use a layout file called **activity\_main.xml** that should have been automatically created by Android Studio and will serve as the main layout file for the application. Locate the file by navigating through **Project** > **app** > **res** > **layout** and open it in Code view. We'll design this layout from scratch, so replace all the code in the file with the following:

```
<?xml version="1.0" encoding="utf-8"?>  
<androidx.swiperefreshlayout.widget.SwipeRefreshLayout xmlns:app="http://schemas.android.com/apk/res-auto"  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
  
    <androidx.constraintlayout.widget.ConstraintLayout  
        android:layout_width="match_parent"  
        android:layout_height="match_parent"  
        android:padding="16dp">
```

```
<TextView
    android:id="@+id/txtCity"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

```
<TextView
    android:id="@+id/txtUpdated"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="13sp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@id/txtCity" />
```

```
<ImageView
    android:id="@+id/imgWeatherIcon"
    android:layout_width="140dp"
    android:layout_height="140dp"
    android:layout_marginBottom="150dp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toBottomOf="parent"
    android:contentDescription="@string/weather_icon" />
```

```
<TextView
    android:id="@+id/txtTemperature"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textSize="40sp"
    app:layout_constraintTop_toBottomOf="@+id/imgWeatherIcon"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent" />
```

```
<TextView
    android:id="@+id/txtDetails"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/txtTemperature" />
```

```
<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="16dp"
    android:contentDescription="@string/use_current_location"
    app:srcCompat="@drawable/ic_location"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent" />
```

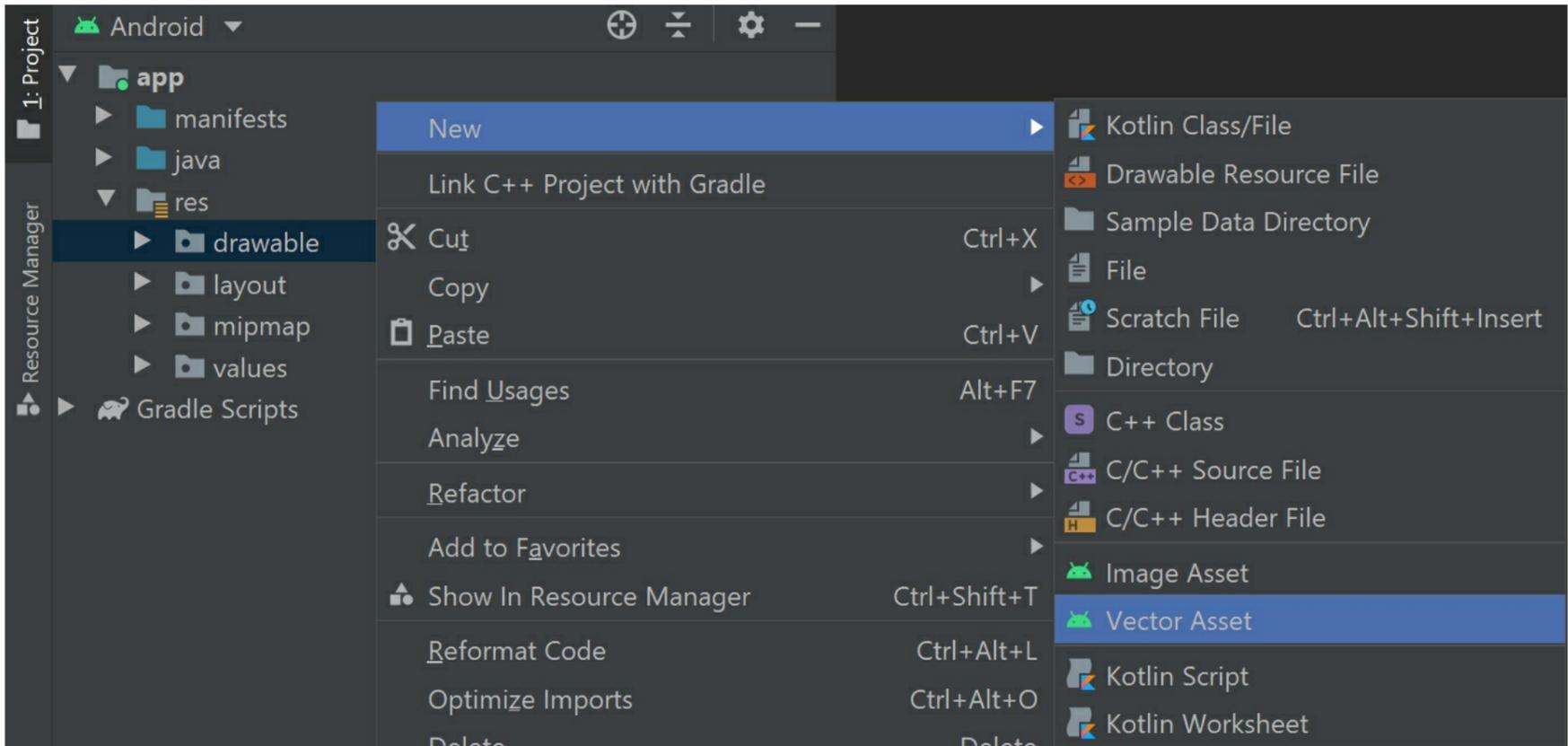
```
</androidx.constraintlayout.widget.ConstraintLayout>
</androidx.swiperefreshlayout.widget.SwipeRefreshLayout>
```

The root element of the activity\_main layout is a SwipeRefreshLayout widget. The SwipeRefreshLayout widget will allow the user to refresh the weather data using a swipe down gesture. Inside the SwipeRefreshLayout is a ConstraintLayout widget, which will organise the layout's content. The first two widgets inside the ConstraintLayout are TextView widgets, which will display the name of the city that the weather data refers to and the time the data was last updated, respectively. Next, the ConstraintLayout features an ImageView widget that will display an icon representing the current weather (e.g. sunny, cloudy etc.). The start (left), end (right), top and

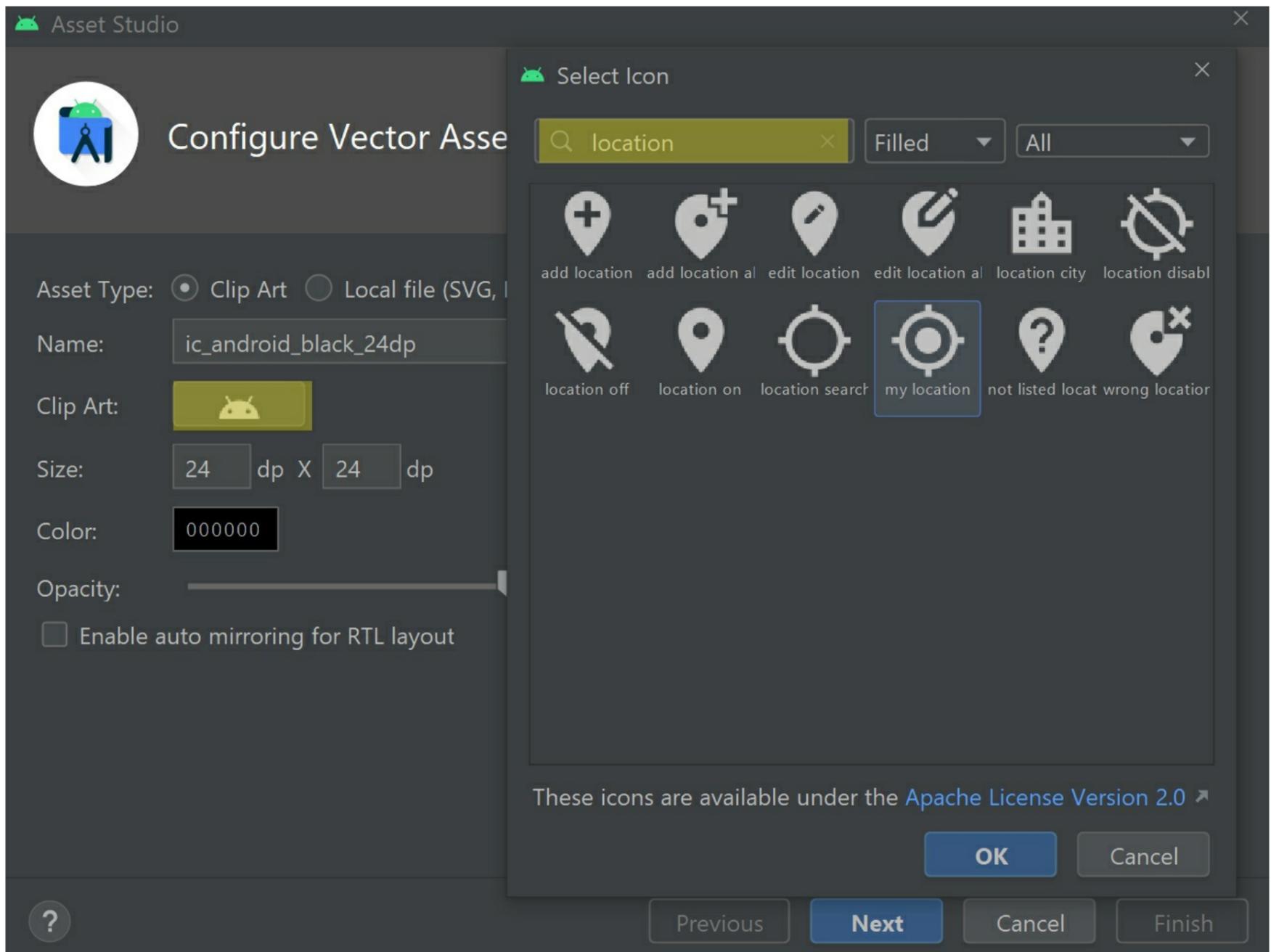
bottom edges of the ImageView widget are constrained to the parent container, which in this case is the ConstraintLayout. These equal and opposite constraints ensure the ImageView widget will position itself in the centre of the layout.

Below the ImageView widget are two further TextView widgets, which will display the current temperature and a description of the weather, respectively. Finally, there is a FloatingActionButton widget, which will load weather data for the user's current location when clicked. All widgets (except the floating action button) will align themselves centrally down the middle of the layout. The reason for this is that the widgets have equal and opposing start and end constraints linked to the parent ConstraintLayout container. In contrast, the floating action button will position itself in the bottom right corner because it is constrained to the bottom and end of the ConstraintLayout.

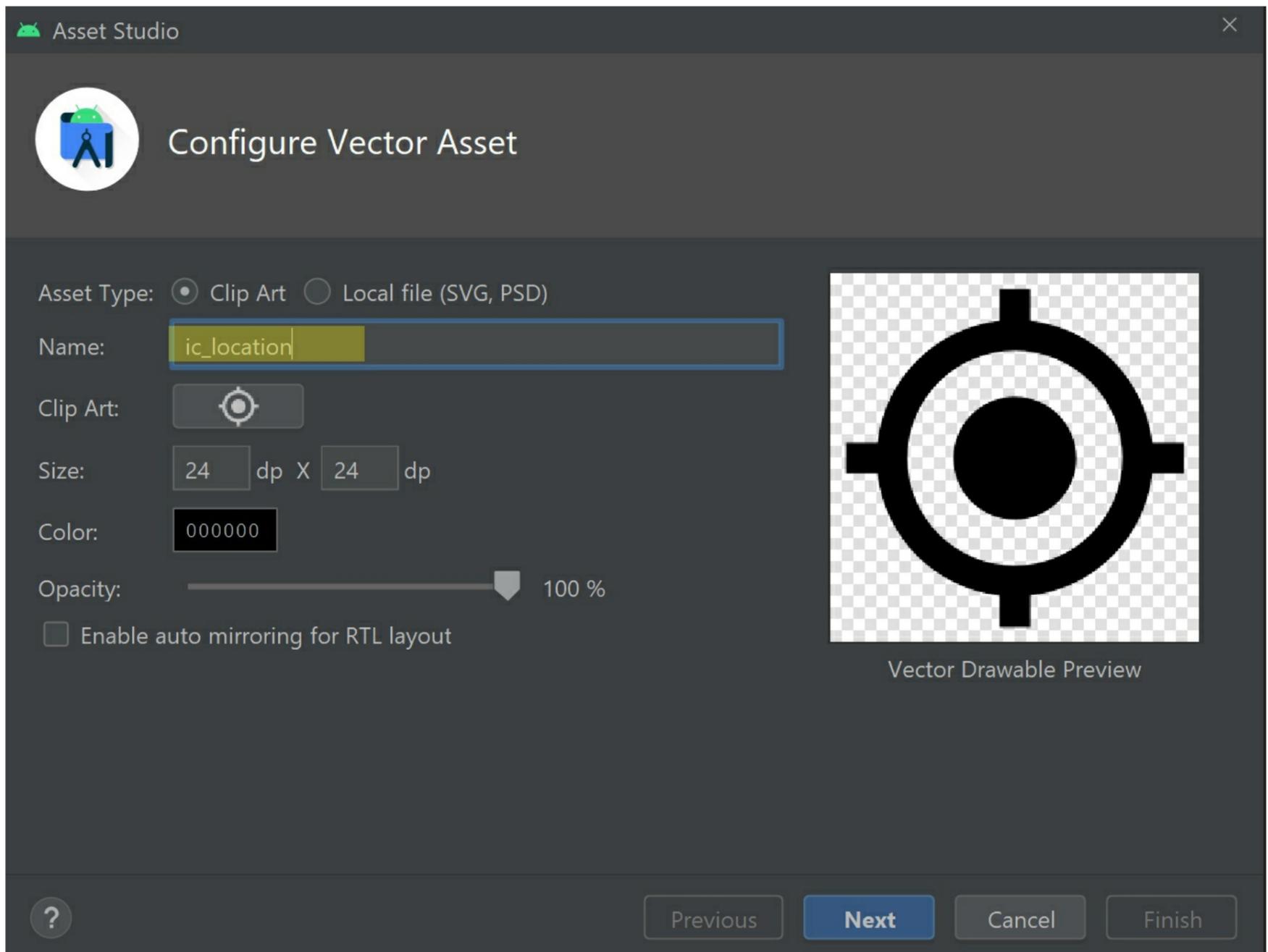
The floating action button features a srcCompat attribute which defines an icon that will appear in the button. The icon will be stored as a drawable resource. To create a new drawable resource, navigate through **Project > app > res** and right-click the **drawable** directory. Next, select **New > Vector Asset**.



In the Asset Studio window which opens, click the Clip Art and search for an icon called my location. Select the icon then press OK.



Next, set the name of the asset to `ic_location` then press Next followed by Finish to save the icon as a drawable resource.



## Finding the user's location

The user can request weather data for their current location by clicking the `FloatingActionButton` widget in the `activity_main.xml` layout. To enable this functionality, we must configure the `MainActivity` class to interact with the `activity_main` layout via the layout's binding class. First, return to the `MainActivity.kt` file (**Project > app > java > name of your project**) and add the following variables to the top of the class above the `onCreate` method:

```
private lateinit var binding: ActivityMainBinding
private lateinit var fusedLocationClient: FusedLocationProviderClient
private lateinit var sharedPreferences: SharedPreferences
```

Note you may need to add the following import statement to the top of the file:

```
import android.content.SharedPreferences
```

The above code defines several variables. The first variable provides access to the contents of the `activity_main.xml` layout via its binding class. Next, the `fusedLocationClient` variable will hold an instance of the `FusedLocationProviderClient` class. The `FusedLocationProviderClient` class interacts with Google's Fused Location Provider API, which is a battery-efficient tool for finding the user's location based on GPS and WiFi data. The third variable will store an instance of the `SharedPreferences` class, which we will use to store the user's preferences within the app.

All variables feature the `lateinit` modifier, which means we must initialise them elsewhere in the code before they can be used. To initialise the variables, locate the `MainActivity` class's `onCreate` method and replace `setContentView(R.layout.activity_main)` with the following code:

```
binding = ActivityMainBinding.inflate(layoutInflater)
setContentView(binding.root)

fusedLocationClient = LocationServices.getFusedLocationProviderClient(this)
sharedPreferences = PreferenceManager.getDefaultSharedPreferences(this)
```

```
binding.fab.setOnClickListener {
    getLocation()
}
```

Note you may need to add the following import statement to the top of the file:

```
import androidx.preference.PreferenceManager
```

The above code initialises the activity\_main layout's binding class. It also sets the content view of the activity to the activity\_main layout's root element (the SwipeRefreshLayout widget) so the layout and all its constituent widgets will be visible to the user. The code also initialises the fusedLocationClient and sharedPreferences variables so the app can interact with Google's Fused Location Provider API and the shared preferences file. Finally, the code assigns an onClick listener to the FloatingActionButton widget from the activity\_main layout. If the button is clicked, then a method called getLocation will load the weather data for the user's current location. To define the getLocation method, add the following code below the onCreate method:

```
private fun getLocation() {
    if (ContextCompat.checkSelfPermission(applicationContext,
        android.Manifest.permission.ACCESS_COARSE_LOCATION) !=
        PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(this,
            arrayOf(android.Manifest.permission.ACCESS_COARSE_LOCATION), 1)
    } // TODO: Load the user's location here
}
```

The getLocation method checks whether the app has permission to access the device's location. If permission has not been granted ('!=' is the not-equal-to operator) then the getLocation method will request the user's permission to access the device's location. Once the user responds to the permission request, the outcome will be handled by a method called onRequestPermissionsResult which you can define by adding the following code below the onCreate method:

```
override fun onRequestPermissionsResult(requestCode: Int, permissions: Array<out String>, grantResults:
IntArray) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults)
    if (requestCode == 1 && grantResults[0] == PackageManager.PERMISSION_GRANTED) getLocation()
    else if (requestCode == 1 && grantResults[0] != PackageManager.PERMISSION_GRANTED) Toast.makeText(
        this,
        getString(R.string.permission_required),
        Toast.LENGTH_LONG
    ).show()
}
```

The onRequestPermissionsResult method will run when the user responds to a permission request. If the result of the request is PERMISSION\_GRANTED, then the above code runs the getLocation method again because it should now have the requisite user permissions. On the other hand, if the user refuses permission, then a toast notification will inform the user that the app requires permission to find the weather data for their location.

Let's now return to the getLocation method and write the code that will find the user's location if the necessary permissions have been granted. In the getLocation method, replace the TODO comment with the following code:

```
else {
    fusedLocationClient.lastLocation.addOnSuccessListener { location: Location? ->
        location?.let {
            val apiCall = GEO_COORDINATES_URL + location.latitude + "&lon=" + location.longitude
            updateWeatherData(apiCall)
            sharedPreferences.edit().apply {
                putString("location", "currentLocation")
            }.apply()
        }
    }
}
```

The above code uses the Fused Location Client class to find the last known location of the user's device. Once the

request has been processed, the `onSuccess` listener callback will return the location as a `Location` object. `Location` objects contain a variety of details about the location, including the longitude and latitude geographical coordinates. In rare cases, the location returned by the Fused Location Client is null. For this reason, the remainder of the code is wrapped in a null-safe `let` block, as indicated by the `?.let` section of code. The code inside the `let` block will only run if the value of the location parameter is non-null. First, the `let` block prepares the URL that will be used to request data from the OpenWeather API. The URL will start with the contents of the `GEO_COORDINATES_URL` variable from the companion object and end with the latitude and longitude coordinates for the user's location. Altogether, the complete URL will look like this: `https://api.openweathermap.org/data/2.5/weather?lat=35&lon=139`.

Once the URL has been constructed, a method called `updateWeatherData` (which we'll define shortly) will initiate the API call and process the results. The `getLocation` method finishes by writing a value of "currentLocation" under the key "location" to the app's shared preferences file. The app will retrieve the value of the "location" key whenever it launches, and a value of "currentLocation" will tell the app that it should attempt to load the weather data for the user's current location, rather than a specific city. Whenever you update the shared preferences file, you must use the `apply` method as shown above to finalise the changes and close the shared preferences file editor.

## Requesting data from the OpenWeather API

In this section, we will cover how to retrieve weather data from the OpenWeather API. First, staying with the `MainActivity` class, add the following code below the `getLocation` method to define a method called `updateWeatherData` that will handle calls to the OpenWeather API:

```
private fun updateWeatherData(apiCall: String) {
    object : Thread() {
        override fun run() {
            val jsonObject = getJSON(apiCall)
            runOnUiThread {
                if (jsonObject != null) renderWeather(jsonObject)
                else Toast.makeText(this@MainActivity, getString(R.string.data_not_found),
                    Toast.LENGTH_LONG).show()
            }
        }
    }.start()
}
```

The `updateWeatherData` method constructs a `Thread` object, which is used to coordinate a workflow in parallel with other app processes. It is advantageous to make API calls from a `Thread` because we do not know how long it will take to receive a response from the web server. Other app processes mustn't be blocked if there is a delay in receiving a response. The API call is initiated in the `Thread` using a method called `getJSON`. The `getJSON` method contacts the API and returns the results as a JSON string. Next, the `runOnUiThread` block dispatches the JSON string to the main application thread for processing. Only the main thread can interact with the user interface, so it is important to return to this thread once the API call is complete so we can display the weather data to the user.

The `runOnUiThread` block first checks whether the API call has returned a JSON String. If a response has been received, then it is processed by a method called `renderWeather`; however, if no output is received, then this means something has gone wrong with the API call and a toast notification informs the user that the app was unable to retrieve the weather data.

To define the `getJSON` method and request weather data from the OpenWeather API, add the following code below the `updateWeatherData` method:

```
private fun getJSON(apiCall: String): JSONObject? {
    try {
        val con = URL("$apiCall&appid=$APP_ID&units=metric").openConnection() as HttpURLConnection
        con.apply {
            doOutput = true
            connect()
        }

        val inputStream = con.inputStream
        val br = BufferedReader(InputStreamReader(inputStream!!))
        var line: String?
        val buffer = StringBuffer()
    }
}
```

```
while (br.readLine().also { line = it } != null) buffer.append(line + "\n")
inputStream.close()
con.disconnect()
```

```
val jsonObject = JSONObject(buffer.toString())
```

```
return if (jsonObject.getInt("cod") != 200) null
```

```
else jsonObject
```

```
} catch (t: Throwable) {
```

```
return null
```

```
}
```

Note you may need to add the following import statement to the top of the file:

```
import java.net.URL
```

The getJSON method uses the URL sent by the updateWeatherData method to establish a connection with the OpenWeather API. Before the API call is initiated, the URL must be modified to include the application's API key and weather measurement unit. In this case, the unit is set to metric, which means temperature data will be reported in degrees Celsius (see the OpenWeather API documentation for information on other available units <https://openweathermap.org/current#data>). Next, an apply block is used to configure the URL connection. In Kotlin, apply blocks apply a section of code to an object. The above code uses the apply block to customise the HttpURLConnection object that is stored in the con variable. For example, the doOutput field of the HttpURLConnection object is set to true to indicate that we will use the connection to request data. Next, the HttpURLConnection class's connect method will initiate the connection and request the weather data.

The remainder of the getJSON method processes the output from the API request. First, an InputStream object captures the flow of data. The data is then decoded by the BufferedReader class and transformed into text. The text is processed line-by-line using the BufferedReader class's readLine method, which extracts each line of text as a string. The above code interacts with the string using an also block. In Kotlin, an also block allows you to interact with an object without modifying the source of the object. For example, if an also block is applied to a variable, then the variable's value will be unchanged by the code in the also block. In this case, the also block assigns the string to a variable called line. If the extracted string is not null, then it is incorporated into an instance of the StringBuffer class, which combines fragments of text into a single string. Once all of the data has been processed, the InputStream instance and URL connection are closed to prevent memory leaks.

An example of the full JSON String we might receive from the OpenWeather API is provided below:

```
{
  "coord": {
    "lon": -122.08,
    "lat": 37.39
  },
  "weather": [
    {
      "id": 800,
      "main": "Clear",
      "description": "clear sky",
      "icon": "01d"
    }
  ],
  "base": "stations",
  "main": {
    "temp": 7,
    "pressure": 1012,
    "humidity": 81,
    "temp_min": 5,
    "temp_max": 8
  },
  "visibility": 16093,
  "wind": {
```

```

    "speed": 1.5,
    "deg": 350
  },
  "clouds": {
    "all": 1
  },
  "dt": 1560350645,
  "sys": {
    "type": 1,
    "id": 5122,
    "message": 0.0139,
    "country": "US",
    "sunrise": 1560343627,
    "sunset": 1560396563
  },
  "timezone": -25200,
  "id": 420006353,
  "name": "Mountain View",
  "cod": 200
}

```

In addition to describing the weather, the JSON output also includes a field called ‘cod’, which is an internal parameter used by the OpenWeather API to determine whether the API call was successful. If the value of the cod field is 200 then that means the API call was processed correctly; however, if another value is reported then that means something went wrong. For this reason, the `getJSON` method will only return the JSON object if the cod value equals 200. Otherwise, the method will return a value of null because the API call was unsuccessful. This entire process is enclosed in a try/catch block that will intercept any errors that may occur. For example, there could be an error establishing a connection to the URL. If an exception is thrown, then the catch block is triggered and the `getJSON` method will return a value of null.

## Displaying weather data to the user

In this section, we will process the JSON output from the OpenWeather API and display the weather data to the user. This processing is handled by a method called `renderWeather`, which uses the data from the JSON object to update the user interface and display information about the weather. To define the `renderWeather` method, add the following code below the `getJSON` method:

```

private fun renderWeather(json: JSONObject) {
  try {
    val city = json.getString("name").uppercase(Locale.US)
    val country = json.getJSONObject("sys").getString("country")
    binding.txtCity.text = resources.getString(R.string.city_field, city, country)

    val weatherDetails = json.optJSONArray("weather")?.getJSONObject(0)
    val main = json.getJSONObject("main")
    val description = weatherDetails?.getString("description")
    val humidity = main.getString("humidity")
    val pressure = main.getString("pressure")
    binding.txtDetails.text = resources.getString(R.string.details_field, description, humidity, pressure)

    // The backup icon is 03d (cloudy) for null results
    // Full list of icons available here https://openweathermap.org/weather-conditions#Weather-Condition-Codes-2
    val iconID = weatherDetails?.getString("icon") ?: "03d"
    val urlString = "https://openweathermap.org/img/wn/$iconID@2x.png"

    Glide.with(this)
      .load(urlString)
      .transition(DrawableTransitionOptions.withCrossFade())
      .centerCrop()
      .into(binding.imgWeatherIcon)

    val temperature = main.getDouble("temp")

```

```
binding.txtTemperature.text = resources.getString(R.string.temperature_field, temperature)
```

```
val df = DateFormat.getDateTimeInstance()
```

```
val lastUpdated = df.format(Date(json.getLong("dt") * 1000))
```

```
binding.txtUpdated.text = resources.getString(R.string.updated_field, lastUpdated)
```

```
} catch (ignore: Exception) { }
```

Note you may need to add the following import statement to the top of the file:

```
import java.text.DateFormat
```

The `renderWeather` method extracts information from the JSON output. An example of the JSON output was included at the end of the previous section. For instance, the city name the weather data refers to is stored under the key 'name' in the JSON object. The city name is stored as a string, and so can be retrieved from the JSON object using the `getString` method. For display purposes, the city name is converted to uppercase. Next, the country code is retrieved. Referring back to the example JSON output, the 'country' key is inside a nested JSON object called 'sys', as indicated by the curly brackets `{}` after the `sys` key. Hence, to retrieve the country code, we must first access the `sys` JSON object before using the `getString` method to retrieve the 'country' key/value pair. Once the city and country have been retrieved, they are incorporated into the `city_field` string resource and displayed in the `txtCity` `TextView` from the `activity_main` layout.

A similar process is used to display the weather data. Referring back to the JSON output, the weather data is stored under the 'weather' key in a JSON array, as indicated by the square `[]` brackets. The JSON array will contain one or more JSON objects. In this instance, each object describes a type of weather that applies to the target location. Often, there will only be one applicable weather type. For this reason, the above code uses the `getJSONObject(0)` method to retrieve the first JSON object from the array only.

Another field of data that we will use is stored under the key 'icon'. The icon key holds a code that can be used to load an image representing the current weather state. The full list of codes and their corresponding icons can be found in the OpenWeather API documentation: <https://openweathermap.org/weather-conditions>. To access the weather icon for a given icon code, simply replace **\$iconID** in the following URL with the icon code: [https://openweathermap.org/img/wn/\\$iconID@2x.png](https://openweathermap.org/img/wn/$iconID@2x.png). In the above code, we use an image rendering framework called Glide to load the image associated with the URL into the `imgWeatherIcon` `ImageView` widget from the `activity_main` layout. Also, we instruct Glide to crop the icon so it fits in the center of the `ImageView`, and apply a crossfade animation so the icon will fade into the `ImageView` when loaded.

Other items of weather information are extracted from the JSON output as required. For example, the temperature is listed as a field called 'temp' in a JSON object called 'main'. The temperature will be displayed using a string resource called `temperature_field`, which is defined in the `strings.xml` resource file as follows: `<string name="temperature_field"> %1$.2f °C </string>`. The `'%1$.2f'` part represents a Float parameter that will be rounded to two decimal places. In this case, the Float parameter will display the temperature, so the completed `temperature_field` string will be formatted like **12.83 °C** and displayed in the `txtTemperature` `TextView` widget.

Finally, the time the weather data was last updated is determined by extracting the value stored under the 'dt' key and formatting the value as a date/time instance. The `dt` key stores the duration in milliseconds that have passed since January 1st 1970, which is referred to as the epoch. The timestamp of the last update is loaded into `txtUpdated` `TextView`, thereby concluding the `renderWeather` method. All of the above code is enclosed in a try/catch block to intercept any exceptions that might otherwise cause the application to crash.

# Weather ⋮

MADRID, ES  
Last update: 22 May 2021 13:55:52



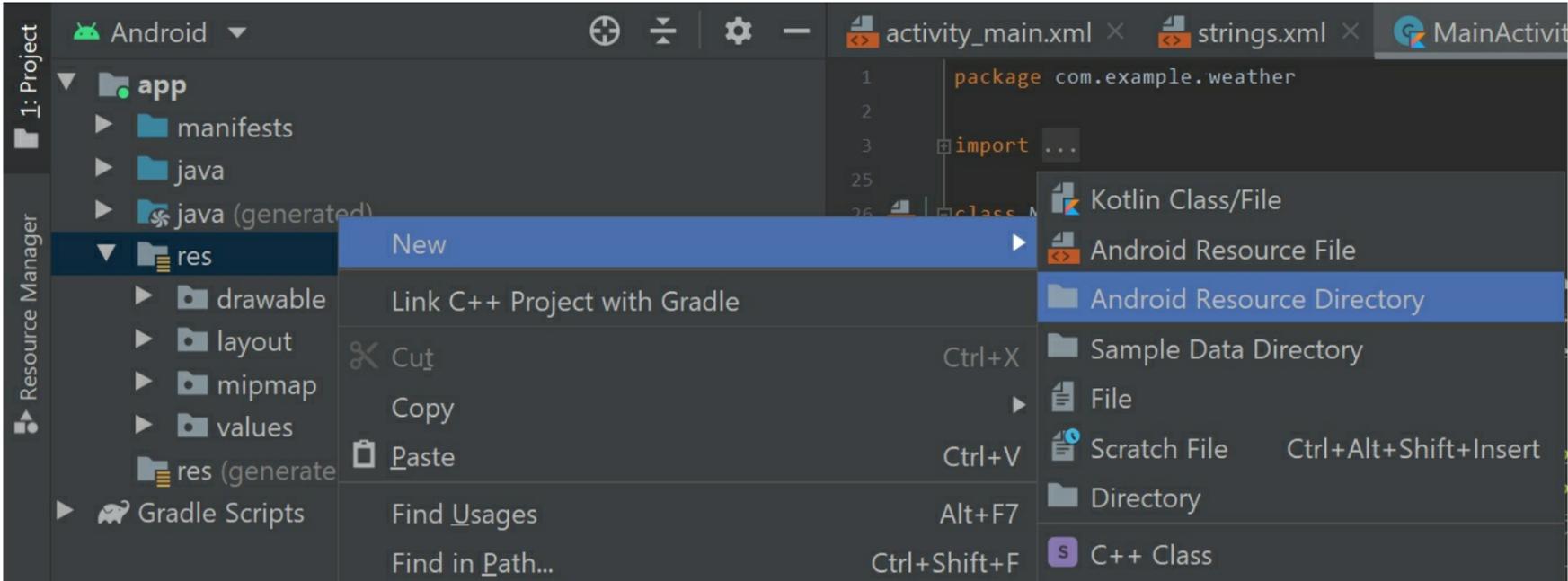
## 20.51 °C

clear sky  
Humidity: 22  
Pressure: 1016 hPa



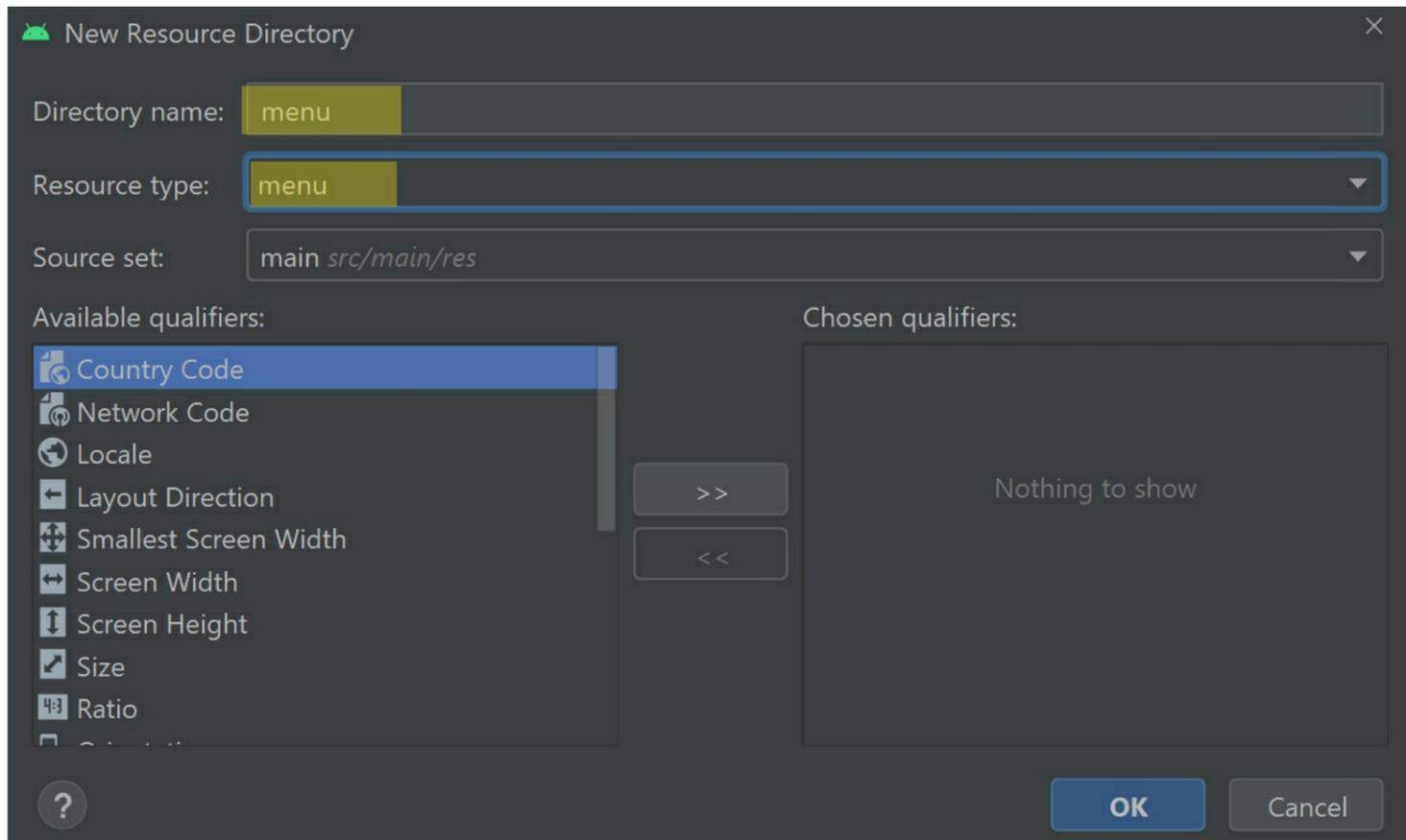
### Loading weather data for a specific city

In addition to loading the weather data for the user's current location, the app will also allow the user to view the weather for other cities around the world. If the user wishes to search for a city, then they can press a menu item in the toolbar to open a dialog window and search for the city name. To create the menu, right-click the **res** directory (**Project > app**) then select **New > Android Resource Directory**.

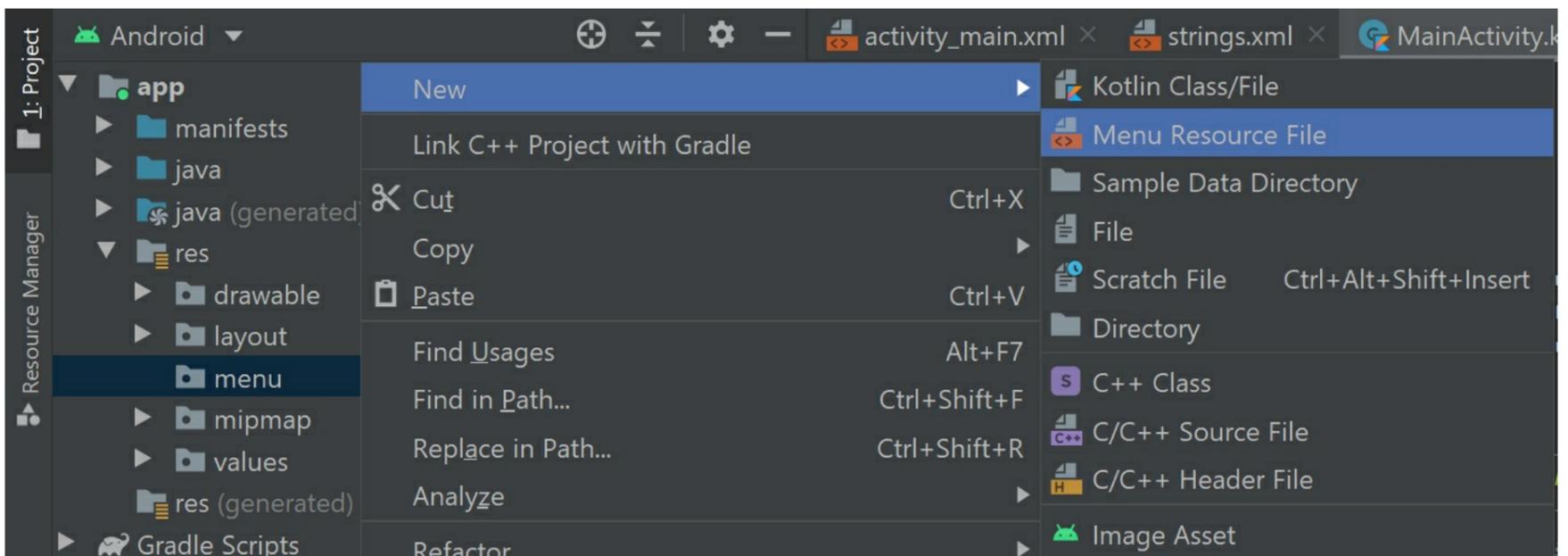


In the New Resource Directory window, set the directory name to menu and the resource type to menu then press

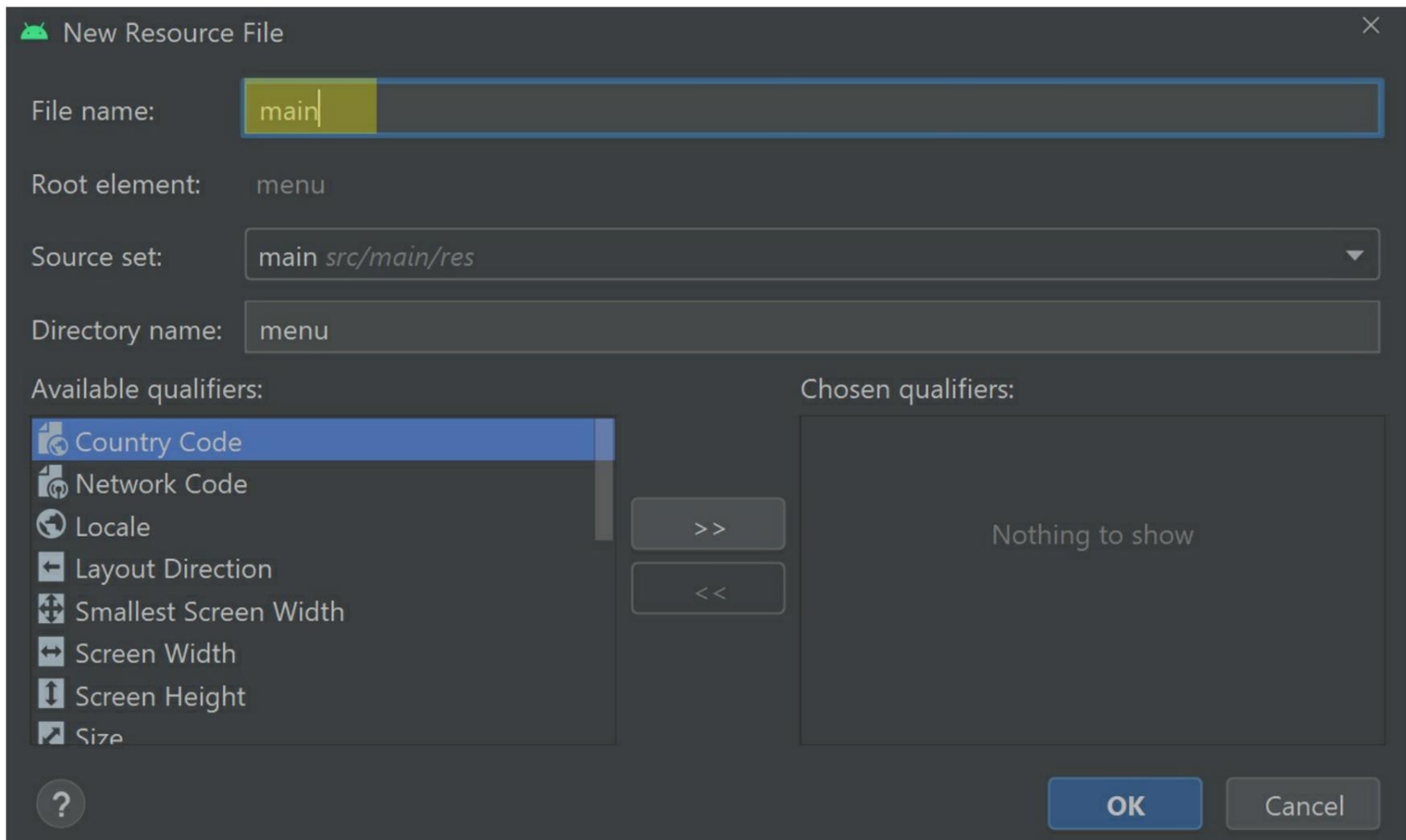
OK to create the directory.



Next, right-click the newly created **menu** directory and press **New > Menu Resource File**.



Set the file name to main then press OK to create a menu resource called **main.xml**.



Once the **main.xml** file is open in the editor, switch to Code view and edit the file so it reads as follows:

```
<?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/refresh"
        android:title="@string/refresh"
        app:showAsAction="never"/>

    <item
        android:id="@+id/change_city"
        android:title="@string/change_city"
        app:showAsAction="never"/>
</menu>
```

The above code defines a menu containing two items. The first item will refresh the weather data, while the second item will open a dialog window that allows the user to enter the name of a city they wish to view the weather for. To make the menu items operational, return to the MainActivity class and add the following code below the onCreate method:

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
    menuInflater.inflate(R.menu.main, menu)
    return super.onCreateOptionsMenu(menu)
}
```

The onCreateOptionsMenu method defined above uses the MenuInflater class to load the **main.xml** menu resource and interact with its items. To define what happens when an item is clicked, add the following code below the onCreateOptionsMenu method:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        // TODO: Respond to the refresh menu item here

        R.id.change_city -> showInputDialog()
    }
    return super.onOptionsItemSelected(item)
}
```

The `onOptionsItemSelected` method states that if the `change_city` menu item is clicked then a method called `showInputDialog` should run. The `showInputDialog` method will display a dialog window that allows the user to enter the name of the city they wish to view the weather data for. To define the `showInputDialog` method, add the following code below the `renderWeather` method:

```
private fun showInputDialog() {
    val input = EditText(this@MainActivity)
    input.inputType = InputType.TYPE_CLASS_TEXT

    AlertDialog.Builder(this).apply {
        setTitle(getString(R.string.change_city))
        setContentView(input)
        setPositiveButton(getString(R.string.go)) { _, _ ->
            val city = input.text.toString()
            updateWeatherData("$CITY_NAME_URL$city")
            sharedPreferences.edit().apply {
                putString("location", city)
            }
        }
    }
    show()
}
```

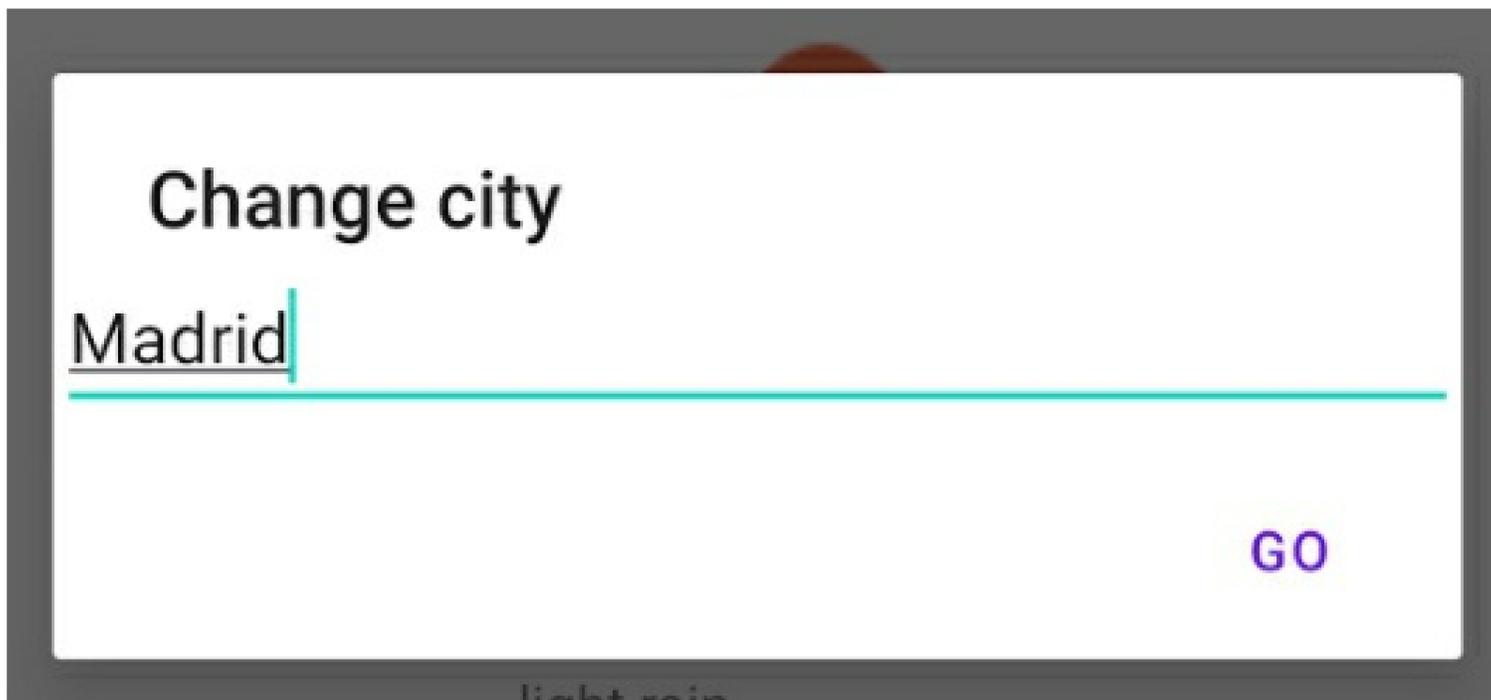
Note you may need to add the following import statement to the top of the file:

```
import androidx.appcompat.app.AlertDialog
```

The `showInputDialog` method must create the dialog window from scratch. First, it initialises an `EditText` widget, into which the user can type the city name. The `inputType` attribute of the `EditText` widget is set to `TYPE_CLASS_TEXT` to show that the user is expected to enter plain text. Next, the `showInputDialog` method constructs an instance of the `AlertDialog` class. The `AlertDialog` class prepares a dialog window featuring up to three buttons. These buttons include a positive button, that is used for affirmative actions; a negative button, that is used to dismiss the dialog window without any further action; and a neutral button, that is used to defer the user's final decision until later.

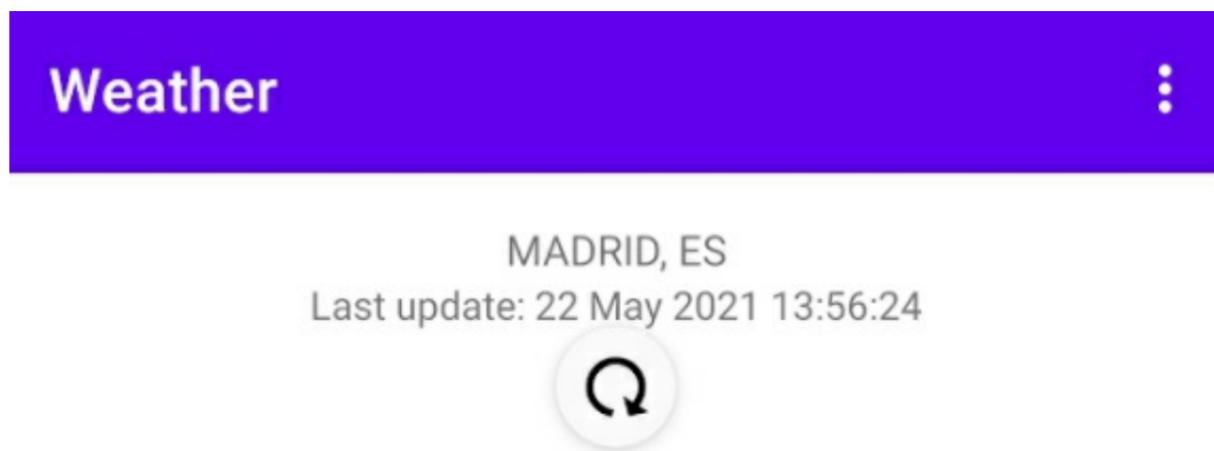
In the `AlertDialog` builder, the `AlertDialog` is assigned a title of 'Change city' which will appear at the top of the dialog window. Next, the `EditText` widget we designed earlier is added to the dialog window. Also, the `AlertDialog`'s positive action button is initialised. When clicked, the positive action button will retrieve the user's input from the `EditText` widget and append it to the `CITY_NAME_URL` variable from the `MainActivity` class's companion object. Once the city name is appended, the URL should read similar to `https://api.openweathermap.org/data/2.5/weather?q=london`. The completed URL is then sent to the `updateWeatherData` method which will request the weather data for the user's chosen city from the OpenWeather API. Finally, the user's chosen city is written into the shared preferences file, so the app knows which city to request weather data for when launched.

Once the title, contents and positive button are in place, the `AlertDialog` class's `show` method displays the dialog to the user. And with that, the user can now search for the weather for any city across the world.



## Enabling swipe-to-refresh

The user can refresh the weather data by swiping down from the top of their screen. This functionality is made possible because of the `SwipeRefreshLayout` widget in the `activity_main.xml` layout. The `SwipeRefreshLayout` widget detects swipe down gestures and responds with a predetermined action.



To define how swipe down gestures should be handled, add the following code to the bottom of the `onCreate` method in the `MainActivity` class:

```
binding.root.setOnRefreshListener {
    refreshData()
}
```

The above code assigns an `onRefresh` listener to the `SwipeRefreshLayout` widget (which is the root element of the `activity_main` layout). The listener will run a method called `refreshData` in response to swipe down gestures. To define the `refreshData` method, add the following code below the `showInputDialog` method:

```
private fun refreshData() {
    when (val location = sharedPreferences.getString("location", null)) {
        null, "currentLocation" -> getLocation()
        else -> updateWeatherData("$CITY_NAME_URL$location")
    }
    binding.root.isRefreshing = false
}
```

The `refreshData` method uses a `when` block to respond to data refresh requests based on the user's location preference. If the location preference is `null` or set to `"currentLocation"`, then the `getLocation` method will request weather data for the user's last known location. Meanwhile, if the location preference is set to a city name then the app will attempt to load the weather data for that city instead. Once the above processing is complete, the `SwipeRefreshLayout` widget's `isRefreshing` attribute is set to `false` to confirm all actions associated with the swipe down gesture have been executed. Setting the `isRefreshing` attribute to `false` also removes the progress bar to show the user that the page has finished refreshing.

Ordinarily, the `SwipeRefreshLayout` widget would display and remove the progress bar itself; however, we explicitly set its `isRefreshing` attribute to false because there is an alternative pathway through which the user can refresh the content. The alternative pathway is by selecting the refresh menu item in the app toolbar. This alternative pathway will help support users that depend on external accessibility devices and may be unable to perform the swipe gesture. If you recall when we created the `main.xml` menu resource file earlier, we defined an item called refresh. The refresh menu item will update the weather data similar to the swipe down gesture. To make the refresh menu item operational, refer to the `onOptionsItemSelected` method in the `MainActivity` class and replace the `TODO` comment with the following code:

```
R.id.refresh -> {  
    binding.root.isRefreshing = true  
    refreshData()  
}
```

The above code responds to clicks on the refresh menu item by setting the `isRefreshing` property of the `SwipeRefreshLayout` widget to true and running the `refreshData` method. And with that, the user can now refresh the weather data for their chosen location through either a swipe gesture or by clicking the refresh menu item in the app toolbar.

The last thing we need to do is run the `refreshData` command whenever the user returns to the app. This measure will ensure the user sees the most up to date weather data when they open the app. To implement this feature, add the following code below the `onCreate` method:

```
override fun onResume() {  
    super.onResume()  
  
    binding.root.isRefreshing = true  
    refreshData()  
}
```

The `onResume` method refers to a stage of the activity lifecycle. It will run after the `onCreate` and `onStart` stages when the activity is launched, and also when the user returns to the app after leaving it running in the background. The `onResume` stage occurs when the activity is ready to begin receiving user input. In this instance, we instruct the `onResume` stage to follow the same pathway as the refresh menu item: it will prompt the `SwipeRefreshLayout` widget to display the progress bar then run the `refreshData` method to retrieve the most up to date weather data. In this way, the app will attempt to fetch weather data for the user's chosen location whenever it is launched.

## Summary

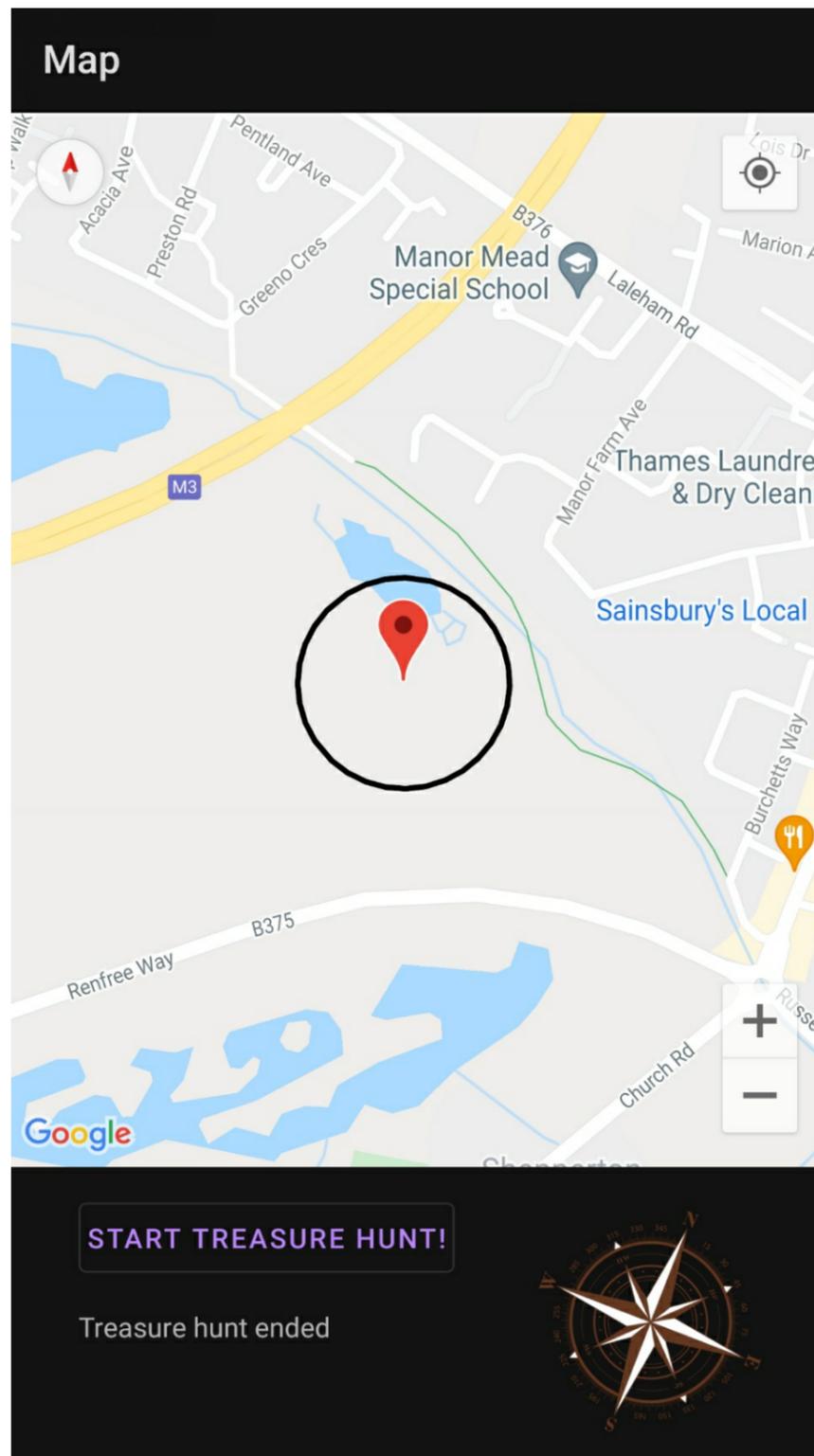
Congratulations on completing the Weather app! In creating this app, you have covered the following skills and topics:

- Create a new application using the Empty Activity project template.
- Build dynamic string resources that incorporate text-based and number-based parameters.
- Request data from an online API.
- Process JSON output and retrieve information.
- Utilise Google's Fused Location Provider API to find the user's last known location.
- Use Kotlin's let, apply and also scope functions to interact with an object while executing a block of code.
- Implement a SwipeRefreshLayout widget to respond to swipe gestures and display a progress bar when content is being refreshed.

# How to create a Treasure Map application

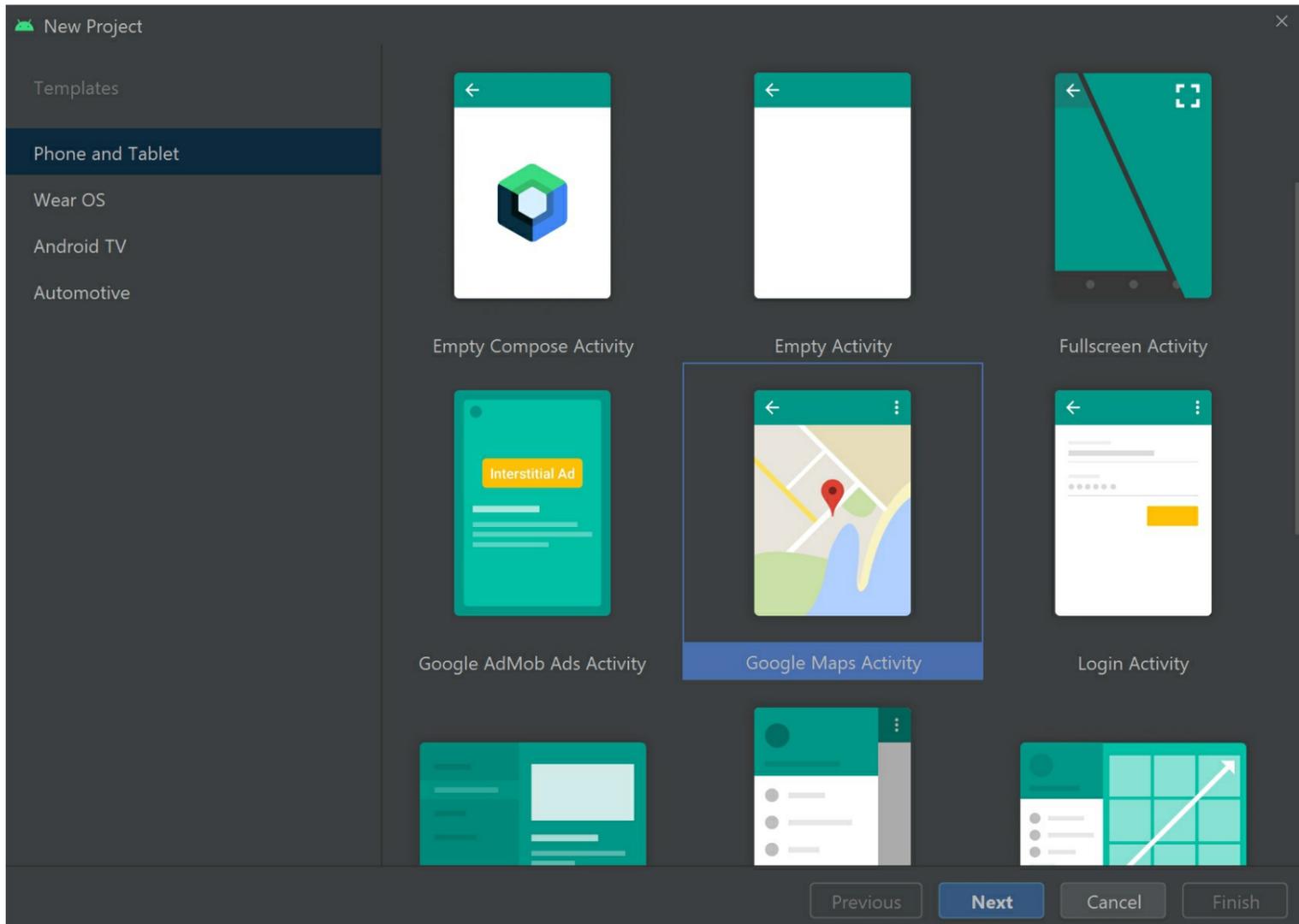
## Introduction

For this project, we will create an app that sends the user on a treasure hunt! Besides being a bit of fun, the treasure hunt app will be a great opportunity to build a maps-based application and learn about the features of the Google Maps API. Once the user initiates the treasure hunt, the app will select a secret nearby location and the user must find their way there. To help the user along their journey, the app will provide directional hints. The app will automatically detect when the user reaches the treasure's location. In building the app, you will learn how to incorporate the Google Maps API, monitor the user's location, and use geofencing to detect when the user reaches their target destination.

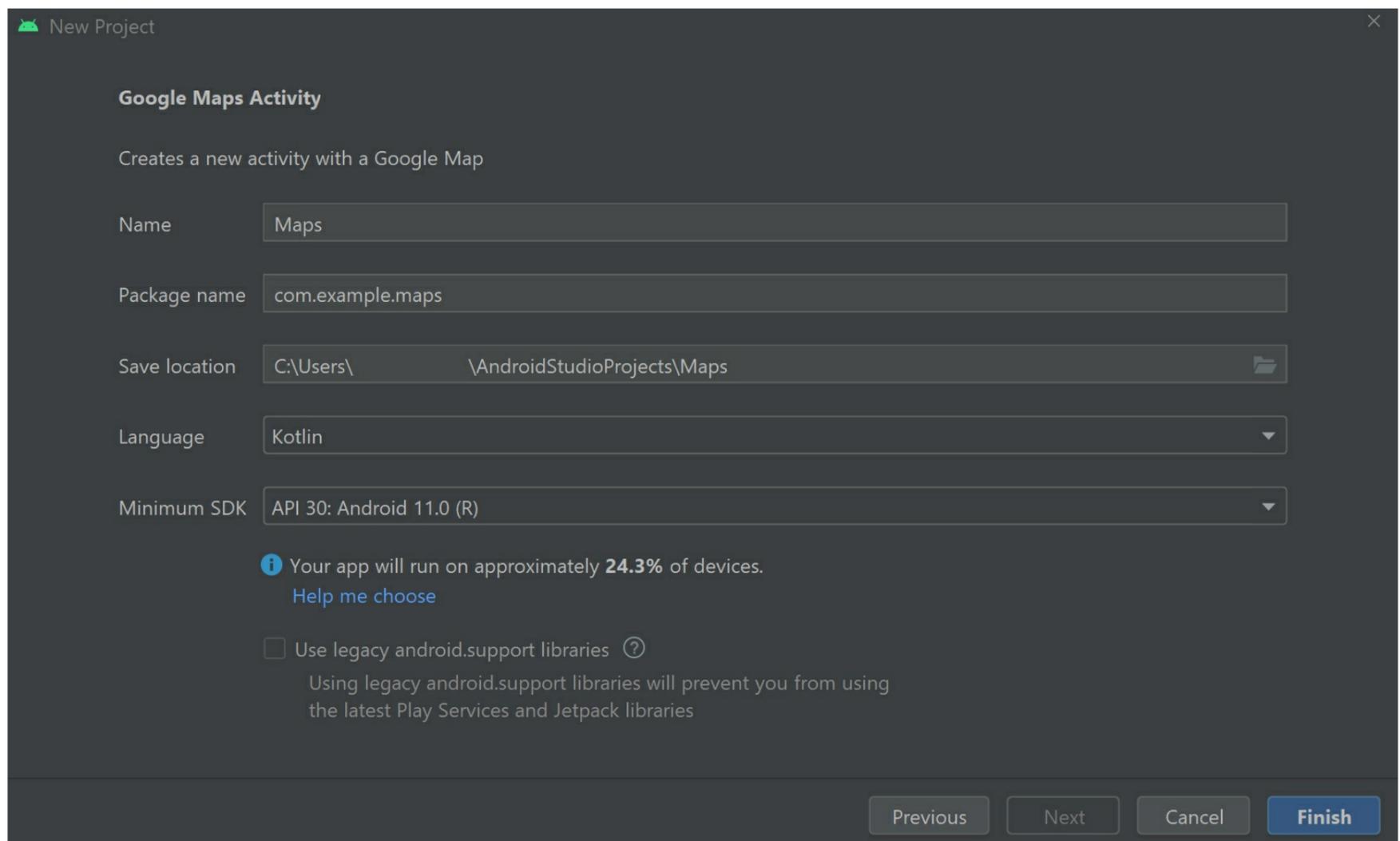


## Getting started

To begin, open Android Studio and create a new project using the Google Maps Activity project template. The Google Maps Activity template provides the app with a fragment ready to display the map and tools for communicating with the Google Maps API.



In the Create New Project window, add a name for the project (e.g. Maps), set the language to Kotlin and select API level 30.



It is recommended you enable Auto Imports to direct Android Studio to add any necessary import statements to your Kotlin files as you code. These import statements are essential for incorporating the external classes and tools required for the app to run. To enable Auto Imports, open Android Studio's Settings window by clicking **File > Settings**. In the Settings window, navigate through **Editor > General > Auto Import** then select 'Add unambiguous imports on the fly' and 'Optimise imports on the fly' for both Java and Kotlin then press

Apply and OK.

Android Studio should now add most of the necessary import statements to your Kotlin class files automatically. Sometimes there are multiple classes with the same name and the Auto Import feature will not work. In these instances, the requisite import statement(s) will be specified explicitly in the example code. You can also refer to the finished project code which accompanies this book to find the complete files including all import statements.

## Defining the string resources used in the app

Like the other projects covered in this book, the Maps app will store all the strings of text used throughout the application in a resource file. To define the string resources, navigate through **Project** > **app** > **res** and open the file called **strings.xml**. Once the file opens in the editor, modify its contents so it reads as follows:

```
<resources>
  <string name="app_name">Maps</string>
  <string name="title_activity_maps">Map</string>
  <string name="hint">Hint</string>
  <string name="compass">Compass</string>
  <string name="ok">OK</string>
  <string name="timer">Seconds remaining: %1$d</string>
  <string name="times_up">Time's up!</string>

  <!-- Treasure hunt -->
  <string name="begin_search">Begin your search for the treasure!</string>
  <string name="start_treasure_hunt">Start treasure hunt!</string>
  <string name="hunt_ended">Treasure hunt ended</string>
  <string name="end_the_treasure_hunt">End treasure hunt</string>
  <string name="treasure_found">You found the treasure!</string>

  <!-- Errors -->
  <string name="address_error">Error retrieving the address</string>
  <string name="no_address">No address found</string>
  <string name="geofence_error">A Geofence error has occurred: %1$s</string>
  <string name="location_error">There has been an error finding your current location</string>
  <string name="permission_required">This application requires access your device's location to organise the
treasure hunt.</string>
  <string name="treasure_error">Could not load treasure location. Error: %1$s</string>

  <!-- Directions -->
  <string name="north">north</string>
  <string name="east">east</string>
  <string name="south">south</string>
  <string name="west">west</string>
  <string name="direction">Head %1$s %2$s</string>
</resources>
```

You'll notice these string resources crop up throughout the project. They will help display messages to the user, populate TextView widgets, provide content descriptions for images and more!

## Setting up the Google Maps SDK

The Maps app will request navigational and location data from the Google Maps software development kit (SDK). Google monitors the use of its SDKs via its API Console, so you will need to register for a free API key and add the key to the project. To find where to enter the key, navigate through **Project** > **app** > **res** > **values** and open the file called **google\_maps\_api.xml**. In this file, Android Studio will likely have generated some instructions for how to create a Google Maps API key, including a custom link to register your app with Google. Locate the link that begins [https://console.developers.google.com/...](https://console.developers.google.com/) and copy and paste it into your web browser.

```
google_maps_api.xml x
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
7  https://console.developers.google.com/flows/enableapi?apiid=maps\_android\_backend&keyT
8
9  You can also add your credentials to an existing key, using these values:
10
11  Package name:
```

When you visit the link, you will be invited to log into (or create) your Google account and register the application in Google's API console.

The screenshot shows the Google APIs console registration page. At the top, there is a navigation bar with the Google APIs logo and a "Select a project" dropdown. The main heading is "Register your application for Maps SDK for Android in Google API Console". Below this, there is a brief description of the Google API Console. The "Select a project where your application will be registered" section includes a dropdown menu with "Create a project" selected. The "Terms of Service" section has two checkboxes, both of which are unchecked. The "Country of residence" section has a dropdown menu with "United Kingdom" selected. At the bottom, there is a section for "I would like to receive periodic emails on news, product updates and special offers from Google Cloud and Google Cloud Partners." with "Yes" and "No" radio buttons. A blue "Agree and continue" button is at the bottom.

Once you have registered the application, you should receive confirmation that the Maps SDK has been enabled and that you can generate an API key.

### The API is enabled

The project has been created and Maps SDK for Android has been enabled.

Next, you'll need to create an API key in order to call the API.

[Create API key](#)

Press the Create API key button to generate an API key that the app can use to request data from the Google Maps SDK. The API key should begin 'AIza...'.

## API Keys

<input type="checkbox"/>	Name	Creation date ↓	Restrictions	Key		
<input type="checkbox"/>	API key 1	Feb 28, 2021	Android apps	AIzaSyCRct...0hDk5V8_UA		

Copy and paste the full key in between the opening and closing string tags in the **google\_maps\_api.xml** file (i.e. replace YOUR\_KEY\_HERE with your API key).

```
<string name="google_maps_key" templateMergeStrategy="preserve"
translatable="false">YOUR_KEY_HERE</string>
```

And with that, your app is now registered with the Google API Console and can begin requesting data from the Maps SDK!

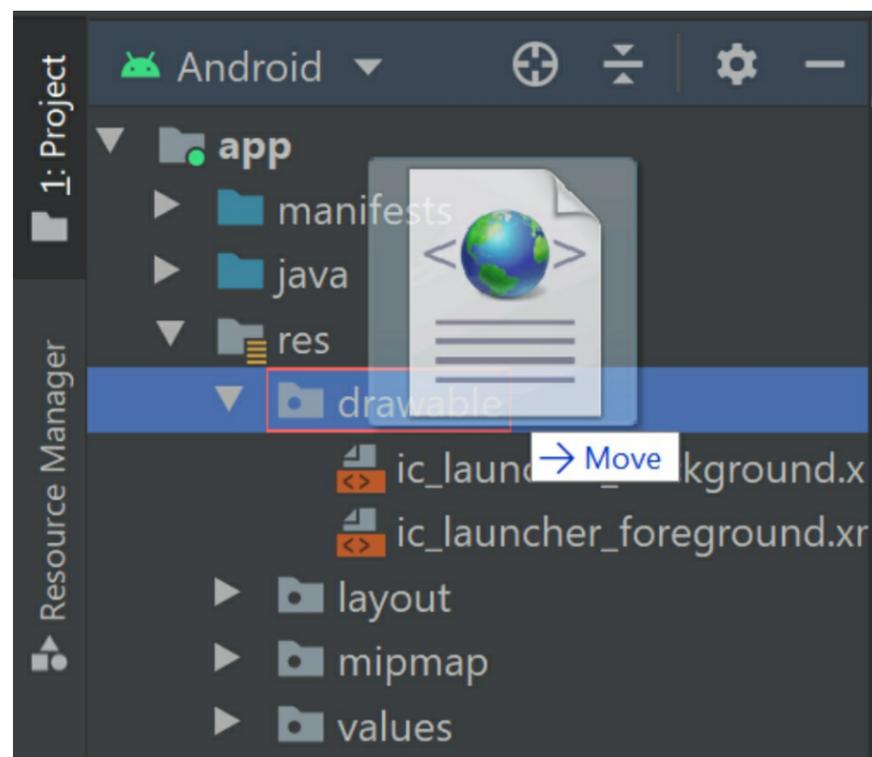
## Designing the activity\_maps layout

The Maps app will require only one layout. The layout will display an interactive map, several widgets to allow the user to start and end the treasure hunt and display hints, and an image of a compass that will rotate depending on the device's orientation. The compass image will be stored as a drawable resource. The easiest way to add the drawable to your project would be to copy the **compass.xml** file from the example code. To locate the file, navigate through the **app > src > main > res > drawable** folders in the example code for the Maps project.

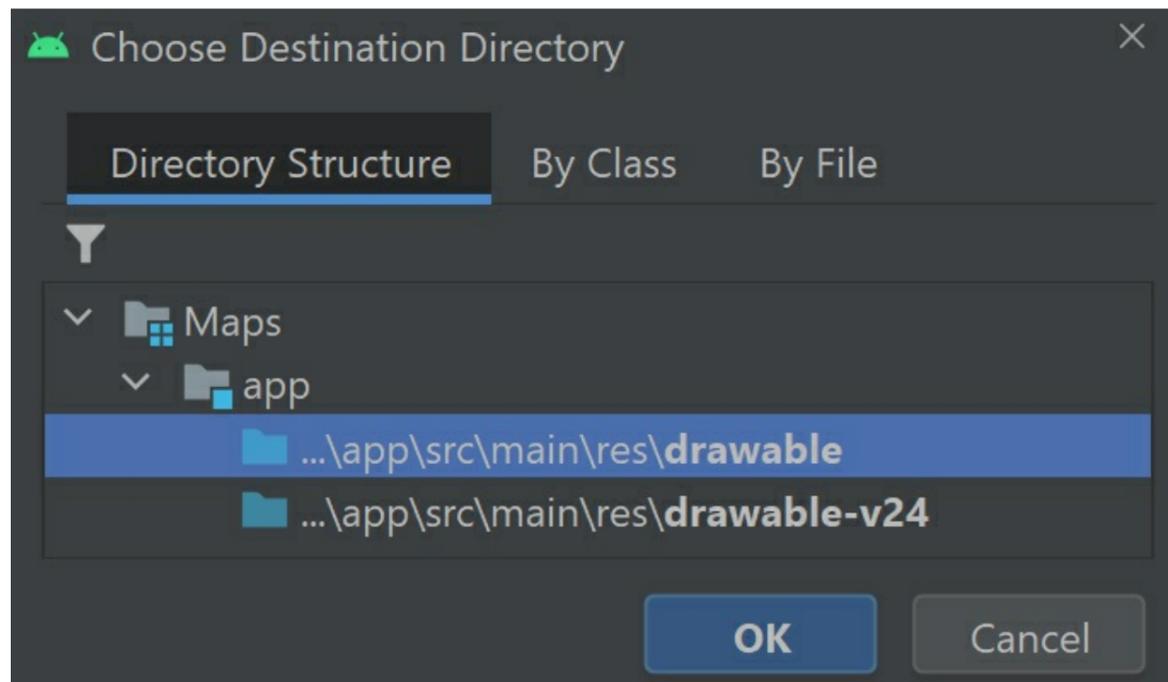
Maps > app > src > main > res > drawable

<input type="checkbox"/>	Name	Date modified	Type	Size
<input type="checkbox"/>	compass	30/09/2020 22:41	XML Document	106 KB
<input type="checkbox"/>	ic_launcher_background	28/07/2020 11:01	XML Document	6 KB

Locate the **compass.xml** file and drag and drop it into the **drawable** directory (**Project > app > res**) for your project in Android Studio.



If prompted, it is fine to select the regular **drawable** directory rather than **drawable-v24**.



Next, let's design the main app layout. Navigate through **Project > app > res > layout** and open the file called **activity\_maps.xml**. Switch the layout to Code view and edit the file so it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:map="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <androidx.fragment.app.FragmentContainerView
        android:id="@+id/map"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:name="com.google.android.gms.maps.SupportMapFragment"
        map:layout_constraintTop_toTopOf="parent"
        map:layout_constraintBottom_toTopOf="@id/controls" />

    <RelativeLayout
        android:id="@+id/controls"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingVertical="12dp"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toStartOf="@id/compass"
        app:layout_constraintBottom_toBottomOf="parent" >

        <Button
            android:id="@+id/treasureHuntButton"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginBottom="12dp"
            android:padding="4dp"
            android:text="@string/start_treasure_hunt"
            style="?attr/materialButtonOutlinedStyle" />

        <TextView
            android:id="@+id/timer"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignStart="@id/treasureHuntButton"
            android:layout_alignEnd="@id/treasureHuntButton"
            android:layout_below="@id/treasureHuntButton" />

        <Button
            android:id="@+id/hintButton"
```

```

        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="6dp"
        android:layout_marginEnd="6dp"
        android:layout_alignStart="@id/treasureHuntButton"
        android:layout_alignEnd="@id/treasureHuntButton"
        android:layout_below="@id/timer"
        android:text="@string/hint"
        android:visibility="invisible"
        style="?attr/materialButtonOutlinedStyle" />
</RelativeLayout>

<ImageView
    android:id="@+id/compass"
    android:layout_width="130dp"
    android:layout_height="130dp"
    android:layout_marginEnd="22dp"
    android:src="@drawable/compass"
    android:contentDescription="@string/compass"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@id/map"
    app:layout_constraintBottom_toBottomOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>

```

The contents of the `activity_maps` layout will be coordinated by a `ConstraintLayout` widget. Inside the `ConstraintLayout`, there is a fragment that will display an interactive map the user can navigate around. The height of the fragment is set to `0dp`, which means the height will automatically adjust to reflect its constraints. This is a useful technique for designing layouts that will accommodate varying screen sizes. In this instance, the fragment is constrained to the top of the parent `ConstraintLayout` and a `RelativeLayout` at the bottom of the screen, so the fragment will occupy as much space as possible after leaving room for the `RelativeLayout`.

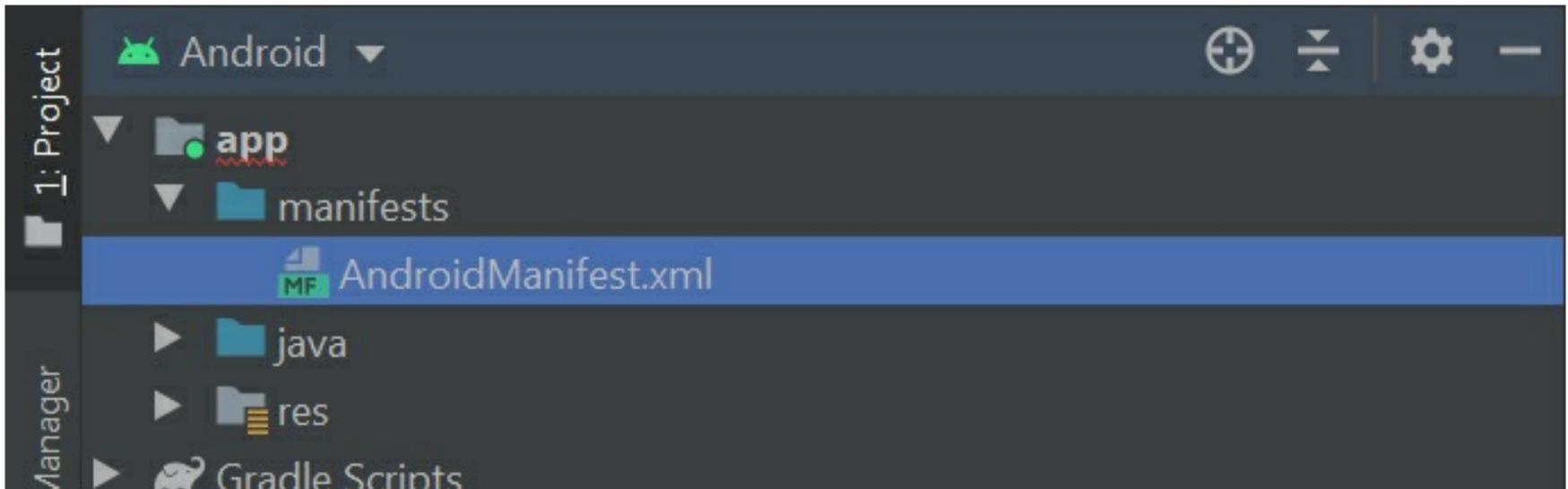
The `RelativeLayout` widget will contain three widgets: a button that will allow the user to start and end the treasure hunt, a timer that will display the remaining time for the treasure hunt, and a hint button that will advise the user which direction to travel in to find the treasure. By default, the hint button will be invisible because it will only appear when a treasure hunt is in progress. The buttons will incorporate a readymade outlined button style from Material Design (see <https://material.io/components/buttons/android#outlined-button>), which creates a thin border around the button for emphasis.

The final widget in the layout is an `ImageView` that is constrained to the right-hand side of the `RelativeLayout` at the bottom of the screen. The `ImageView` widget will display an image of a compass and rotate as the orientation of the device changes. Also, the `ImageView` contains a content description attribute that describes the image being displayed for the benefit of users who require a screen reader.

The main layout for the app is now in place. In the upcoming sections, we will write the code that coordinates the treasure hunt and interacts with the Google Maps SDK to find the user's location and monitor their progress towards the treasure.

## Requesting user permissions

The application will require permission from the user to access the device's location. All required permissions must be declared in the manifest file. The Google Play store will use the information in the manifest file to inform potential users about what permissions they will need to provide. To define the list of permissions, locate and open the **AndroidManifest.xml** file by navigating through **Project > app > manifests**.



Next, add the following code above the application element (Android Studio may have automatically added the ACCESS\_FINE\_LOCATION permission):

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_BACKGROUND_LOCATION" />
```

The above items define which permissions the app will require to operate. In this case, the app will need permission to access the user's precise location and continuously monitor their location in the background. The user's location will be monitored so the app can detect when the user reaches the location containing the treasure. New versions of Android will give the user the option to share their approximate location only; however, the app will work better if the user shares their precise location.

Let's now implement the code which prompts the user to grant the permissions. To handle all permission-related processes, we will create an object called LocationPermissionHelper in the MainActivity class. The LocationPermissionHelper object will contain methods for initiating and handling permissions-related requests. Locate and open the **MainActivity.kt** file by navigating through **Project > app > java > name of the project**. Next, add the following code below the onMapReady method to define the LocationPermissionHelper object:

```
object LocationPermissionHelper {
    private const val BACKGROUND_LOCATION_PERMISSION =
Manifest.permission.ACCESS_BACKGROUND_LOCATION
    private const val COARSE_LOCATION_PERMISSION = Manifest.permission.ACCESS_COARSE_LOCATION
    private const val FINE_LOCATION_PERMISSION = Manifest.permission.ACCESS_FINE_LOCATION

    fun hasLocationPermission(activity: Activity): Boolean {
        return ContextCompat.checkSelfPermission(activity, FINE_LOCATION_PERMISSION) ==
PackageManager.PERMISSION_GRANTED &&
ContextCompat.checkSelfPermission(activity, BACKGROUND_LOCATION_PERMISSION) ==
PackageManager.PERMISSION_GRANTED
    }

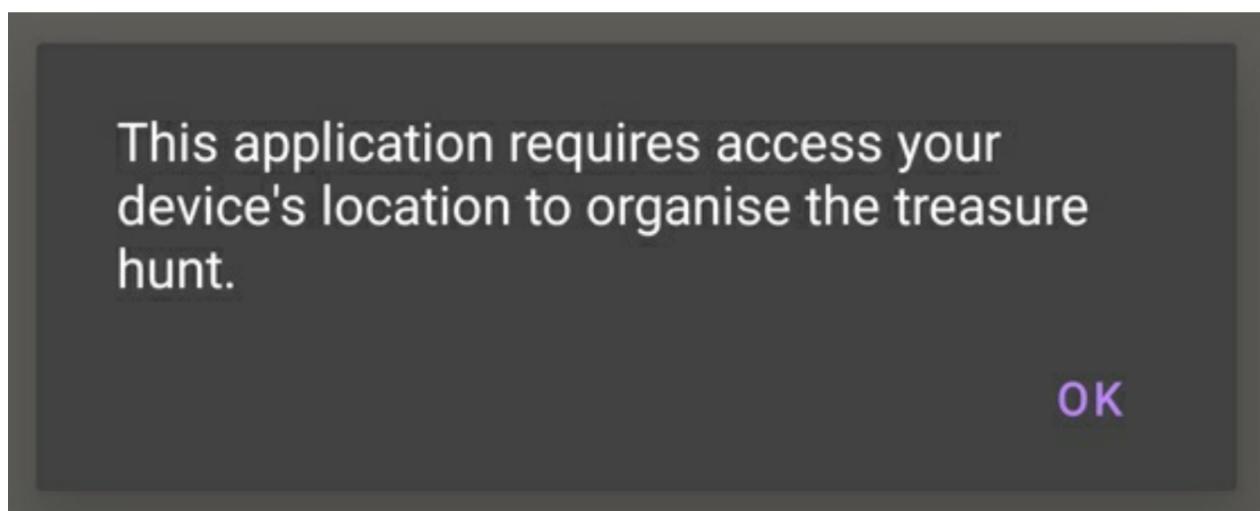
    fun requestPermissions(activity: Activity) {
        if (ActivityCompat.shouldShowRequestPermissionRationale(activity, FINE_LOCATION_PERMISSION)) {
            AlertDialog.Builder(activity).apply {
                setMessage(activity.getString(R.string.permission_required))
                setPositiveButton(activity.getString(R.string.ok)) { _, _ ->
                    ActivityCompat.requestPermissions(activity, arrayOf(FINE_LOCATION_PERMISSION,
COARSE_LOCATION_PERMISSION, BACKGROUND_LOCATION_PERMISSION), 0)
                }
            }.show()
        } else {
            ActivityCompat.requestPermissions(activity, arrayOf(FINE_LOCATION_PERMISSION,
COARSE_LOCATION_PERMISSION, BACKGROUND_LOCATION_PERMISSION), 0)
        }
    }
}
```

Note you may need to add the following import statements to the top of the file:

```
import android.app.AlertDialog
import android.Manifest
```

The first method in the `LocationPermissionHelper` object is called `hasLocationPermission` and will check whether the user has granted the app permission to access the device's fine location and continue monitoring the location in the background. The method will return a boolean (true/false value) indicating whether or not permission has been granted. Note the above code does not check if the coarse location permission is granted because the Geofence feature requires the fine location permission. If the user grants access to the coarse location but not the fine location, then the application will not work so we only check access to the fine location.

The next method is called `requestPermissions` and it is used to request the necessary location permissions. The user can either grant or refuse permission. If they refuse, the user might not understand why the permissions have been requested. In which case, we can use a method called `shouldShowRequestPermissionRationale` to check whether the user has refused a given permission. If the permission has been refused, then the `shouldShowRequestPermissionRationale` method returns a value of true. In which case, the above code builds an alert dialog that will display the message contained in the `permission_required` string resource. The alert dialog will also feature an OK button that will request the permissions again when clicked. In the above code, the rationale is only shown for the fine location permission; however, you are welcome to show the rationale for the background permission also.



Moving on, we'll now write the code which processes the user's response to the permissions request. Add the following code below the `onCreate` method:

```
override fun onRequestPermissionsResult(requestCode: Int, permissions: Array<out String>, grantResults: IntArray) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults)
    if (!LocationPermissionHelper.hasLocationPermission(this)) {
        LocationPermissionHelper.requestPermissions(this)
    } else prepareMap()
}
```

The `onRequestPermissionsResult` function handles the user's response to the permissions request. In the above code, the `LocationPermissionHelper` object's `hasLocationPermission` method is used to check whether all required permissions have been granted. If the `hasLocationPermission` method returns a value of false, then this means the user has refused to grant some or all of the required permissions. Consequently, the app will run the `LocationPermissionHelper` object's `requestPermissions` method again and show the rationale for the permission request. If the `hasLocationPermission` method returns a value of true, then a method called `prepareMap` will load the user's location and prepare the maps fragment found in the `activity_maps` layout.

The app should check whether the location permissions have been granted each time it launches. To handle this, add the following code to the bottom of the `onCreate` method:

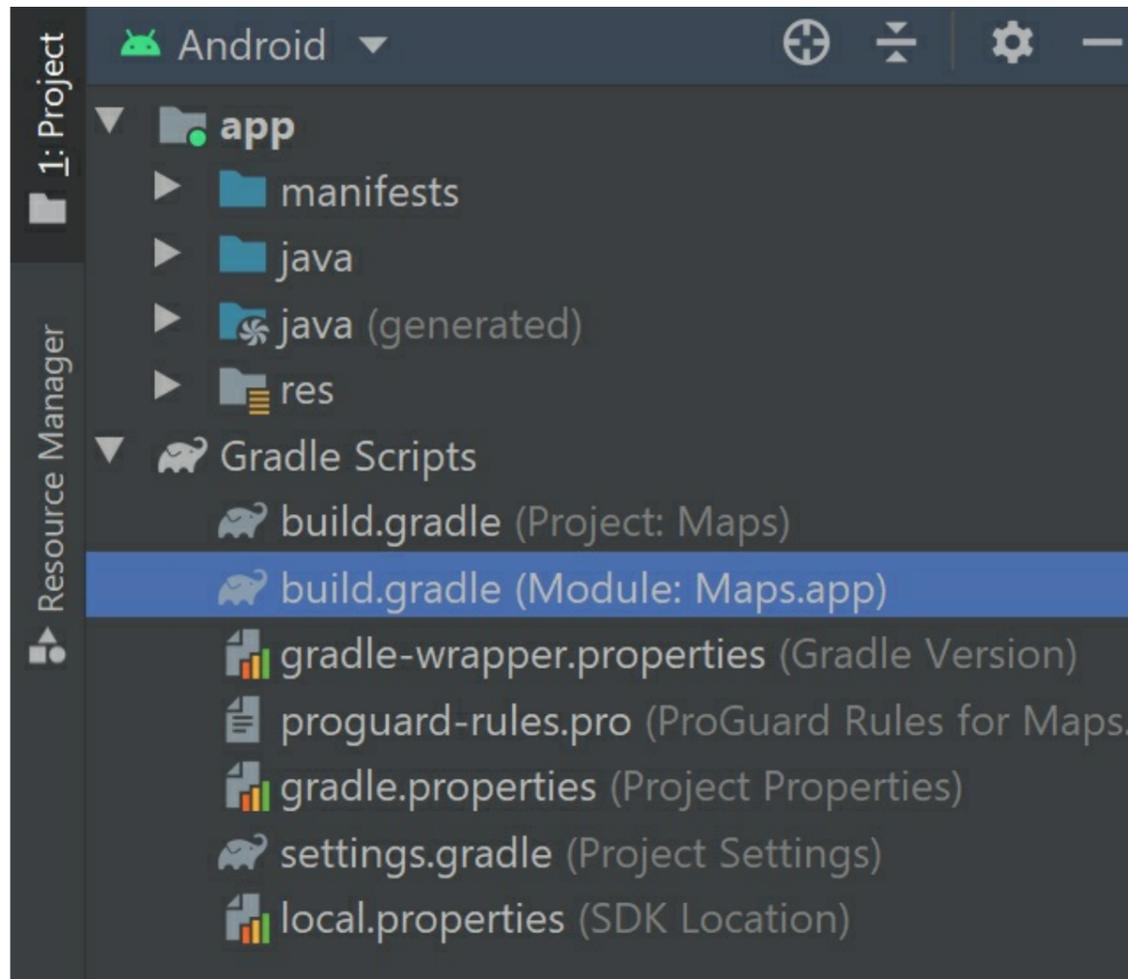
```
if (!LocationPermissionHelper.hasLocationPermission(this)) LocationPermissionHelper.requestPermissions(this)
```

Now, whenever the `MapsActivity` activity launches, it will use the `LocationPermissionHelper` object to check whether the required permissions have been granted and request them if necessary.

## Finding the user's location

The app will use Google Play's location service to find the user's current or last known location. To use the location service, we must import the necessary packages using a toolkit called Gradle. To import packages using Gradle,

navigate through **Project > Gradle Scripts** and open the Module-level **build.gradle** file:



Next, add the following implementation statement to the dependencies element to allow the app to access Google Play's location services:

```
implementation 'com.google.android.gms:play-services-location:19.0.1'
```

We're now finished with the Gradle Scripts files. Don't forget to re-sync your project when prompted!

Gradle files have changed since last project sync. A project sync may be necessary for the IDE ... [Sync Now](#)

Let's now return to the MapsActivity class and write the code that finds the user's location and prepares the map. First, add the following variables below the mMap variable at the top of the class:

```
private lateinit var fusedLocationClient: FusedLocationProviderClient
private lateinit var lastLocation: Location
```

The above code defines a variable that incorporates Google's Fused Location Provider API, which is a battery-efficient tool for finding the user's location, and a variable called lastLocation, which will store a Location object detailing the coordinates of the user's last known position. The Fused Location Provider API uses a variety of data sources to determine the user's location, including GPS and WiFi, and it does most of this processing behind the scenes, so it is easy to implement in our applications.

Both variables feature the lateinit modifier, which means we must assign them a value elsewhere in the code before they can be used. To initialise the fusedLocationClient variable, add the following code to the bottom of the onCreate method:

```
fusedLocationClient = LocationServices.getFusedLocationProviderClient(this)
```

The lastLocation variable will be initialised by a method called prepareMap, which will retrieve the user's last known location and apply it to the map. To define the prepareMap method, add the following code below the onMapReady method:

```
@SuppressWarnings("MissingPermission")
private fun prepareMap() {
    if (LocationPermissionHelper.hasLocationPermission(this)) {
        mMap.isMyLocationEnabled = true

        // Find the user's last known location
        fusedLocationClient.lastLocation.addOnSuccessListener { location: Location? ->
            location?.apply {
```

```

        lastLocation = location
        val currentLatLng = LatLng(location.latitude, location.longitude)
        mMap.animateCamera(CameraUpdateFactory.newLatLngZoom(currentLatLng, 12f))
    }
}
}
}
}

```

The `prepareMap` method first checks whether the app has permission to access the device's location. Providing permission has been granted, then the map's `isMyLocationEnabled` attribute is set to true. This means a blue dot will appear in the map fragment to indicate the user's current location. The `isMyLocationAttribute` also facilitates other features such as a button that allows the user to recenter the view to their current location and location tracking.

Next, the Fused Location Client requests the last known location of the user's device. Once the request has been processed, the `onSuccess` listener will return the location as a `Location` object. `Location` objects contain a variety of details about the location, including the longitude and latitude geographical coordinates. In rare cases, the location returned by the Fused Location Client is null. For this reason, the remainder of the code is wrapped in an `apply` block that will only run if the location is non-null.

First, the location is stored in the `lastLocation` variable so it can be referred to elsewhere in the activity. Next, the latitude and longitude coordinates are packaged in a `LatLng` object. The map's camera is positioned over the location specified in the `LatLng` object so the user can see their place on the map by a `GoogleMaps` class method called `animateCamera`. The `CameraUpdate` object that is supplied as an argument to the `animateCamera` method includes the `LatLng` object and a float value between 2.0 and 21.0, which indicates how close the map camera should zoom in.

The `prepareMap` method will run whenever the user launches the app and the map becomes available (providing the user has granted the necessary permissions). Once the map is available, a callback method called `onMapReady` will run. Android Studio should automatically have added the `onMapReady` method to the `MapsActivity` class when you selected the Google Maps Activity project template. The method likely contains some example code that plots a marker over Sydney on the map. Delete this code and modify the `onMapReady` method so it reads as follows:

```

override fun onMapReady(googleMap: GoogleMap) {
    mMap = googleMap

    mMap.uiSettings.isZoomControlsEnabled = true

    prepareMap()
}

```

Now, whenever the map fragment becomes available, the `onMapReady` method will run the `prepareMap` method and find the user's location. The above code also sets the `isZoomControlsEnabled` attribute of the map's user interface settings to true. This means zoom in and zoom out buttons will be added to the map so the user can control the camera view.

## Create a Geofence around the treasure location

In this section, we'll discuss how to apply a Geofence perimeter around a location on the map. The Geofence will alert the app whenever the user enters the perimeter and provides a useful mechanism for identifying when the user reaches their target destination. In this app, the Geofence will contain the treasure. To coordinate the Geofence, add the following companion object and variables to the top of the `MapsActivity` class:

```

companion object {
    const val MINIMUM_RECOMMENDED_RADIUS = 100F
    const val GEOFENCE_KEY = "TreasureLocation"
}

private val geofenceList = arrayListOf<Geofence>()
private var treasureLocation: LatLng? = null
private lateinit var geofencingClient: GeofencingClient

```

The companion object stores two constant values: the radius of the area covered by the Geofence (stored in `Float` format where 1F = 1 metre on the map), and a key that will identify the Geofence. Next, in the regular list of variables, the `geofenceList` variable will store an array list of all the active Geofence objects. Meanwhile, the

treasureLocation variable will store the LatLng object containing the coordinates of the treasure location, and the geofencingClient variable will hold an instance of the GeofencingClient class, which we can use to interact with the Geofencing APIs. To initialise the geofencingClient variable, add the following code to the bottom of the onCreate method:

```
geofencingClient = LocationServices.getGeofencingClient(this)
```

Moving on, let's define a method called generateTreasureLocation. The generateTreasureLocation method will create a Geofence around a random nearby location that will serve as the site of the treasure. To define the method, add the following code below the prepareMap method:

```
private fun generateTreasureLocation() {  
    val choiceList = listOf(true, false)  
    var choice = choiceList.random()  
    val treasureLat = if (choice) lastLocation.latitude + Random.nextFloat()  
    else lastLocation.latitude - Random.nextFloat()  
    choice = choiceList.random()  
    val treasureLong = if (choice) lastLocation.longitude + Random.nextFloat()  
    else lastLocation.longitude - Random.nextFloat()  
    treasureLocation = LatLng(treasureLat, treasureLong)  
}
```

Note you may need to add the following import statement to the top of the file:

```
import kotlin.random.Random
```

The above code will find a random nearby location. It does this by selecting a boolean value of true or false by random. If a value of true is randomly selected, then the method will add a number to the user's current latitude location coordinate. Meanwhile, if the value is false then the method will subtract a number from the coordinate. The number that will be added or subtracted will be a random Float number between 0 and 1. This process is then repeated for the longitude coordinate. For example, imagine the user's current location coordinates are Lat: 45.91, Lon: 8.08. To define the latitude coordinate of the treasure, the method will select a random boolean value and random Float number between 0 and 1 such as 0.24. If the boolean is true, then this means the Float will be added to the user's current latitude coordinate and the latitude coordinate of the treasure will be 46.15 (45.91 + 0.24). This process is repeated to determine the Longitude coordinate also. The advantage of this method of finding a random location is that the new location coordinates will be within 1 Float of the user's current location coordinates. This means the location of the treasure will be relatively near to the user.

Moving on, to create a Geofence around the treasure, add the following code to the generateTreasureLocation method:

```
removeTreasureMarker()  
geofenceList.add(Geofence.Builder()  
    .setRequestId(GEOFENCE_KEY)  
    .setCircularRegion(  
        treasureLat,  
        treasureLong,  
        MINIMUM_RECOMMENDED_RADIUS  
    )  
    .setExpirationDuration(Geofence.NEVER_EXPIRE)  
    .setTransitionTypes(Geofence.GEOFENCE_TRANSITION_ENTER)  
    .build()  
)
```

```
try {  
    geofencingClient.addGeofences(createGeofencingRequest(), createGeofencePendingIntent())  
        .addOnSuccessListener(this) {  
            Toast.makeText(this, getString(R.string.begin_search), Toast.LENGTH_SHORT).show()  
            // TODO: Start the timer and display an initial hint  
        }  
        .addOnFailureListener(this) { e ->  
            Toast.makeText(this, getString(R.string.treasure_error, e.message), Toast.LENGTH_SHORT).show()  
        }  
} catch (ignore: SecurityException) {}
```

The above code begins by running a method that we will define shortly called `removeTreasureMarker`, which will remove any markers on the map associated with previous treasure locations. Next, the `GeofenceBuilder` class is used to construct a Geofence around the location of the treasure. The Geofence builder assigns the Geofence an ID so we can reference it later if needed. The area covered by the Geofence is defined by passing the latitude and longitude coordinates of the treasure and our chosen radius (100F as defined in the companion object) as parameters to the `setCircularRegion` method. Next, the `setExpirationDuration` method allows you to define how long the Geofence will last (in milliseconds). If you would like the Geofence to never expire then the expiration duration should be set to `Geofence.NEVER_EXPIRE`. Finally, the `setTransitionTypes` command defines what type of activity you would like to receive alerts for. Acceptable activities include `GEOFENCE_TRANSITION_ENTER`, which will notify the app when the user enters the Geofence area; `GEOFENCE_TRANSITION_EXIT`, which will notify the app when the user leaves the Geofence area; and `GEOFENCE_TRANSITION_DWELL`, which will notify the app when the user remains in the Geofence area for a given amount of time. Once all the properties of the Geofence have been established, the above code builds an instance of the completed Geofence object and adds it to the `geofenceList` variable.

The Geofence is sent to the Geofencing client, which will attempt to add it to the map. If the Geofence is added successfully, then the `onSuccess` listener will display a toast notification informing the user that they can begin the treasure hunt. Meanwhile, if an error occurs, then the `onFailure` listener will display a notification advising that there was a problem plotting the location of the treasure. All the above code is wrapped in a try/catch block because the `GeofencingClient` class's `addGeofences` method will throw a security exception if the user has not granted the app permission to access the device's location. The app is designed such that the `generateTreasureLocation` method cannot run if the user has not granted permission, so we simply catch the exception as a formality to prevent code compilation errors rather than because we expect to have to deal with the exception at runtime.

## How to add a visible circle around the Geofence area

In the Maps app, the Geofence will not be visible to the user; however, if you wish to display a circle around the Geofence area then add the following code to the `addGeofence` `onSuccess` listener:

```
val circleOptions = CircleOptions()
    .strokeColor(Color.BLACK)
    .fillColor(Color.TRANSPARENT)
    .center(treasureLocation!!)
    .radius(MINIMUM_RECOMMENDED_RADIUS.toDouble())
mMap.addCircle(circleOptions)
```

The circle is designed by customising the properties of a `CircleOptions` object: the `strokeColor` attribute defines the colour of the circle's body; the `fillColor` attribute sets the colour inside the circle (transparent in this instance); the `center` attribute accepts the `LatLng` object that was used to define the location of the Geofence, and the `radius` attribute defines the radius of the circle. In this case, the radius is set to the value of the `MINIMUM_RECOMMENDED_RADIUS` variable from the companion object.

Once the `CircleOptions` object has been prepared, you can add it to the map by using the map fragment's `addCircle` command to create an effect similar to below:



## Detect Geofence alerts

In this section, we will write the code that adds a Geofence to the map and responds to alerts. First, add the following code below the generateTreasureLocation method. This code will create a GeofencingRequest object that can register Geofences with the Geofencing client:

```
private fun createGeofencingRequest(): GeofencingRequest {
    return GeofencingRequest.Builder().apply {
        setInitialTrigger(GeofencingRequest.INITIAL_TRIGGER_ENTER)
        addGeofences(geofenceList)
    }.build()
}
```

The createGeofencingRequest method sets an initial trigger, which defines what Geofence events should be reported the moment a Geofence is added to the client. In this case, the initial trigger is set to INITIAL\_TRIGGER\_ENTER which means a notification will appear if the user has already arrived at the Geofence area. If you wished to disable these initial notifications then you could set the InitialTrigger to 0:

```
setInitialTrigger(0)
```

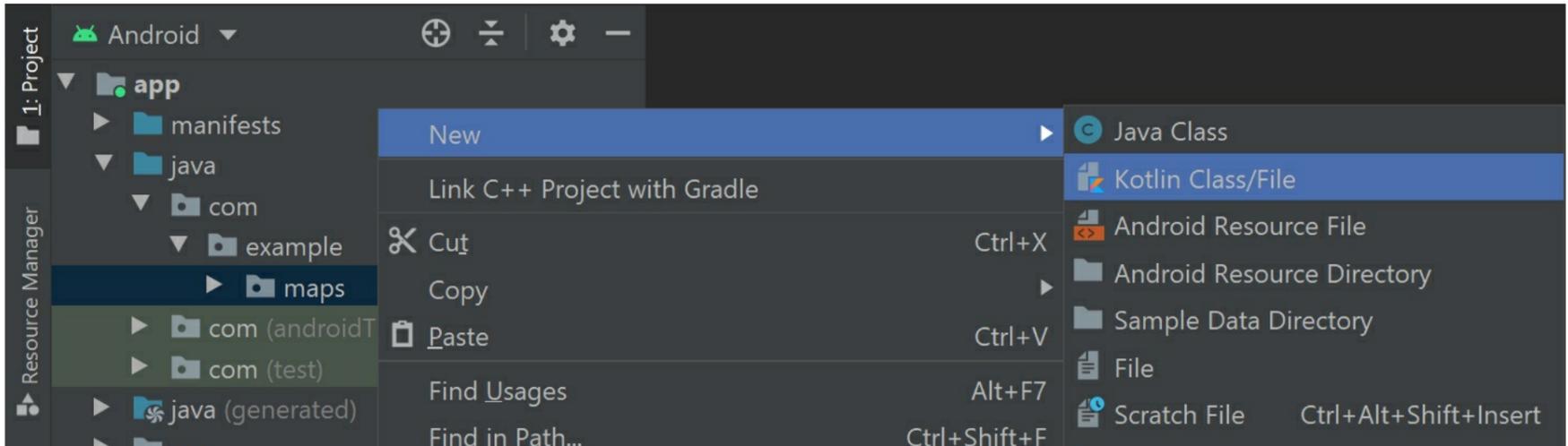
Next, the list of Geofences that will be monitored is added to the GeofencingRequest instance. The GeofencingRequest instance is then returned to the generateTreasureLocation method and submitted to the Geofencing client.

Moving on, add the following code below the createGeofencingRequest method to register a broadcast receiver that will monitor and respond to Geofence alerts:

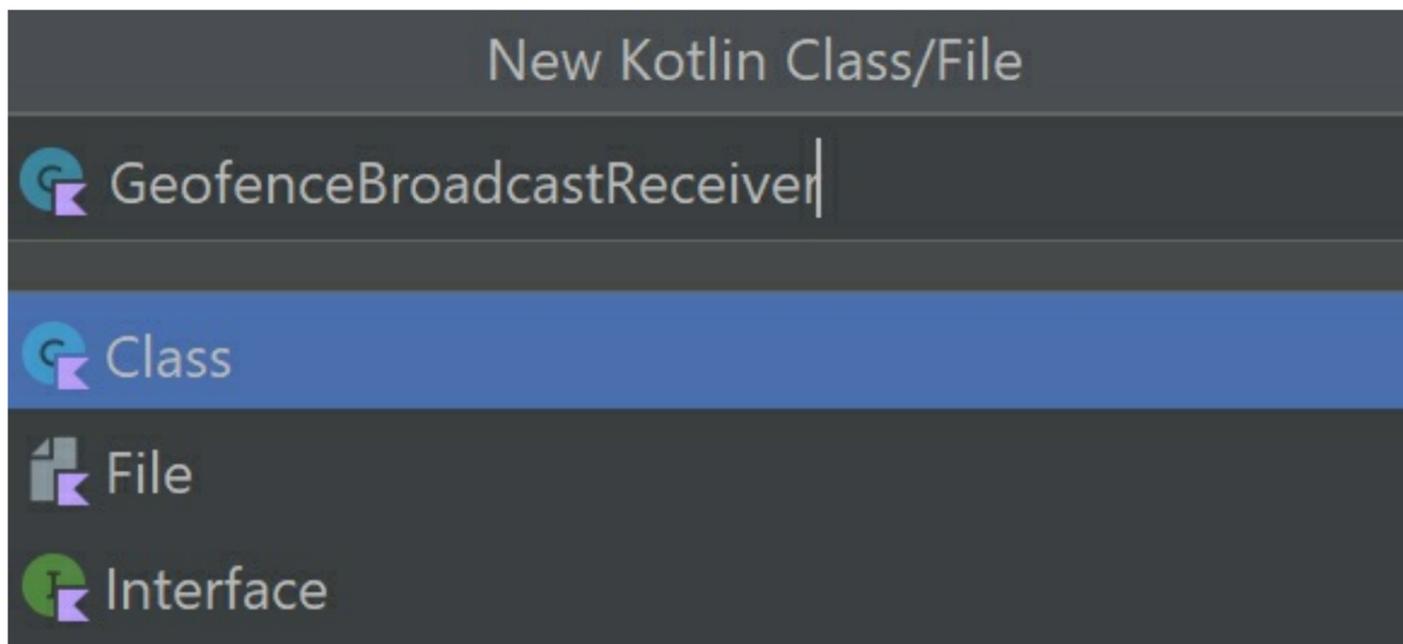
```
private fun createGeofencePendingIntent(): PendingIntent {
    val intent = Intent(this, GeofenceBroadcastReceiver::class.java)
    return PendingIntent.getBroadcast(this, 0, intent, PendingIntent.FLAG_IMMUTABLE or
        PendingIntent.FLAG_UPDATE_CURRENT)
}
```

The `createGeofencePendingIntent` method generates a `PendingIntent` object. Typically, pending intents are used to direct another application to perform a task. The intent is considered pending because there is no guarantee the other application will act immediately. In this case, the pending intent will signal to a broadcast receiver class within our app called `GeofenceBroadcastReceiver`. A pending intent will launch whenever the `GEOFENCE_TRANSITION_ENTER` alert occurs, and the `GeofenceBroadcastReceiver` class will handle the alert.

Let's now create the `GeofenceBroadcastReceiver` class that will process Geofence alerts. Right-click the directory that contains `MapsActivity` (**Project > app > java > name of the project**) and select **New > Kotlin Class/File**.



Name the file `GeofenceBroadcastReceiver` and select Class from the list of options.



Once the `GeofenceBroadcastReceiver.kt` file opens in the editor, modify its code so it reads as follows:

```
class GeofenceBroadcastReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context?, intent: Intent) {
        val geofencingEvent = GeofencingEvent.fromIntent(intent)
        if (geofencingEvent.hasError()) {
            Toast.makeText(context, context?.getString(R.string.geofence_error,
                geofencingEvent.errorCode.toString()), Toast.LENGTH_SHORT).show()
            return
        }

        // Get the transition type.
        val geofenceTransition = geofencingEvent.geofenceTransition
        if (geofenceTransition == Geofence.GEOFENCE_TRANSITION_ENTER)
            context?.sendBroadcast(Intent("GEOFENCE_ENTERED"))
    }
}
```

The `onReceive` method of the `GeofenceBroadcastReceiver` will run whenever an intent is sent by the Geofencing client. The `onReceive` method will check whether the intent contains a Geofence monitoring error. If an error has occurred, then the error code will be displayed using a toast notification. On the other hand, if no error has occurred, then the transition type that triggered the alert is assessed. If the transition type is `GEOFENCE_TRANSITION_ENTER`, then this means the user has entered the Geofence area. The broadcast receiver will respond to this event by sending a broadcast intent with a reference tag of "GEOFENCE\_ENTERED"

to the context in which the broadcast receiver is running. In this app, the broadcast receiver context will be tied to the MapsActivity activity. In the next section, we will configure MapsActivity class to respond to the broadcast intent and congratulate the user on entering the Geofence area and finding the treasure.

To receive alerts, we must also register the broadcast receiver as a service in the application's manifest file. Locate and open the **AndroidManifest.xml** file by navigating through **Project > app > manifests**. Next, add the following line of code inside the application element to register the GeofenceBroadcastReceiver class as a broadcast receiver that belongs to the app:

```
<receiver android:name=".GeofenceBroadcastReceiver" />
```

## Respond to Geofence area enter transitions

So far, we have written the code that creates a Geofence around a randomly selected nearby location and registers a broadcast receiver to detect whenever the user enters the Geofence area. Upon receiving an alert that the user has entered the Geofence area, the broadcast receiver will emit a broadcast intent that the MapsActivity activity will intercept and use to congratulate the user on finding the treasure. To handle this, return to the MapsActivity class and add the following code to the list of variables at the top of the class:

```
private val broadcastReceiver = object : BroadcastReceiver() {  
    override fun onReceive(context: Context?, intent: Intent?) {  
        endTreasureHunt()  
        Toast.makeText(this@MapsActivity, getString(R.string.treasure_found), Toast.LENGTH_LONG).show()  
    }  
}
```

The above variable defines a BroadcastReceiver object that will respond to broadcast intents. If an intent is received, the onReceive callback method will run a method called endTreasureHunt that will end the various processes relating to the treasure hunt and display a toast notification congratulating the user on finding the treasure. To register the BroadcastReceiver instance and direct it to intercept broadcast intents with a reference tag of "GEOFENCE\_ENTERED", add the following code to the bottom of MapsActivity class's onCreate method:

```
registerReceiver(broadcastReceiver, IntentFilter("GEOFENCE_ENTERED"))
```

The MapsActivity class must also unregister the broadcast receiver when it shuts down to prevent memory leaks and any unnecessary drains on the app resources. To unregister the broadcast receiver when the activity is being destroyed, add the following code below the onCreate method:

```
override fun onDestroy() {  
    super.onDestroy()  
    unregisterReceiver(broadcastReceiver)  
}
```

Once the treasure hunt has ended, the app will reveal the location of the treasure by placing a marker on the map, stopping the countdown timer and updating the user interface so the user can start a new treasure hunt if they wish. To facilitate these processes, add the following variables to the list of variables at the top of the class:

```
private var treasureMarker: Marker? = null  
private var huntStarted = false
```

The treasureMarker variable will store a reference to a Marker object that is placed on the map to indicate the location of the treasure. Meanwhile, the huntStarted variable will store a boolean value indicating whether or not a treasure hunt is underway. Once those two variables are in place, write the following code below the createGeofencePendingIntent method to define a method called endTreasureHunt:

```
private fun endTreasureHunt() {  
    geofencingClient.removeGeofences(createGeofencePendingIntent()).run {  
        addSuccessListener {  
            geofenceList.clear()  
        }  
        addOnFailureListener { }  
    }  
    if (treasureMarker == null) treasureMarker = placeMarkerOnMap(treasureLocation!!)  
    binding.treasureHuntButton.text = getString(R.string.start_treasure_hunt)  
    binding.hintButton.visibility = View.INVISIBLE
```

```

    huntStarted = false
    // TODO: Cancel the timer here
    binding.timer.text = getString(R.string.hunt_ended)
}

```

The `endTreasureHunt` method uses the `GeofencingClient` class's `removeGeofences` method to remove all the Geofences that are associated with the `PendingIntent` we used earlier to register the `GeofencesBroadcastReceiver`. In this way, the method can deactivate the Geofence that is monitoring the treasure location. Once this operation is complete, the `geofenceList` variable is cleared because its contents are no longer valid. Next, a method called `placeMarkerOnMap` (that we'll define in the next section) uses the `LatLng` object stored in the `treasureLocation` variable to plot a marker on the map and reveal the location of the treasure. Finally, the user interface is updated so that the text in the `treasureHuntButton` widget reads 'Start treasure hunt!', the button that allows the user to generate hints for the treasure's location disappears, and the text displayed in the timer `TextView` is set to 'Treasure hunt ended'. Also, the value of the `huntStarted` variable is set to `false` to indicate that no treasure hunt is currently underway.

The marker placed on the map to indicate the location of the treasure should be removed if the user decides to start a new treasure hunt. The process of removing markers from the map will be handled by a method called `removeTreasureMarker`. To define the `removeTreasureMarker` method, add the following code below the `endTreasureHunt` method to remove the `Marker` stored in the `treasureMarker` variable ready for a new treasure hunt to begin:

```

private fun removeTreasureMarker() {
    if (treasureMarker != null) {
        treasureMarker?.remove()
        treasureMarker = null
    }
}

```

## Plotting markers on the map

Once the treasure hunt ends it would be useful to reveal the location of the treasure by plotting a marker on the map. This functionality will be handled by a method called `placeMarkerOnMap` that will plot a marker at a location specified within a `LatLng` object. To define the `placeMarkerOnMap` method, add the following code below the `removeTreasureMarker` method:

```

private fun placeMarkerOnMap(location: LatLng): Marker? {
    val markerOptions = MarkerOptions()
        .position(location)
        .title(getAddress(location))

    return mMap.addMarker(markerOptions)
}

```

The `placeMarkerOnMap` method generates a marker for the position defined in a given `LatLng` object. It then assigns a title to the marker which will be shown if the user clicks the marker. In this case, the title will display the address associated with the marker. Finally, the marker is added to the map. A reference to the marker is returned by the `placeMarkerOnMap` method so the marker can be located and removed if necessary.



The street address that will be used as the marker title will be retrieved by a method called `getAddress`. To define the `getAddress` method, add the following code below the `placeMarkerOnMap` method:

```
private fun getAddress(latLng: LatLng): String {
    var addressText = getString(R.string.no_address)

    try {
        val addresses = Geocoder(this).getFromLocation(latLng.latitude, latLng.longitude, 1)
        if (!addresses.isNullOrEmpty()) {
            addressText = addresses[0].getAddressLine(0) ?: addressText
        }
    } catch (e: IOException) {
        addressText = getString(R.string.address_error)
    }

    return addressText
}
```

The `getAddress` method uses the `Geocoder` class's `getFromLocation` method to find the list of addresses registered at the latitude and longitude coordinates stored in the `LatLng` object. The above code also sets the `getFromLocation` method's `maxResults` parameter to 1 because we are only interested in displaying the first registered address. Next, if an address is found, then the address data is transferred to a variable called `addressText` using the `Address` class's `getAddressLine` method. There is a chance that Google may not provide any address information. In which case, the `getAddressLine` method will return a value of null.

To handle null address data, the above code uses the Elvis operator `?:`. If the value on the left-hand side of the Elvis operator is null, then the value on the right-hand side of the operator will be used instead. In this way, the Elvis

operator allows you to provide a backup value if your first choice is null. On this occasion, if the address information is null, then we instruct the `addressText` variable to retain its original value. If you refer to the beginning of the method you will see we originally assign the `addressText` variable a string that informs the user that no address was found; however, we attempt to replace this value with actual address data if available.

All of the above code is wrapped in a try/catch block to intercept any input/output exceptions (IOExceptions) that may occur. For instance, network connectivity issues could prevent the retrieval of address data. In which case, the value of the `addressText` variable is set to “Error retrieving the address”.

## Initiating the treasure hunt

In this section, we will configure the user interface widgets that will allow the user to start the treasure hunt, view how much time is remaining and generate hints for which direction to travel to find the treasure. The user can start and end the treasure hunt by pressing the Button widget with the ID `treasureHuntButton` from the `activity_maps` layout. To enable this functionality, add the following code to the bottom of `MapsActivity` class’s `onCreate` method:

```
binding.treasureHuntButton.setOnClickListener {
    when {
        !this::lastLocation.isInitialized -> Toast.makeText(this, getString(R.string.location_error),
        Toast.LENGTH_LONG).show()
        huntStarted -> endTreasureHunt()
        else -> {
            generateTreasureLocation()
            binding.treasureHuntButton.text = getString(R.string.end_the_treasure_hunt)
            binding.hintButton.visibility = View.VISIBLE
            huntStarted = true
        }
    }
}
```

The above code attaches an `onClick` listener to the treasure hunt button and uses a `when` block to decide the appropriate course of action when the button is clicked. In the first scenario, the `lastLocation` variable has not been initialised, which means there has been a problem finding the user’s location and the treasure hunt cannot begin. In which case, a toast notification advises the user why the treasure hunt cannot start. If the `lastLocation` variable has been set, then the `when` block checks whether a treasure hunt is in progress. If a treasure hunt is underway, then the `endTreasureHunt` method will stop the treasure hunt and place a marker on the map to indicate where the treasure had been located. Finally, if none of the previous conditions applies, then this means a treasure hunt is not in progress. In which case, the `generateTreasureLocation` method will create a Geofence around the treasure location, the hint button will become visible so the user can request directional hints, and the `huntStarted` variable is set to `true`.

A timer will advise the user how much time remains for them to complete the treasure hunt. To incorporate the timer, add the following variable to the list of variables at the top of the `MapsActivity` class:

```
// 3600000 ms equals one hour
private val timer = object : CountdownTimer(3600000, 1000) {
    override fun onTick(millisUntilFinished: Long) {
        binding.timer.text = getString(R.string.timer, millisUntilFinished / 1000)
    }

    override fun onFinish() {
        endTreasureHunt()
        binding.timer.text = getString(R.string.times_up)
    }
}
```

The timer variable defines a `CountDownTimer` object that will count down for one hour (3600000 milliseconds). The ticker interval is set to one second (1000 milliseconds). At each interval, the `CountDownTimer` object’s `onTick` method will update the text displayed in the timer `TextView` widget to reflect the remaining time. Once the time has elapsed, the `onFinish` method will stop the treasure hunt using the `endTreasureHunt` method and change the text in the timer `TextView` widget to “Time’s up!”.

While a treasure hunt is underway, the user can request directional hints (e.g. “Head north west”) by pressing the hint button. To generate the hints, add the following code below the `getAddress` method:

```
private fun showHint() {
    if (treasureLocation != null && this::lastLocation.isInitialized) {
        val latDir = if (treasureLocation!!.latitude > lastLocation.latitude) getString(R.string.north)
        else getString(R.string.south)
        val lonDir = if (treasureLocation!!.longitude > lastLocation.longitude) getString(R.string.east)
        else getString(R.string.west)
        Toast.makeText(this, getString(R.string.direction, latDir, lonDir), Toast.LENGTH_SHORT).show()
    }
}
```

The showHint method will generate the hint text by calculating whether the treasure's latitude and longitude coordinates are greater than the coordinates of the user's current location. If the treasure's latitude and longitude coordinates are greater than the user's current location coordinates then they must head north and east, respectively. The bidirectional instructions are then packaged in the direction string resource and displayed in a toast notification (e.g. "Head south east").

To run the showHint method whenever the hint button is pressed, add the following code to the bottom of the onCreate method:

```
binding.hintButton.setOnClickListener {
    showHint()
}
```

When the treasure hunt begins, the app will start the timer and display an initial directional hint to the user. To arrange this, locate the generateTreasureLocation method and replace the TODO comment with the following:

```
timer.start()
showHint()
```

Likewise, to cancel the timer if the user decides to end the treasure hunt early, locate the endTreasureHunt method and replace the TODO comment with the following:

```
timer.cancel()
```

## Monitoring the device's location

During the treasure hunt, the app will continuously monitor the user's location to assess their progress towards the treasure. To facilitate this, add the following variables to the list of variables at the top of the MapsActivity class:

```
private var receivingLocationUpdates = false
private lateinit var locationCallback: LocationCallback
private lateinit var locationRequest: LocationRequest
```

In the above list of variables, the receivingLocationUpdates variable will store a boolean value indicating whether or not the app is actively receiving location updates, the locationCallback variable will handle changes in the device's location, and the locationRequest variable will define the frequency and accuracy of location updates.

If the device's settings do not allow the app to continuously monitor the user's location then the app will need to request authorisation from the user. To handle this, add the following code below the showHint method. The below code defines a method called createLocationRequest that will request regular location updates from Google's Location Services:

```
private fun createLocationRequest() {
    locationRequest = LocationRequest.create().apply {
        interval = 10000
        fastestInterval = 5000
        priority = LocationRequest.PRIORITY_HIGH_ACCURACY
    }

    val locationSettingsRequest = LocationSettingsRequest.Builder()
        .addLocationRequest(locationRequest)
        .build()

    val client = LocationServices.getSettingsClient(this)
    client.checkLocationSettings(locationSettingsRequest).apply {
        addSuccessListener {
```



```
super.onResume()
if (!receivingLocationUpdates) createLocationRequest()
}
```

The onResume method refers to a stage of the Android application lifecycle that occurs when the app is launched, or the user returns to the app after leaving it running in the background. If the user closes the app or leaves the app running in the background while doing something else on their device, location updates will no longer be required until the user returns. For this reason, it is best to use the onPause stage of the Android application lifecycle to stop the location updates. To arrange this, add the following code below the onResume method:

```
override fun onPause() {
    super.onPause()
    if (this::locationCallback.isInitialized) fusedLocationClient.removeLocationUpdates(locationCallback)
}
```

Now, whenever the app is closed (even temporarily) it will stop requesting location updates and prevent the device's resource and battery life from being used unnecessarily. If the user later returns to the app, then the location updates will start again as defined in the onResume method.

## Monitoring the device's orientation

All the code required for the app to initiate the treasure hunt and monitor the user's progress is now in place. For the remainder of this project, we will set up the compass image that appears in the bottom right corner of the activity\_maps layout. The compass will rotate according to the device's orientation and enable the user to determine which direction they are facing. This will help them respond to the hints provided while a treasure hunt is underway (e.g. "Head north west"). It will also be a useful opportunity to explore and interact with the sensors on an Android device.

To begin, add the following variables to the list of variables at the top of the MapsActivity class:

```
private val accelerometerReading = FloatArray(3)
private val magnetometerReading = FloatArray(3)
private val rotationMatrix = FloatArray(9)
private val orientationAngles = FloatArray(3)
private var isRotating = false
private lateinit var sensorManager: SensorManager
```

In the above code, the first four variables define Float arrays that will store data about different sensors on the device. The isRotating variable will store a boolean value indicating whether or not the compass is currently rotating, and the sensorManager variable will provide access to the sensors on the device via the system's sensor manager. Altogether, the sensors will provide the app with the necessary information to calculate the orientation of the device and rotate the compass image accordingly.

To initialise the sensorManager variable, add the following line of code to the bottom of the onCreate method:

```
sensorManager = getSystemService(SENSOR_SERVICE) as SensorManager
```

With regards to sensors, it is worth mentioning not all devices contain an accelerometer, which is the sensor that detects the acceleration of the device in a given direction (including the effects of gravity). If you wanted to ensure that the Google Play store will only display the app to suitable devices that include an accelerometer, you could add the following line to the **AndroidManifest.xml** file (**Project > app > manifest**) below the uses-permission statements:

```
<uses-feature android:name="android.hardware.sensor.accelerometer"
    android:required="true" />
```

Returning to the MapsActivity class, we need to configure the class to extend the SensorEventListener interface. In extending an interface, the base class can inherit that interface's data including its variables and methods. In other words, the MapsActivity class will inherit the SensorEventListener interface's ability to monitor and respond to the device's sensors. To implement this, locate the MapsActivity class declaration line and edit it so it reads as follows:

```
class MapsActivity : AppCompatActivity(), OnMapReadyCallback, SensorEventListener {
```

Inheriting from the SensorEventListener interface requires several mandatory methods to be implemented. An easy way to address this is to hover over the MapsActivity class declaration (that is likely highlighted in red) and click Implement members.

```

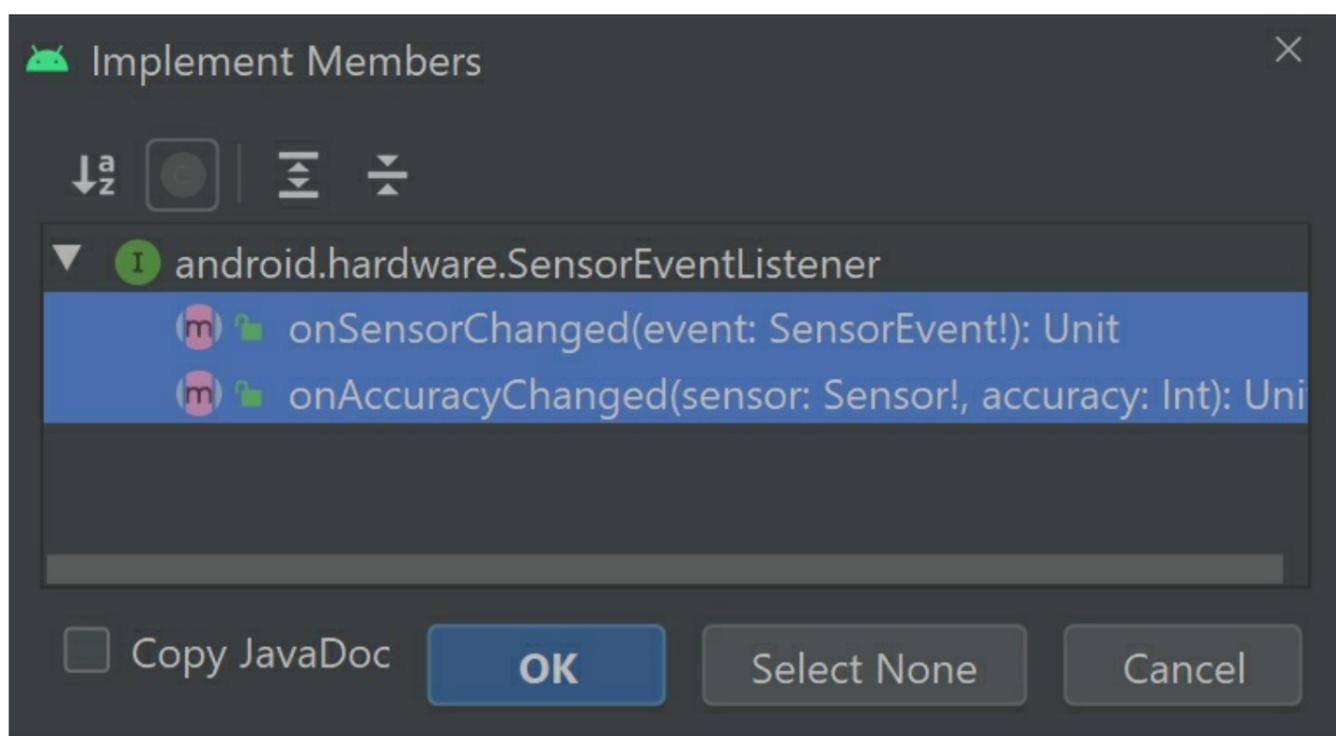
37 class MapsActivity : AppCompatActivity(), OnMapReadyCallback, SensorEventListener {
38
39
40
41
42
43
44

```

Class 'MapsActivity' is not abstract and does not implement abstract member **public abstract fun** onSensorChanged(event: SensorEvent!): Unit defined in android.hardware.SensorEventListener

[Implement members](#) Alt+Shift+Enter [More actions...](#) Alt+Enter

In the Implement Members window that opens, select both onSensorChanged and onAccuracyChanged then press OK.



Both methods should then appear at the bottom of the MapsActivity class. The onAccuracyChanged method detects when the accuracy of the sensors changes. This method will not be used in this application; however, we still have to include it because it is a mandatory method of the SensorEventListener interface. Meanwhile, the onSensorChanged method detects whenever a sensor emits a new set of values. This app will use the onSensorChanged method to monitor the accelerometer and magnetic field sensors. To do this, modify the onSensorChanged method so it reads as follows:

```

override fun onSensorChanged(event: SensorEvent?) {
    if (event == null) return

    when (event.sensor.type) {
        Sensor.TYPE_ACCELEROMETER -> System.arraycopy(event.values, 0, accelerometerReading, 0, accelerometerReading.size)
        Sensor.TYPE_MAGNETIC_FIELD -> System.arraycopy(event.values, 0, magnetometerReading, 0, magnetometerReading.size)
    }

    if (!isRotating) updateOrientationAngles()
}

```

The above code uses a when block to determine how the onSensorChanged method should respond when a given sensor reports new data. Specifically, the when block responds to the accelerometer sensor, which monitors the device's acceleration force, and the magnetic field sensor, which detects the geometric field acting upon the device. The onSensorChanged method copies the data output from the sensors into the relevant array variable (accelerometerReading or magnetometerReading). Once this is done, a method called updateOrientationAngles will process the data and rotate the compass image. The updateOrientationAngles method will only run if the isRotating variable is set to false because the app should not submit a new compass rotation request if a previous request has not finished. Otherwise, the compass may jolt from one direction to another. Each compass rotation request will take 500 milliseconds to complete, as specified in the updateOrientationAngles method.

To define the updateOrientationAngles method, add the following code below the startLocationUpdates method:

```

private fun updateOrientationAngles() {

```

```

    SensorManager.getRotationMatrix(rotationMatrix, null, accelerometerReading, magnetometerReading)
    SensorManager.getOrientation(rotationMatrix, orientationAngles)
    val degrees = (Math.toDegrees(orientationAngles[0]).toDouble())

    val newRotation = degrees.toFloat() * -1
    val rotationChange = newRotation - binding.compass.rotation

    binding.compass.animate().apply {
        isRotating = true
        rotationBy(rotationChange)
        duration = 500
        setListener(object : AnimatorListenerAdapter() {
            override fun onAnimationEnd(animation: Animator) {
                isRotating = false
            }
        })
    }.start()
}

```

The `updateOrientation` method uses the values of the `magnetometerReading` and `accelerometerReading` arrays to generate a rotation matrix and determine the device's orientation. Some quite detailed Physics goes into calculating these values which are not necessary for your understanding of this method; however, if you are interested in learning more then you can refer to the Android documentation on the `SensorEvents` class <https://developer.android.com/reference/android/hardware/SensorEvent>.

The orientation of the device is output as a Float array. In this instance, we are only interested in the first value of this array, which represents the angle between the device's y-axis and the north pole. The angle is then converted from radians to degrees and stored in a variable called `newRotation`. To determine how far to rotate the compass image, the difference between the `newRotation` value and the current rotation of the compass image is calculated and output to a variable called `rotationChange`.

Next, the above code uses the `ImageView` class's `animate` method to construct a `ViewPropertyAnimator` object and create a smooth rotation animation. While the animation is underway, the `isRotating` variable is set to `true` to prevent competing animations from being initiated. Once the rotation is complete, the `isRotating` variable is set back to `false` by the `onAnimationEnd` callback method.

For the final step, we must register listeners to the sensors that we want the app to monitor. To do this, add the following code to the bottom of the `onCreate` method:

```

sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER)?.also { accelerometer ->
    sensorManager.registerListener(
        this,
        accelerometer,
        SensorManager.SENSOR_DELAY_NORMAL,
        SensorManager.SENSOR_DELAY_UI
    )
}

sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD)?.also { magneticField ->
    sensorManager.registerListener(
        this,
        magneticField,
        SensorManager.SENSOR_DELAY_NORMAL,
        SensorManager.SENSOR_DELAY_UI
    )
}

```

The above code registers listeners to the accelerometer and magnetic field sensors. Whenever new data is available, it will be processed by the `onSensorChanged` callback method. For both listeners, the sampling period is set to `SENSOR_DELAY_NORMAL` and the reporting latency is set to `SENSOR_DELAY_UI`. Both values are defined in the `SensorManager` class and refer to predefined configurations for sensor data reporting (see <https://developer.android.com/reference/android/hardware/SensorManager>). Together, the values help ensure the application is provided with new sensor data at a regular rate that is suitable for updating the user interface. Sensor data is stored for the duration specified by the `SENSOR_DELAY_UI` value and discarded if the app is too busy

processing previous data. In this way, we ensure that sensor updates are processed efficiently and that power is not wasted processing surplus data.

## Summary

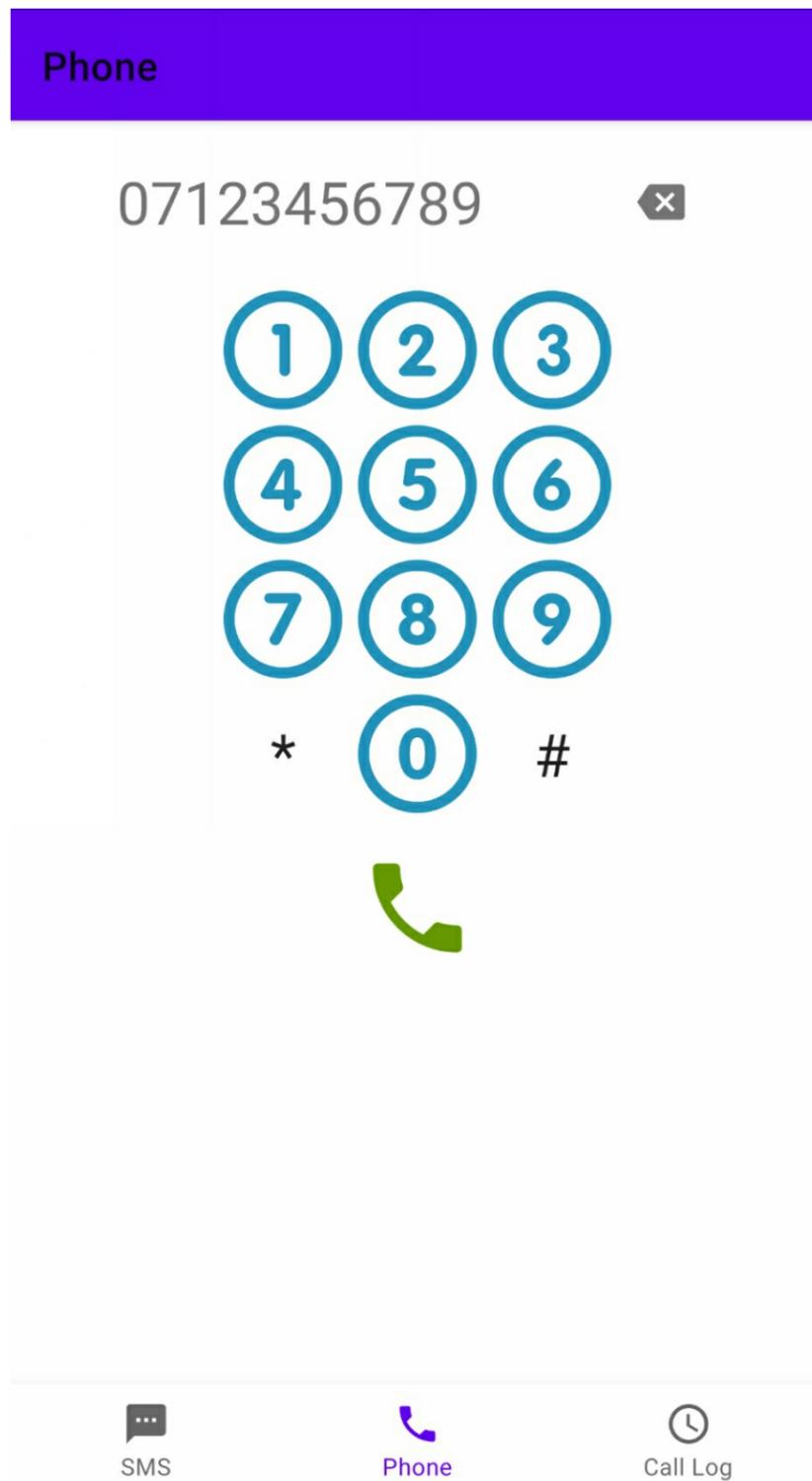
Congratulations on completing the Maps app! In creating this app, you have covered the following skills and topics:

- Creating a new application using the Google Maps Activity project template.
- Incorporate the Google Maps SDK into your app.
- Use Google Play's location services to find the device's current location.
- Initiate location requests and continuously monitor the device's location.
- Establish Geofences to receive alerts when the user enters, exits or dwells in a given location.
- Register a broadcast receiver to detect and respond to new broadcast events, such as Geofence alerts.
- Plot markers on the map.
- Use the CountdownTimer class to record how much time has elapsed and set a time limit for completing a task.
- Update the application Manifest file to inform the Google Play store that the app requires access to certain hardware such as the accelerometer sensor.
- Interact with sensors on the device to determine the device's orientation and the physical forces that are being applied.
- Use a ViewPropertyAnimator object to rotate an ImageView widget.

# How to create a phone call and SMS communications app

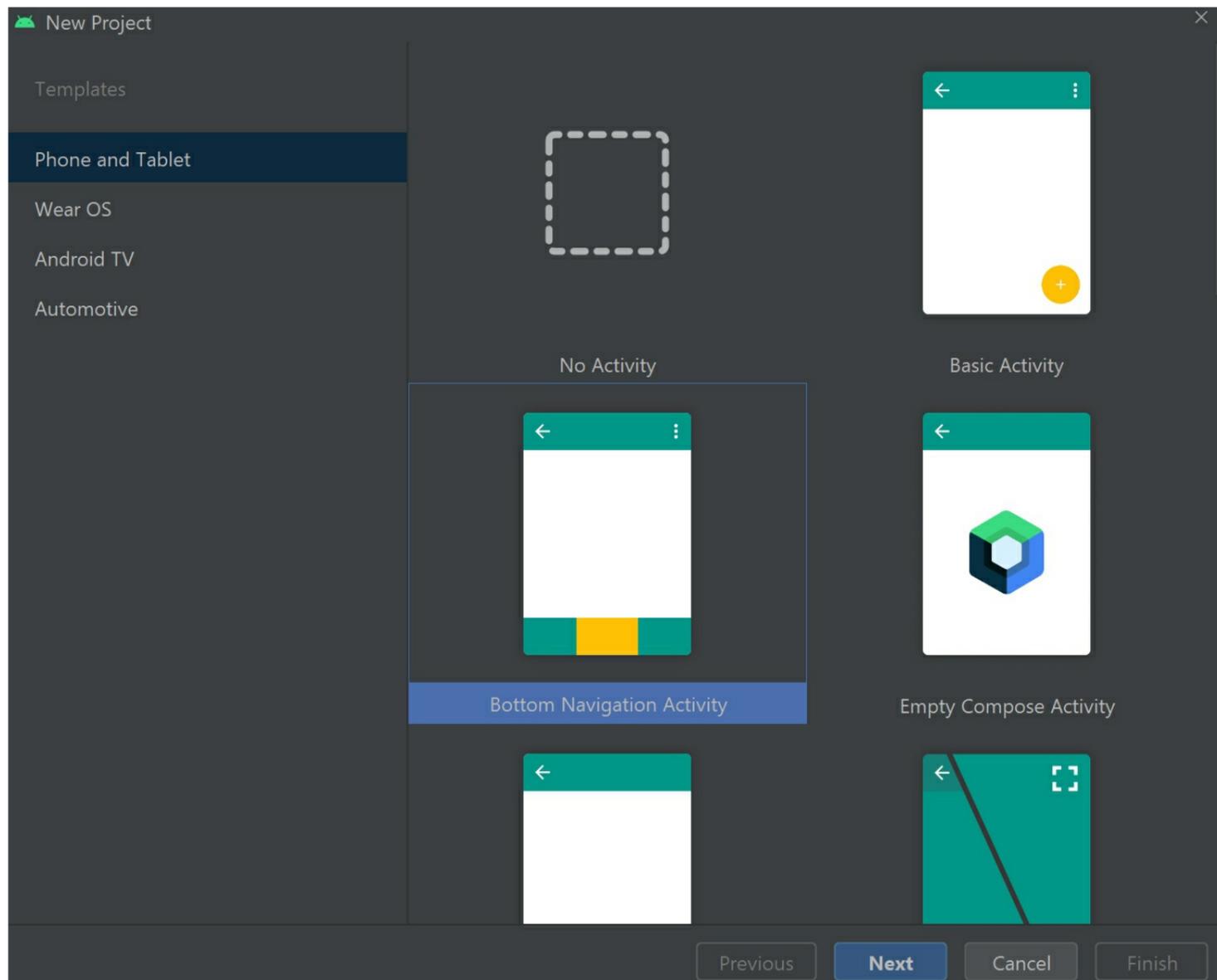
## Introduction

For this project, we will build a communications-based app that will enable the user to make phone calls, view their call history and browse and respond to SMS text messages. In creating this app, you will learn how to use a broadcast receiver to detect external events, such as when the device receives an SMS message, utilise the SMS manager to dispatch single or multipart messages, and initiate a phone call.

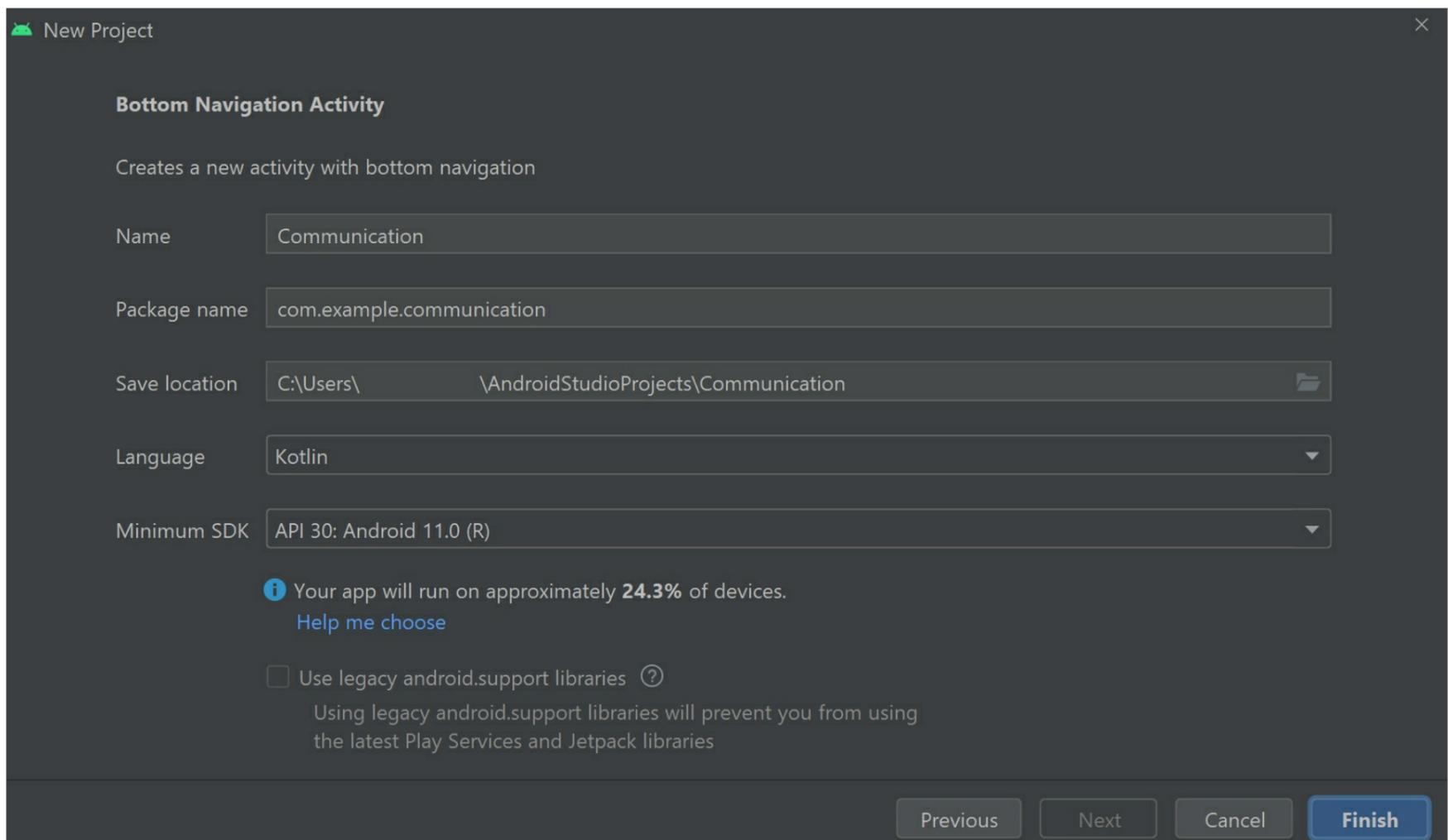


## Getting started

To begin, open Android Studio and create a new project using the Bottom Navigation Activity project template. You might remember this project template from the Camera application. The template provides your app with a navigation bar at the bottom of the screen, as well as several readymade fragments. Each fragment represents a different destination in the app, which the user will be able to navigate to using the navigation bar. In this app, there will be separate fragments for the user's call history, a phone dial pad and an SMS message log.



In the Create New Project window, name the project Communication, set the language to Kotlin and select API level 30.



It is recommended you enable Auto Imports to direct Android Studio to add any necessary import statements to your Kotlin files as you code. These import statements are essential for incorporating the external classes and tools required for the app to run. To enable Auto Imports, open Android Studio's Settings window by clicking **File > Settings**. In the Settings window, navigate through **Editor > General > Auto Import** then select 'Add unambiguous

imports on the fly' and 'Optimise imports on the fly' for both Java and Kotlin then press Apply and OK.

Android Studio should now add most of the necessary import statements to your Kotlin class files automatically. Sometimes there are multiple classes with the same name and the Auto Import feature will not work. In these instances, the requisite import statement(s) will be specified explicitly in the example code. You can also refer to the finished project code which accompanies this book to find the complete files including all import statements.

## Configuring the manifest file

Every Android application requires a manifest file. The manifest file offers an overview of the application's core components, such as activities and broadcast receivers, and the user permissions and hardware required to run. To configure the Communication app's manifest file, navigate through **Project > app > manifests**, open the **AndroidManifest.xml** file, and add the following code above the application element:

```
<uses-permission android:name="android.permission.CALL_PHONE"/>
<uses-permission android:name="android.permission.READ_SMS"/>
<uses-permission android:name="android.permission.SEND_SMS"/>
<uses-permission android:name="android.permission.RECEIVE_SMS" />
<uses-permission android:name="android.permission.READ_CALL_LOG"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

The above code defines uses-permission elements that signal to the device (and the Google Play store) that the app will require multiple permissions from the user. The required permissions include reading, receiving and sending SMS messages, viewing the call log and making new phone calls, and accessing the device's storage.

Moving on, add the following code inside the application element to define a broadcast receiver class called SMSBroadcastReceiver:

```
<receiver android:name=".SMSBroadcastReceiver"
    android:permission="android.permission.BROADCAST_SMS"
    android:exported="true">
    <intent-filter>
        <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
    </intent-filter>
</receiver>
```

The SMSBroadcastReceiver class is a broadcast receiver that we will create to detect incoming SMS messages. In the above code, we specify that the SMSBroadcastReceiver class will handle all SMS\_RECEIVED type actions, which occur when a new SMS message is delivered to the user's device. The permission attribute of the receiver is set to BROADCAST\_SMS, which means the broadcast receiver will only respond to intents from applications that have permission to acknowledge the receipt of new SMS messages. Invariably, SMS acknowledgement receipts will be emitted by the device's default SMS application only but the permission attribute helps ensure the SMS receipt is from a reliable source. Finally, the exported property is set to true so the receiver can be invoked by other applications.

## Defining the string resources used in the app

Like the other projects covered in this book, the Communication app will store all the strings of text used throughout the application in a resource file. To define the string resources, navigate through **Project > app > res** and open the file called **strings.xml**. Once the file opens in the editor, modify its contents so it reads as follows:

```
<resources>
    <string name="app_name">Communication</string>
    <string name="call_log">Call Log</string>
    <string name="cancel">Cancel</string>
    <string name="no_number">No number entered</string>
    <string name="phone">Phone</string>
    <string name="hash">#</string>
    <string name="make_call">Call this number</string>
    <string name="send_sms">Send SMS</string>
    <string name="sms">SMS</string>
```

```

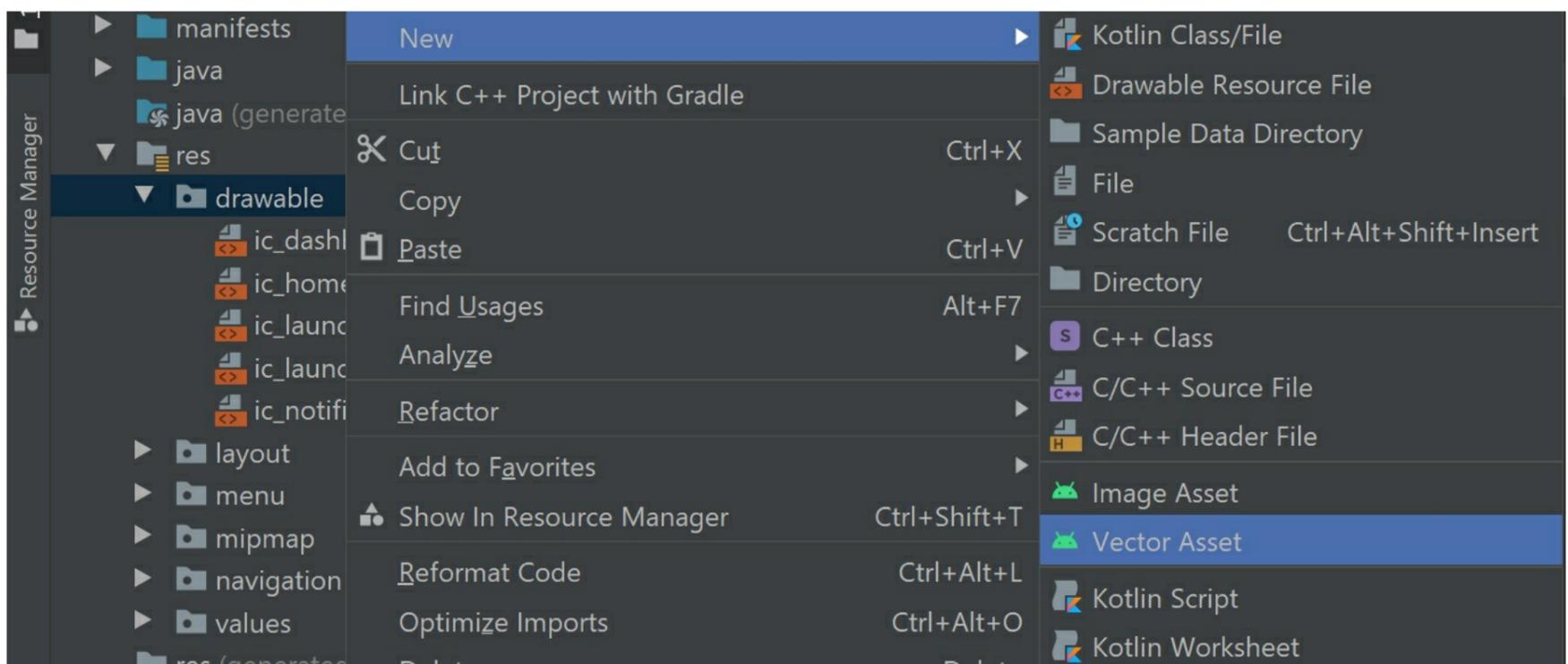
<string name="star">*</string>
<string name="block_unblock_number">Block/Unblock number</string>
<string name="call_this_number_back">Call this number back</string>
<string name="the_direction_of_the_call">The direction of the call</string>
<string name="number_keypad">Number on keypad</string>
<string name="backspace">Backspace</string>
<string name="number">Number</string>
<string name="message">Message</string>
<string name="reply">Reply</string>
<string name="send">Send</string>
<string name="view_message">View message</string>
<string name="error_sender">Could not load message sender</string>
<string name="error_body">Could not load message body</string>
<string name="error_sending_sms">Check the phone number and message are not empty</string>
<string name="new_sms_received">New message received: %1$s</string>
</resources>

```

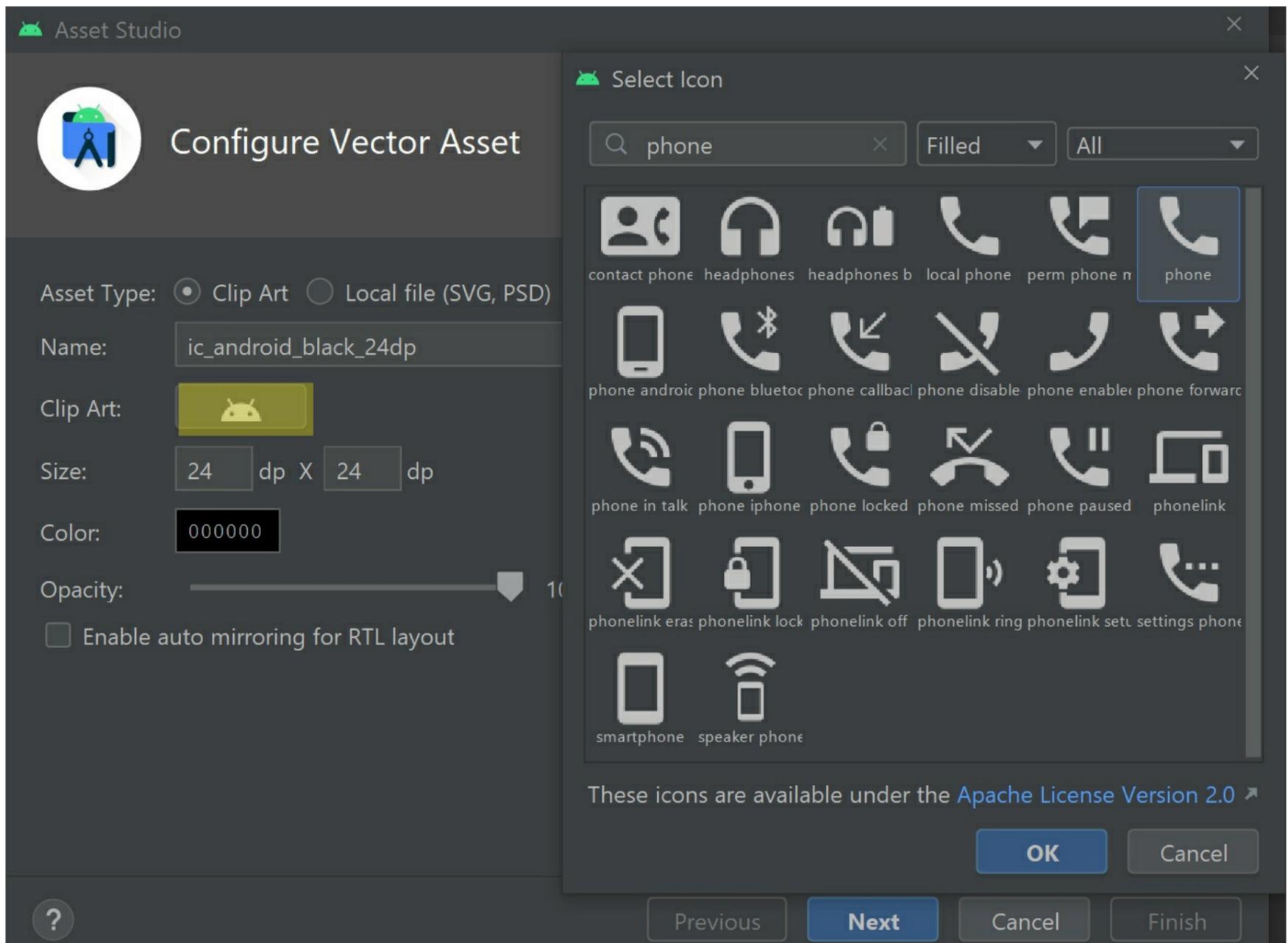
You'll notice these String resources crop up throughout the project. They will be used to display messages to the user, populate TextView widgets, provide content descriptions for images and more!

## Defining the drawable resources used in the project

The icons and graphics that are used in the app are called drawable resources. Android Studio comes with a set of readymade drawable resources which you can use to make your app look better. For example, you could create a drawable for a phone icon that will be used to identify the phone dial pad fragment in the bottom navigation bar. To create a new drawable resource, locate the **drawable** folder by navigating through **Project > app > res**. Right-click the folder then select **New > Vector Asset**.



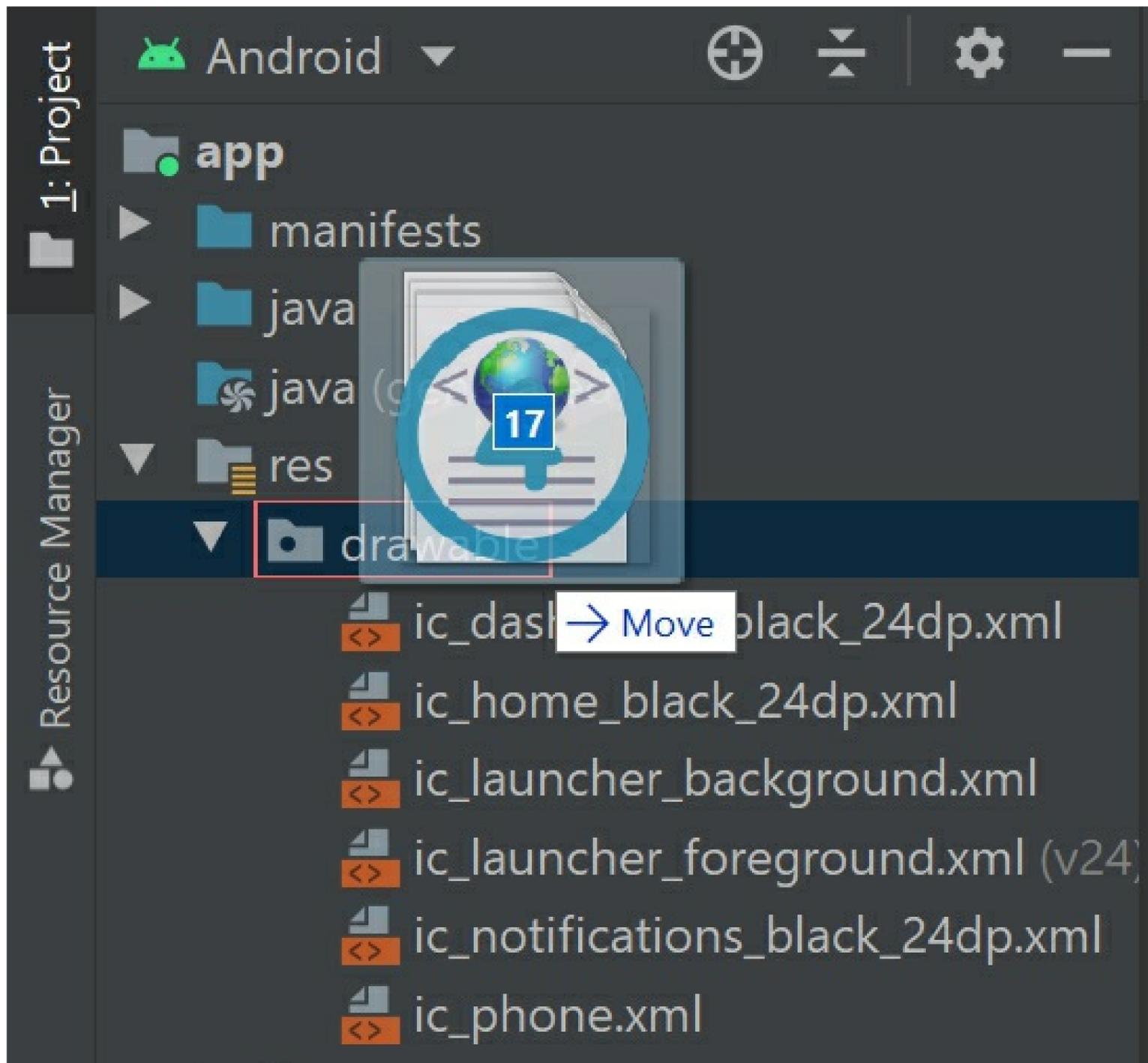
In the Configure Vector Asset window that opens, click on the clip art image then locate the phone icon and press OK.



Set the name of the vector asset to `ic_phone` then press Next and Finish to save the drawable. In this project, we will use many drawable resources because each button on the phone dial pad requires an icon and there are other resources used elsewhere also. It would be quite tedious to create each drawable one by one so instead, I would invite you to copy the source files from the example code into the **drawable** folder of your project. To locate the drawable files in the example code, navigate through the following folders: **app > src > main > res > drawable**. The files you should copy are highlighted below. You could also copy the unhighlighted files over; some are default drawable files created by Android Studio and others are custom files we will discuss later.

<input type="checkbox"/> Name	Date modified	Type	Size
<input checked="" type="checkbox"/>  eight	19/11/2020 11:34	PNG File	11 KB
<input checked="" type="checkbox"/>  five	19/11/2020 11:33	PNG File	10 KB
<input checked="" type="checkbox"/>  four	19/11/2020 11:33	PNG File	10 KB
<input checked="" type="checkbox"/>  ic_backspace	19/11/2020 13:39	XML Document	1 KB
 ic_dashboard_black_24dp	28/01/2022 17:03	XML Document	1 KB
 ic_home_black_24dp	28/01/2022 17:03	XML Document	1 KB
<input checked="" type="checkbox"/>  ic_incoming	12/12/2020 13:26	XML Document	1 KB
 ic_launcher_background	28/01/2022 17:03	XML Document	6 KB
<input checked="" type="checkbox"/>  ic_missed	12/12/2020 13:27	XML Document	1 KB
<input checked="" type="checkbox"/>  ic_new_sms	27/12/2020 23:21	XML Document	1 KB
 ic_notifications_black_24dp	28/01/2022 17:03	XML Document	1 KB
<input checked="" type="checkbox"/>  ic_outgoing	12/12/2020 13:26	XML Document	1 KB
 ic_phone	20/02/2021 23:35	XML Document	1 KB
<input checked="" type="checkbox"/>  ic_sms	22/11/2020 15:17	XML Document	1 KB
<input checked="" type="checkbox"/>  ic_time	07/12/2020 00:16	XML Document	1 KB
<input checked="" type="checkbox"/>  nine	19/11/2020 11:34	PNG File	11 KB
<input checked="" type="checkbox"/>  one	19/11/2020 11:32	PNG File	10 KB
<input checked="" type="checkbox"/>  seven	19/11/2020 11:34	PNG File	10 KB
<input checked="" type="checkbox"/>  six	19/11/2020 11:33	PNG File	11 KB
<input checked="" type="checkbox"/>  three	19/11/2020 11:32	PNG File	11 KB
<input checked="" type="checkbox"/>  two	19/11/2020 11:32	PNG File	10 KB
<input checked="" type="checkbox"/>  zero	19/11/2020 11:34	PNG File	11 KB

To copy image files into the Android Studio project, simply drag and drop them into the **drawable** directory as shown below:



Most of the icons used in this project are open source Material Design icons (see <https://material.io/resources/icons>), except for the images that are used for the numbers in the phone dial pad:

- **one.png**
- **two.png**
- **three.png**
- **four.png**
- **five.png**
- **six.png**
- **seven.png**
- **eight.png**
- **nine.png**
- **zero.png**

These icons were made by Pixel Buddha from [www.flaticon.com](http://www.flaticon.com) and are free for commercial use with attribution.

## Styling the phone's dial pad buttons

Let's now turn our attention to the theme resource **themes.xml** files, which determine the appearance of the app. For the Communication app, we will need to define custom style guidelines in the theme resource files to ensure the phone's dial pad buttons behave as expected. By default, Android Studio will generate resource files for the base (day) and night theme modes. To locate the resource files, navigate through **Project > app > res > values > themes**. The night mode **themes.xml** file will feature the word night in brackets after the filename, while the base theme file will not. Open the base **themes.xml** file in the editor and add the following style element to the file:

```

<style name="PhoneButton" parent="Widget.AppCompat.ImageButton">
  <item name="backgroundTint">@android:color/transparent</item>
  <item name="android:background">?attr/selectableItemBackgroundBorderless</item>
  <item name="android:scaleType">centerCrop</item>
  <item name="android:stateListAnimator">@null</item>
  <item name="android:padding">0dp</item>
  <item name="android:layout_marginHorizontal">20dp</item>
</style>

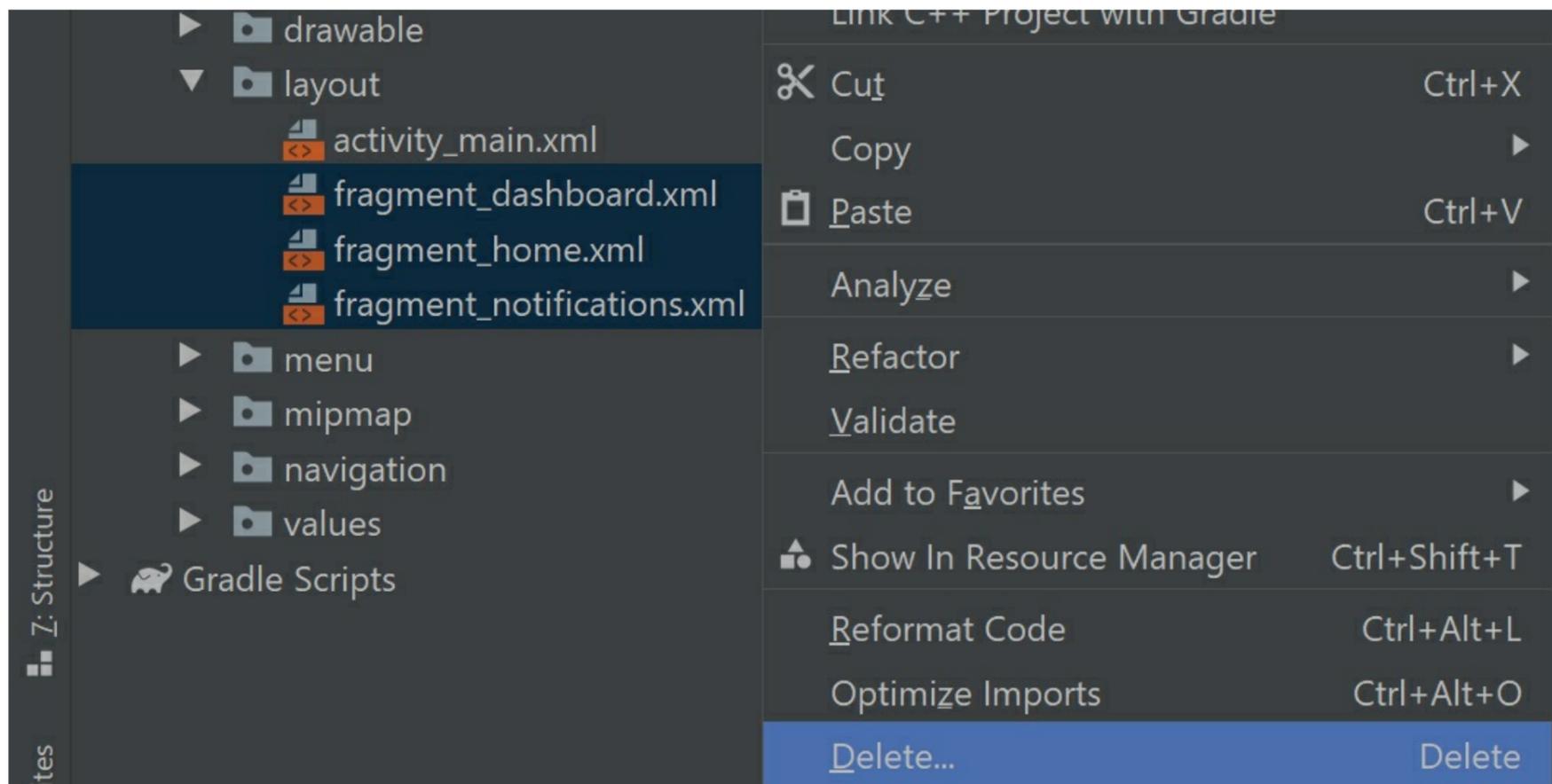
```

The above code defines a style element called PhoneButton. By default, the PhoneButton element will use all the regular style guidelines that apply to ImageButton widgets, as specified in the parent attribute. However, there are certain guidelines that we wish to change. To do this, we add items to the style element. Each item details the name of the property that we wish to customise and the value that should be used. First, the background tint is set to transparent because the phone dial pad buttons will not require a background colour. Next, the button's background is set to selectableItemBackgroundBorderless, which will create a borderless ripple effect when the button is pressed; the scaleType property is set to centerCrop, which will ensure the button's image is centrally positioned within the button's frame; and the stateListAnimator is set to null, which removes the small elevation and shadow that is applied to buttons by default. Altogether, these attributes create ImageButton widgets with centrally positioned images, no shadow, and a circular ripple effect that stretches beyond the rectangular border of the button when clicked. Finally, the padding is set to 0dp, which means there will be no space between the image inside the button and the button border, and the horizontal margin is set to 20dp, which will create a gap between neighbouring buttons.

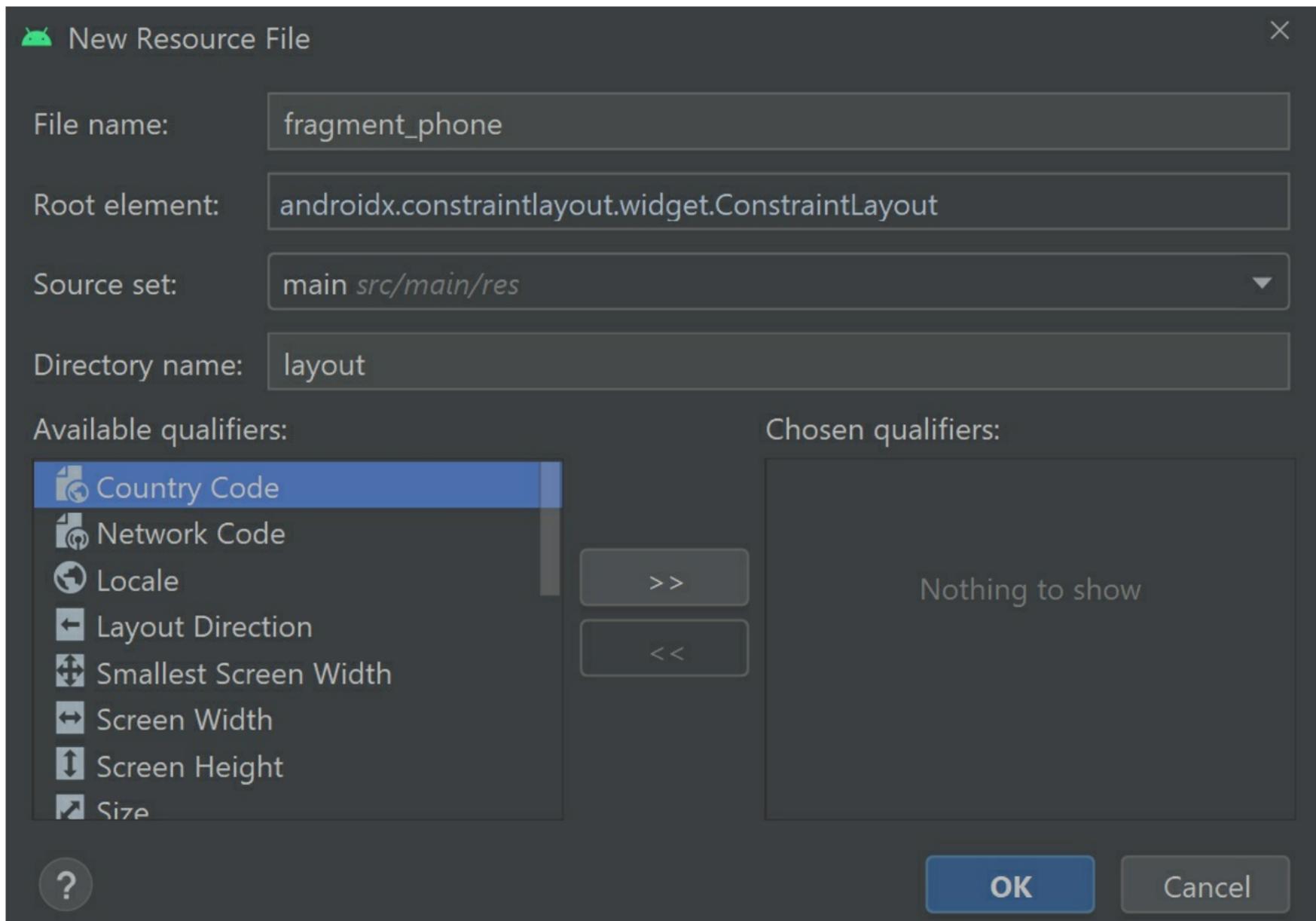
By default, the style elements defined in the base theme will also be applied to the night theme, unless the night theme **themes.xml** file contains conflicting guidelines. In case of a conflict, the style elements in the night **themes.xml** file will take priority when the night theme is active. We will not define custom guidelines for the night theme, so the PhoneButton style element defined above will apply to the night theme also.

## Designing the phone dial pad layout

The Communication app will contain a fragment that will allow the user to dial phone numbers and initiate calls. This fragment will require a user interface layout containing a dial pad, call button and text displaying the phone number the user has dialled. To arrange this, locate the **layout** resources directory by navigating through **Project > app > res**. Android Studio will already have generated three fragment layout files called fragment\_dashboard, fragment\_home and fragment\_notifications. These will not be used in this project so you can delete them.



Once those layout files have been deleted, create a new layout file by right-clicking the **layout** directory and selecting **New > Layout Resource File**. Name the layout fragment\_phone then click OK.



Once the `fragment_phone` layout is open in the editor, switch to Code view and replace the contents of the file with the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="14dp"
    android:orientation="vertical"
    android:gravity="center_horizontal"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:gravity="center_vertical">

        <HorizontalScrollView android:layout_width="250dp"
            android:layout_height="wrap_content">
            <TextView
                android:id="@+id/phoneNumber"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:textSize="30sp"
                android:singleLine="true"
                android:ellipsize="none"
                android:scrollHorizontally="true" />
            </HorizontalScrollView>

        <ImageButton
            android:id="@+id/backspace"
            android:layout_width="54dp"
            android:layout_height="54dp"
            />
    </LinearLayout>
</LinearLayout>
```

```
    android:src="@drawable/ic_backspace"  
    android:contentDescription="@string/backspace"  
    style="@style/Widget.AppCompat.ActionButton.Overflow" />
```

```
</LinearLayout>
```

```
<GridLayout
```

```
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginVertical="14dp"  
    android:useDefaultMargins="true"  
    android:columnCount="3"  
    android:rowCount="4">
```

```
<ImageButton
```

```
    android:id="@+id/one"  
    android:layout_width="60dp"  
    android:layout_height="60dp"  
    android:src="@drawable/one"  
    android:contentDescription="@string/number_keypad"  
    style="@style/PhoneButton" />
```

```
<ImageButton
```

```
    android:id="@+id/two"  
    android:layout_width="60dp"  
    android:layout_height="60dp"  
    android:src="@drawable/two"  
    android:contentDescription="@string/number_keypad"  
    style="@style/PhoneButton" />
```

```
<ImageButton
```

```
    android:id="@+id/three"  
    android:layout_width="60dp"  
    android:layout_height="60dp"  
    android:src="@drawable/three"  
    android:contentDescription="@string/number_keypad"  
    style="@style/PhoneButton" />
```

```
<ImageButton
```

```
    android:id="@+id/four"  
    android:layout_width="60dp"  
    android:layout_height="60dp"  
    android:src="@drawable/four"  
    android:contentDescription="@string/number_keypad"  
    style="@style/PhoneButton" />
```

```
<ImageButton
```

```
    android:id="@+id/five"  
    android:layout_width="60dp"  
    android:layout_height="60dp"  
    android:src="@drawable/five"  
    android:contentDescription="@string/number_keypad"  
    style="@style/PhoneButton" />
```

```
<ImageButton
```

```
    android:id="@+id/six"  
    android:layout_width="60dp"  
    android:layout_height="60dp"  
    android:src="@drawable/six"  
    android:contentDescription="@string/number_keypad"  
    style="@style/PhoneButton" />
```

```
<ImageButton
```

```
    android:id="@+id/seven"  
    android:layout_width="60dp"
```

```
    android:layout_height="60dp"
    android:src="@drawable/seven"
    android:contentDescription="@string/number_keypad"
    style="@style/PhoneButton" />
```

```
<ImageButton
    android:id="@+id/eight"
    android:layout_width="60dp"
    android:layout_height="60dp"
    android:src="@drawable/eight"
    android:contentDescription="@string/number_keypad"
    style="@style/PhoneButton" />
```

```
<ImageButton
    android:id="@+id/nine"
    android:layout_width="60dp"
    android:layout_height="60dp"
    android:src="@drawable/nine"
    android:contentDescription="@string/number_keypad"
    style="@style/PhoneButton" />
```

```
<Button
    android:id="@+id/star"
    android:layout_width="60dp"
    android:layout_height="60dp"
    android:text="@string/star"
    android:textSize="30sp"
    android:textColor="@color/material_on_background_emphasis_high_type"
    style="@style/PhoneButton" />
```

```
<ImageButton
    android:id="@+id/zero"
    android:layout_width="60dp"
    android:layout_height="60dp"
    android:src="@drawable/zero"
    android:contentDescription="@string/number_keypad"
    style="@style/PhoneButton" />
```

```
<Button
    android:id="@+id/hash"
    android:layout_width="60dp"
    android:layout_height="60dp"
    android:text="@string/hash"
    android:textSize="30sp"
    android:textColor="@color/material_on_background_emphasis_high_type"
    style="@style/PhoneButton" />
```

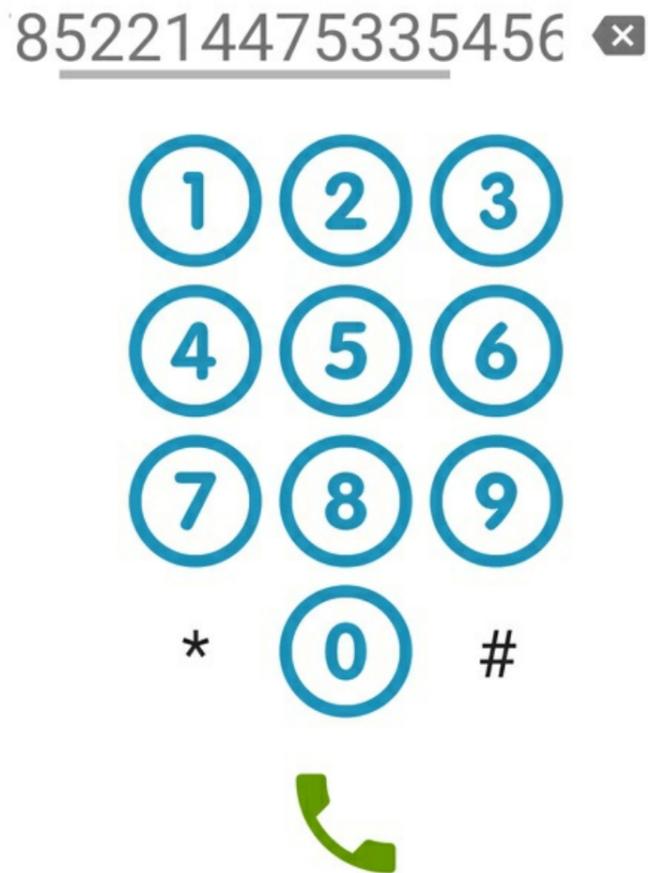
```
</GridLayout>
```

```
<ImageButton
    android:id="@+id/callButton"
    android:layout_width="60dp"
    android:layout_height="60dp"
    android:contentDescription="@string/make_call"
    android:src="@drawable/ic_phone"
    app:tint="@android:color/holo_green_dark"
    style="@style/PhoneButton" />
```

```
</LinearLayout>
```

The above code sets the root element of the fragment\_phone layout to a LinearLayout widget. The LinearLayout widget has an orientation property set to vertical and a gravity property set to center\_horizontal. Taken together, these properties will align the contents of the LinearLayout widget vertically down the center of the layout. The first component inside the root element is another LinearLayout widget. This child LinearLayout widget will horizontally align a TextView widget that will display the phone number that the user is dialling and a backspace

button. The TextView widget is enclosed in a HorizontalScrollView widget which will create a horizontal scroll bar if the contents of the TextView is too large to fit inside the container, such as if the user dials an especially long phone number. To facilitate the scrolling functionality, the TextView widget is directed to only occupy a single line, scroll horizontally and not use an ellipsis if the TextView's content is too large to fit inside the container. If the user wishes to delete a digit from the phone number then they can click the backspace button that appears on the right-hand side of the TextView widget.

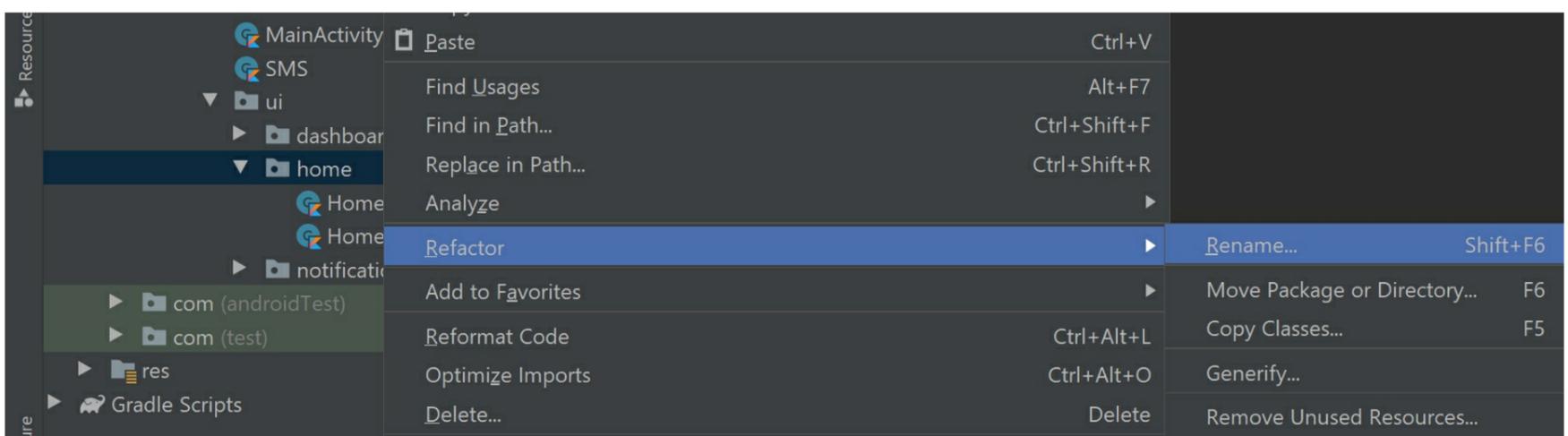


The next element in the layout is a GridLayout widget. The GridLayout widget will coordinate the buttons that comprise the phone dial pad in a grid. Altogether, the grid will comprise four rows and three columns. The GridLayout widget has a useDefaultMargins property set to true, which means Android will automatically apply margins to the contents of the GridLayout unless we explicitly declare margins ourselves. Each button in the phone dial pad has a style attribute that references the custom PhoneButton style guidelines we defined earlier in the base **themes.xml** file. As you may recall, the PhoneButton style guidelines centrally position each button's icon within the button container and create a ripple effect when the button is pressed.

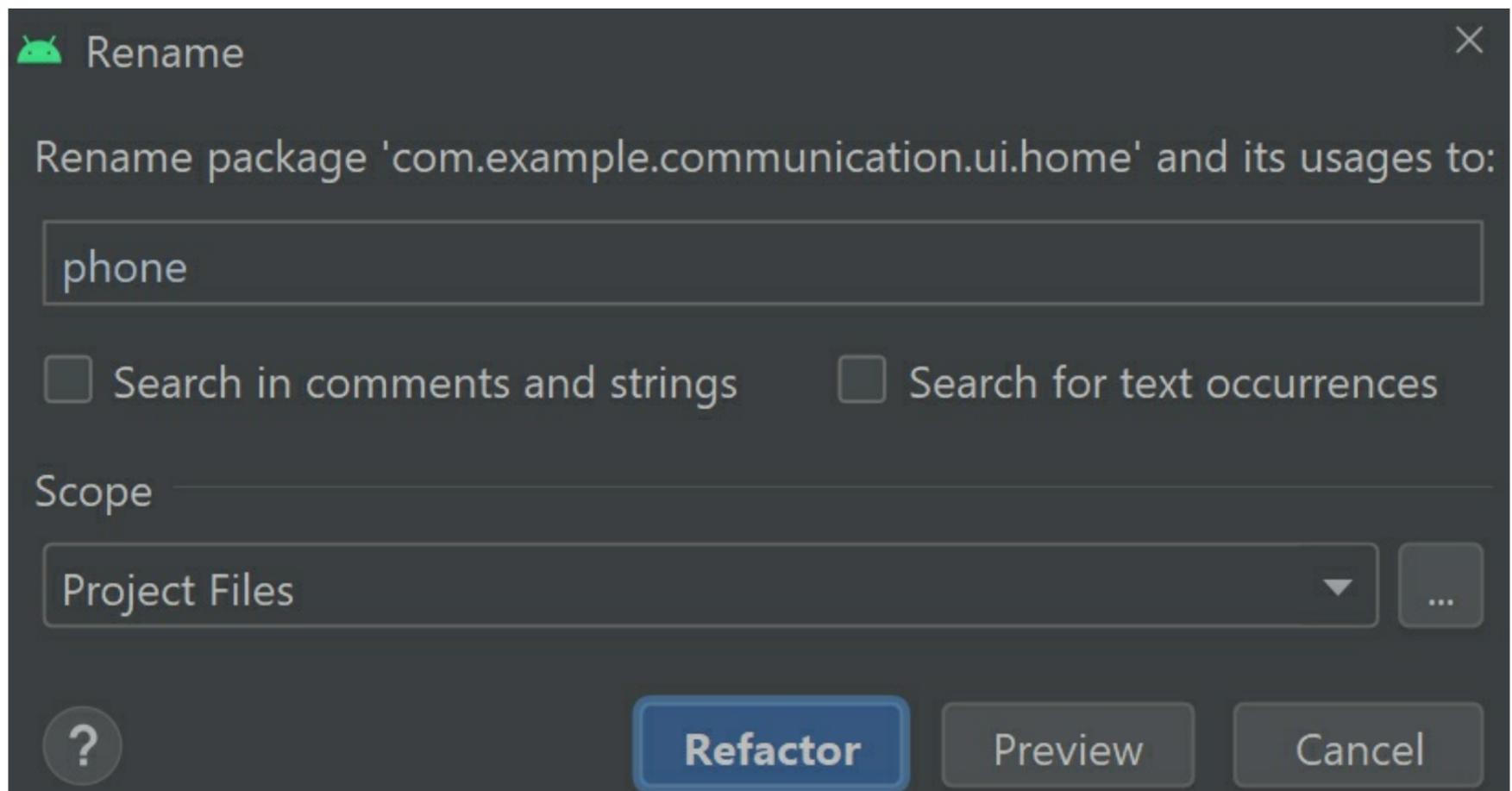
Lastly, the final widget in the layout is an ImageButton widget that will display a phone icon. The user can press this button to initiate a phone call for the number they have dialled. A green tint is applied to the widget for styling purposes.

## Creating the phone dial pad fragment

When creating a project using the Bottom Navigation Activity template, Android Studio will automatically generate three fragments called Home, Dashboard and Notifications. To repurpose these fragments for our project, we will refactor them. Refactoring is the process of renaming an artefact and automatically updating all references to the artefact. First, locate and right-click the **home** directory by navigating through **Project > app > java > name of the project > ui**. Next, select **Refactor > Rename**.



Change the name to phone then click Refactor.



Next, look at the files inside the newly refactored phone directory and delete the file called **HomeViewModel.kt**. There is also a fragment in the directory called **HomeFragment.kt**. Right-click this file and refactor its name to PhoneFragment. Once that's done, open the file in the editor and replace the homeViewModel and \_binding variables at the top of the class with the following code:

```
private var _binding: FragmentPhoneBinding? = null
private lateinit var callingActivity: MainActivity
```

The above code defines a binding variable that will provide access to the fragment\_phone layout via its binding class. A variable called callingActivity is also defined and will hold a reference to the MainActivity class. To initialise the variables, edit the onCreateView method so it reads as follows:

```
override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
): View {
    _binding = FragmentPhoneBinding.inflate(inflater, container, false)
    callingActivity = activity as MainActivity
    return binding.root
}
```

The foundations of the fragment are now in place. For our next task, we will configure the fragment to respond to clicks on the phone's dial pad buttons. First, add the following code below the onCreateView method:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    binding.backspace.setOnClickListener {
        removeDigit()
    }

    binding.one.setOnClickListener {
        addDigit("1")
    }

    binding.two.setOnClickListener {
        addDigit("2")
    }

    binding.three.setOnClickListener {
```

```

        addDigit("3")
    }

    binding.four.setOnClickListener {
        addDigit("4")
    }

    binding.five.setOnClickListener {
        addDigit("5")
    }

    binding.six.setOnClickListener {
        addDigit("6")
    }

    binding.seven.setOnClickListener {
        addDigit("7")
    }

    binding.eight.setOnClickListener {
        addDigit("8")
    }

    binding.nine.setOnClickListener {
        addDigit("9")
    }

    binding.star.setOnClickListener {
        addDigit("*")
    }

    binding.zero.setOnClickListener {
        addDigit("0")
    }

    binding.hash.setOnClickListener {
        addDigit("#")
    }

    binding.callButton.setOnClickListener {
        val number = binding.phoneNumber.text
        if (number.isNotBlank()) callingActivity.callNumber(number.toString())
        else Toast.makeText(callingActivity, getString(R.string.no_number), Toast.LENGTH_SHORT).show()
    }
}

```

The `onViewCreated` method runs after the `onCreateView` method and signifies that the view hierarchy of the fragment's layout has been established and the fragment can interact with the layout's widgets. In this case, we direct the `onViewCreated` method to apply `onClick` listeners to each button in the `fragment_phone` layout. For the backspace button, clicks will trigger a method called `removeDigit` that will remove the last digit from the dialled phone number. Meanwhile, most of the other buttons will run a method called `addDigit`, which will add the corresponding dial pad digit to the dialled phone number. Finally, the call button will retrieve the complete dialled number and initiate a phone call using the `MainActivity` class. However, if no digits have been dialled, then rather than initiate a phone call, the fragment will display a toast notification informing the user that a phone number has not been entered.

Let's now define the `removeDigit` and `addDigit` methods. Add the following code below the `onViewCreated` method:

```

private fun addDigit(digit: String) {
    val previousNumber = binding.phoneNumber.text.toString()
    val newNumber = previousNumber + digit
    binding.phoneNumber.text = newNumber
}

private fun removeDigit() {

```

```

val previousNumber = binding.phoneNumber.text.toString()
if (previousNumber.isEmpty()) return
val newNumber = previousNumber.take(previousNumber.length - 1)
binding.phoneNumber.text = newNumber
}

```

The `addDigit` method retrieves the phone number the user has dialled from the `phoneNumber` `TextView` widget and appends the digit associated with the button the user just pressed. For example, if the user had dialled '07123' then pressed the '4' button, then the '4' character would be appended to the string of dialled numbers and loaded into the `phoneNumber` `TextView` widget. The `phoneNumber` `TextView` widget would then display a value of '071234'.

Meanwhile, the `removeDigit` method reverses the `addDigit` method by removing the last digit from the dialled phone number and loading the amended number into the `phoneNumber` `TextView` widget. For example, if the user pressed the backspace button when '071234' was loaded into the `TextView` widget then the '4' character would be removed and '07123' would be displayed instead. This is achieved using Kotlin's `take` method to remove the last character from the string of text loaded into the `TextView` widget. The index of the last character in a string is the length of the string minus one because the first character in a string always has an index of 0.

## Initiating phone calls

When the user presses the call button in the phone fragment, the phone number dialled by the user is sent to a `MainActivity` method called `callNumber` to initiate the phone call. To define the `callNumber` method, open the `MainActivity.kt` file (**Project > app > java > name of the project**) and add the following code below the `onCreate` method:

```

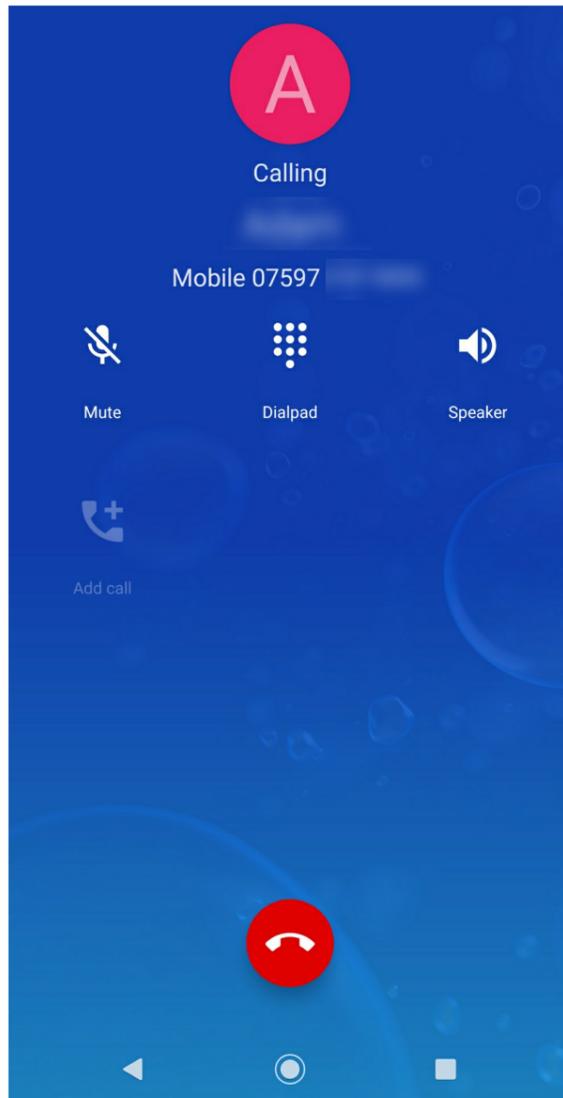
fun callNumber(number: String) {
    if (ContextCompat.checkSelfPermission(this, Manifest.permission.CALL_PHONE)
        == PackageManager.PERMISSION_GRANTED) {
        val intent = Intent(Intent.ACTION_CALL)
        intent.data = Uri.parse("tel:$number")
        startActivity(intent)
    } else ActivityCompat.requestPermissions(this,
        arrayOf(Manifest.permission.CALL_PHONE), 0)
}

```

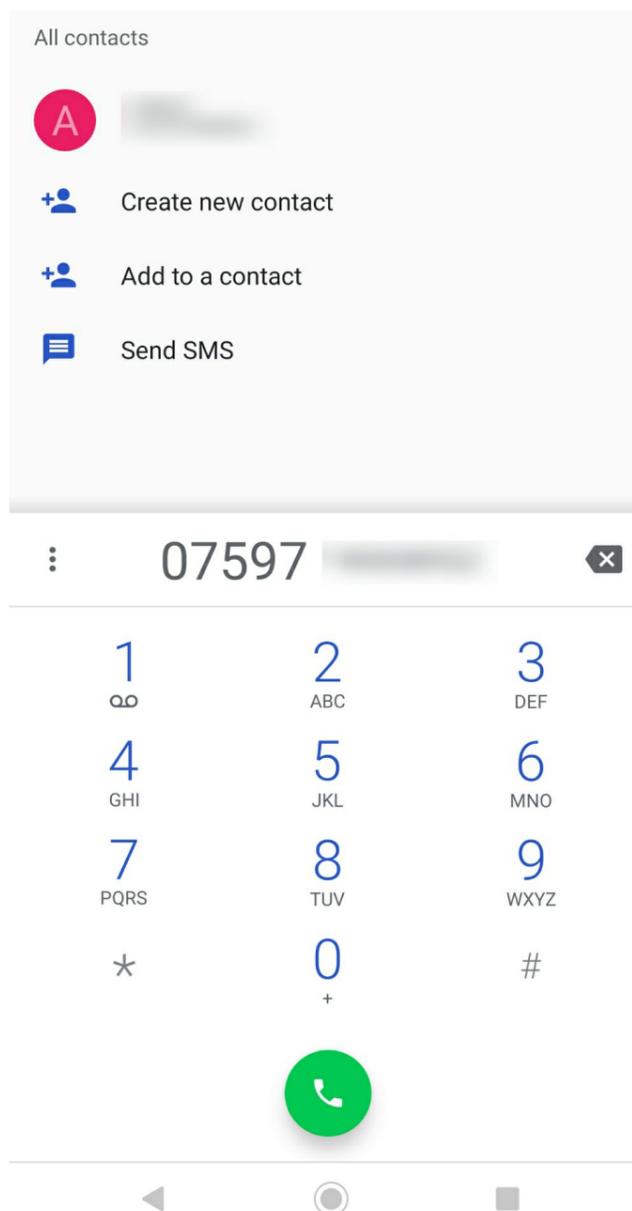
Note you may need to add the following import statement to the top of the file:

```
import android.Manifest
```

Before initiating a phone call, the app must check it has permission from the user. If the user has not granted the app permission to initiate a phone call, then a dialog window will appear and request the relevant permission. Meanwhile, if permission has been granted, the `callNumber` method will convert the phone number to a URI and package the URI in an intent. The intent has an action of `ACTION_CALL`, which means the intent will initiate a phone call with the phone number specified in the URI, as shown below.

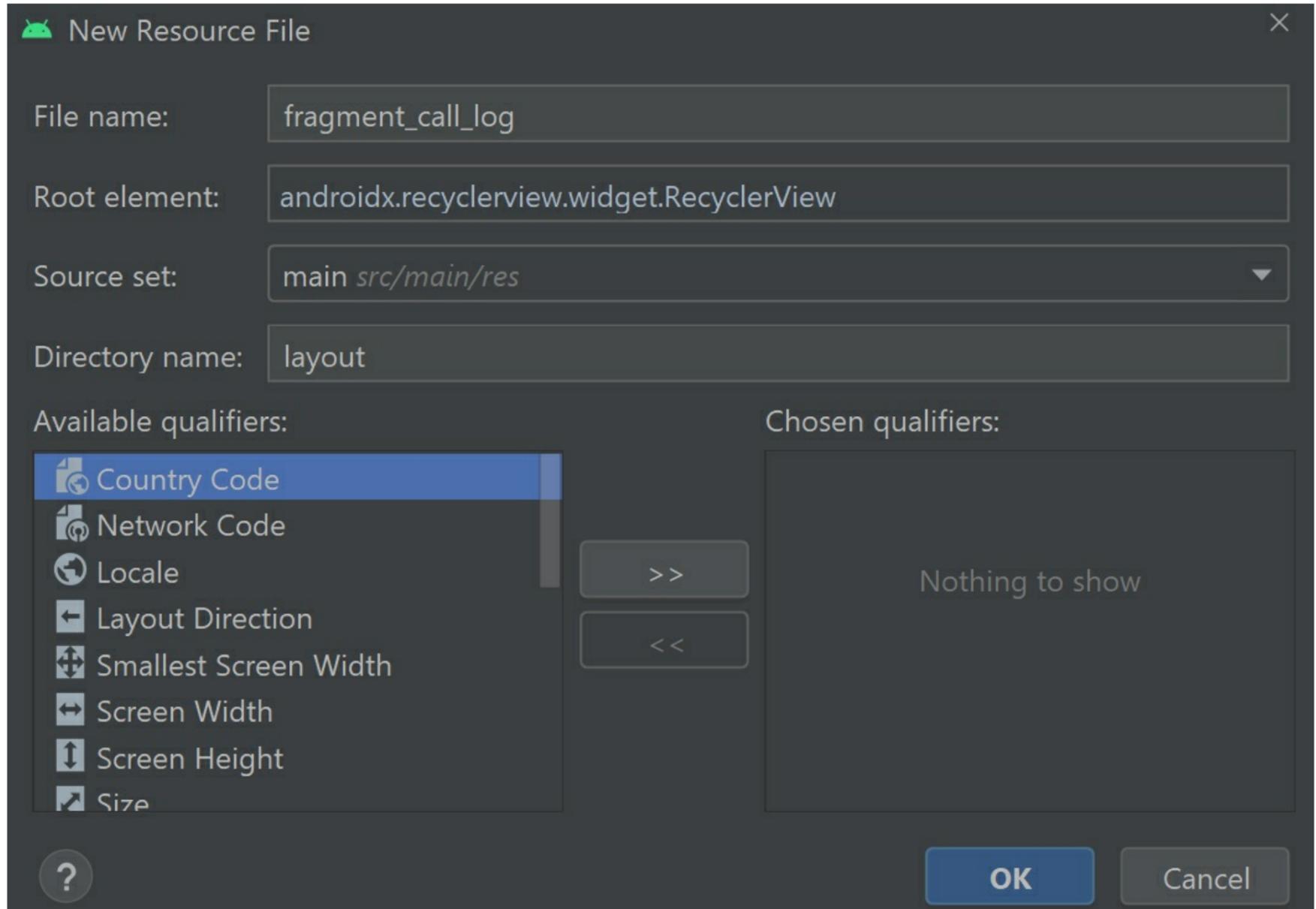


The ACTION\_CALL intent action will prompt the device to dial the phone number and initiate a call. If you wish to dial the phone number without initiating a call, then you should use the ACTION\_DIAL intent action instead. The ACTION\_DIAL action will send the phone number to the device's default phone application, as shown below.



## Designing the phone call log layout

In addition to initiating phone calls, the Communication app will also allow the user to view their call log history and return phone calls if they wish. The list of call log entries will be presented in a RecyclerView widget. To create the layout file that will store the RecyclerView, right-click the **layout** directory (**Project > app > res**) then select **New > Layout Resource File**. Name the file `fragment_call_log`, set the root element to `androidx.recyclerview.widget.RecyclerView` then click OK. The layout will comprise a RecyclerView widget only so we do not need to make any further changes to the file.



A separate layout will be required to display information about each call log entry. For example, we will look to display the caller ID and timestamp of the call and provide the user with a button to return the phone call.



Create another Layout Resource File in the **layout** directory and name it `call_log_entry`. Once the layout opens in the editor, switch to Code view and edit the file so it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="12dp"
    android:background="?attr/selectableItemBackground">
```

```
<ImageView
    android:id="@+id/callDirection"
    android:layout_width="30dp"
    android:layout_height="30dp"
    android:src="@drawable/ic_incoming">
```

```

android:contentDescription="@string/the_direction_of_the_call"
android:layout_alignParentStart="true"
android:layout_centerVertical="true" />

```

```

<LinearLayout
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:layout_toEndOf="@id/callDirection"
    android:layout_toStartOf="@id/callBack"
    android:layout_marginHorizontal="8dp"
    android:layout_centerVertical="true">

```

```

<TextView
    android:id="@+id/number"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    android:singleLine="true"
    android:textColor="@color/material_on_surface_emphasis_high_type"
    android:textSize="16sp" />

```

```

<TextView
    android:id="@+id/date"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:singleLine="true"
    android:textSize="14sp"
    android:textColor="@color/material_on_surface_emphasis_medium" />
</LinearLayout>

```

```

<ImageButton
    android:id="@+id/callBack"
    android:layout_width="50dp"
    android:layout_height="50dp"
    android:src="@drawable/ic_phone"
    android:contentDescription="@string/call_this_number_back"
    android:layout_alignParentEnd="true"
    android:layout_centerVertical="true"
    style="@style/Widget.AppCompat.ActionButton.Overflow" />
</RelativeLayout>

```

The root element of the `call_log_entry` layout is a `RelativeLayout` widget. `RelativeLayout` widgets allow you to position elements relative to one another and the parent layout. In the above code, we set the `RelativeLayout` widget's `background` property to `selectableItemBackground`, which will create a ripple effect when the layout is pressed. The ripple will help the user see which call log entry they have selected. Inside the `RelativeLayout`, the first child element is an `ImageView` widget. The image and tint of the `ImageView` widget will vary depending on the direction of the phone call (incoming, outgoing and missed) as shown below.

Outgoing      Incoming      Missed



Next, a vertically-oriented `LinearLayout` widget is introduced. The `LinearLayout` will contain two `TextView` widgets that will display the phone number associated with the call log entry and the timestamp of the call log event, respectively. Both `TextView` widgets feature a `singleLine` attribute that will restrict their contents to one line. The widgets also feature `textColor` and `textSize` attributes that ensure the text in the top `TextView` widget has a greater emphasis and is slightly larger than the bottom `TextView`, as shown below:

# +447123456789

## 26/02/2021 12:20 AM

To position the LinearLayout widget in the center of the layout, the widget contains RelativeLayout alignment properties that bind the widget to the right-hand side of the call direction ImageView and the left-hand side of an ImageButton widget with the ID callBack. The callBack ImageButton is the final widget in the layout. It will appear on the right-hand side and will contain an image of a phone icon. The user can click the button to phone the number associated with the call log event.

### Creating the call log fragment and call log adapter

Similar to the steps for creating the phone fragment, we need to repurpose one of the packages that were autogenerated by Android Studio to handle the call log. To do this, right-click the **notifications** directory (**Project > app > java > name of the project > ui**) then select **Refactor > Rename** and set the new name to callLog. Next, explore the **callLog** directory, delete the NotificationsViewModel class and refactor the NotificationsFragment class to CallLogFragment. Open the newly refactored CallLogFragment class in the editor and replace the notificationViewModel and \_binding variables with the following code:

```
private var _binding: FragmentCallLogBinding? = null
private lateinit var callingActivity: MainActivity
private lateinit var callLogAdapter: CallLogAdapter
```

In the above code, the binding variable will store an instance of the fragment\_call\_log layout's binding class, which the fragment will use to access the components of the layout. Meanwhile, the callingActivity variable will provide access to the MainActivity class and all its public methods and variables, and the callLogAdapter variable will store an instance of a RecyclerView adapter class called CallLogAdapter that we will create shortly. To initialise the binding and callingActivity variables, edit the fragment's onCreateView method so it reads as follows:

```
override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
): View {
    _binding = FragmentCallLogBinding.inflate(inflater, container, false)
    callingActivity = activity as MainActivity

    // TODO: Initialise the callLogAdapter variable here

    return binding.root
}
```

The binding and callingActivity variables have now been initialised and can be used elsewhere in the fragment. A TODO comment has also been included to indicate where the callLogAdapter variable will be initialised once the adapter class has been created. The adapter will coordinate the list of call log events and help handle user interactions. To create the adapter class, right-click the **callLog** directory then select **New > Kotlin Class/File**. Name the file CallLogAdapter and select Class from the list of options. Once the CallLogAdapter class is open in the editor, modify its code so it reads as follows:

```
class CallLogAdapter(private val activity: MainActivity) :
    RecyclerView.Adapter<CallLogAdapter.CallLogViewHolder>() {

    // TODO: Define the CallLogViewHolder inner class here

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): CallLogViewHolder {
        return CallLogViewHolder(LayoutInflater.from(parent.context).inflate(R.layout.call_log_entry, parent, false))
    }
}
```

Every RecyclerView adapter must implement several mandatory methods such as onCreateViewHolder. The onCreateViewHolder method prepares the view holder(s) that will display items in the RecyclerView. In this case,

each item in the RecyclerView will be a call log event so we only need one type of view holder. In the above code, the onCreateViewHolder method specifies that the view holder will be managed by a class called CallLogViewHolder, and the CallLogViewHolder class will use the **call\_log\_entry.xml** layout to coordinate its contents.

Let's now define the CallLogViewHolder class. Replace the TODO comment at the top of the CallLogAdapter class with the following code:

```
inner class CallLogViewHolder(itemView: View) :  
    RecyclerView.ViewHolder(itemView) {  
  
    internal var mDirection = itemView.findViewById<View>(R.id.callDirection) as ImageView  
    internal var mPhoneNumber = itemView.findViewById<View>(R.id.number) as TextView  
    internal var mCallDate = itemView.findViewById<View>(R.id.date) as TextView  
    internal var mCallback = itemView.findViewById<View>(R.id.callBack) as ImageButton  
  
    init {  
        itemView.setOnLongClickListener{  
            // TODO: Open the CallLogOptions dialog here  
            return@setOnLongClickListener true  
        }  
    }  
}
```

The CallLogViewHolder class contains variables for each widget within the call\_log\_entry layout. These variables will allow the adapter to interact with each layout component and access and modify their properties. The class also contains an init block, which is where we will handle user interactions. In this case, an onLongClick listener is applied to the entire view holder. Once fully implemented, the onLongClick listener will open a dialog window that allows the user to call or send an SMS message to the number associated with the call log event.

Now the CallLogAdapter class has been created, return to the **CallLogFragment.kt** file and replace the TODO comment in the onCreateView method with the following code to initialise the callLogAdapter variable:

```
callLogAdapter = CallLogAdapter(callingActivity)
```

The CallLogFragment class can now interact with the adapter; however, there is more work to do, and we will return to these classes once we have written the code that retrieves the device's call history.

## Retrieving the device's call history

The call history will be retrieved by the MainActivity class and stored in a view model. The call log will comprise a list of call history events, each representing a different incoming, outgoing or missed call. To store information about each event, we need to create a data class. Right-click the directory that contains the **MainActivity.kt** file (**Project > app > java > name of the project**) and select **New > Kotlin Class/File**. Name the file **CallLogEvent** and select Data Class from the list of options. A file called **CallLogEvent.kt** should then open in the editor. Modify the CallLogEvent class so it reads as follows:

```
data class CallLogEvent(  
    var direction: String?,  
    var number: String,  
    var date: String  
)
```

The CallLogEvent class contains three string parameters: a call direction (incoming, outgoing or missed); the phone number involved in the call event; and a timestamp of when the call event occurred. The value of the direction parameter can be null (as indicated by the question mark after the string data type declaration) but the number and date variables must always contain a value.

Moving on, let's now create the view model that will store the list of CallLogEvent objects. Right-click the directory that contains the **MainActivity.kt** file (**Project > app > java > name of the project**) and select **New > Kotlin Class/File**. Name the file **CommunicationViewModel** and select Class from the list of options. Once the **CommunicationViewModel.kt** file opens in the editor, edit the file's code as follows:

```
class CommunicationViewModel : ViewModel() {  
    var callLog = MutableLiveData<List<CallLogEvent>>()
```

```
}
```

The above code defines a variable called callLog, which will store a MutableLiveData list of CallLogEvent objects. MutableLiveData can be observed by different areas of the app. Any area that registers an observer will be notified whenever the underlying data changes. In this way, the CallLogFragment class can register an observer and update the user interface whenever a new call log event occurs.

Once the CallLogEvent data class and view model are in place, we can return to the MainActivity class and retrieve the call history from the user's device. Open the **MainActivity.kt** file (**Project > app > java > name of the project**) and initialise the view model by adding the following variable to the top of the class:

```
private val communicationViewModel: CommunicationViewModel by viewModels()
```

Note you may need to add the following import statement to the top of the file:

```
import androidx.activity.viewModels
```

Next, add the following method below the callNumber method to retrieve the call log:

```
fun getCallLogs() {
    val readStoragePermission = ContextCompat.checkSelfPermission(this,
Manifest.permission.READ_EXTERNAL_STORAGE)
    val readCallLogPermission = ContextCompat.checkSelfPermission(this,
Manifest.permission.READ_CALL_LOG)

    if (readStoragePermission != PackageManager.PERMISSION_GRANTED ||
readCallLogPermission != PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(this, arrayOf(Manifest.permission.READ_EXTERNAL_STORAGE,
Manifest.permission.READ_CALL_LOG), READ_STORAGE_REQUEST_CODE)
    }
    return
}

// TODO: Retrieve call log here
}
```

The getCallLogs method checks whether the user has granted permission for the app to read the device's external storage and call log history. If both permissions have been granted, then the method will continue as normal; however, if either permission has not been granted then a dialog window will request the user grant the relevant permissions. The reference number for the request is READ\_STORAGE\_REQUEST\_CODE, which refers to a value that we will define in a companion object at the top of the class. To define the reference number, add the following code above the binding variable:

```
companion object {
    const val READ_STORAGE_REQUEST_CODE = 1
}
```

The reference code will allow the MainActivity class to track and respond to the permissions request. If the user grants the relevant permissions when prompted, then we will run the getCallLogs method again. To handle the user's response to the permissions request, add the following code below the onCreate method:

```
override fun onRequestPermissionsResult(requestCode: Int, permissions: Array<out String>, grantResults:
IntArray) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults)
    when (requestCode) {
        READ_STORAGE_REQUEST_CODE -> if (grantResults.isNotEmpty() && grantResults[0] ==
PackageManager.PERMISSION_GRANTED) getCallLogs()
    }
}
```

The onRequestPermissionsResult method runs whenever the user responds to a permissions request. In the above code, we use a when block to assess whether the response request code matches the value of the READ\_STORAGE\_REQUEST\_CODE variable. If the values match, then the method knows the response refers to the request for storage read and call log read permissions. In which case, the when block checks whether the result of the request is PERMISSION\_GRANTED. If the permissions have been granted, then the getCallLogs method is run again. Otherwise, nothing happens and the user will be prompted to grant permission the next time they open the app.

Let's now return to the `getCallLogs` method and write the code that retrieves the device's call log history once the relevant user permissions have been approved. Replace the `TODO` comment with the following code:

```
val cursor = application.contentResolver.query(CallLog.Calls.CONTENT_URI, null, null, null,
CallLog.Calls.DATE + " DESC")

val callLog = mutableListOf<CallLogEvent>()

cursor?.use {
    val number = it.getColumnIndexOrThrow(CallLog.Calls.NUMBER)
    val type = it.getColumnIndexOrThrow(CallLog.Calls.TYPE)
    val date = it.getColumnIndexOrThrow(CallLog.Calls.DATE)
    while (it.moveToNext()) {
        val phoneNumber = cursor.getString(number)
        val callType = cursor.getString(type)
        val callDate = cursor.getLong(date)
        val callDateString = SimpleDateFormat("dd/MM/yyyy HH:mm", Locale.getDefault()).format(Date(callDate))
        val direction = when (callType.toInt()) {
            CallLog.Calls.OUTGOING_TYPE -> "OUTGOING"
            CallLog.Calls.INCOMING_TYPE -> "INCOMING"
            CallLog.Calls.MISSED_TYPE -> "MISSED"
            else -> null
        }
        val entry = CallLogEvent(direction, phoneNumber, callDateString)
        callLog.add(entry)
    }
}

cursor?.close()

communicationViewModel.callLog.value = callLog
```

Note you may need to add the following import statement to the top of the file:

```
import java.text.SimpleDateFormat
```

The above code queries the user's device for calls in the call log and transfers the results to a table using the `Cursor` interface. The `sortOrder` parameter of the query is programmed to sort the results by date in descending order. Sorting the results in this way means the call log will be reported from most recent to oldest. Next, a variable called `callLog` is defined, which will contain a list of `CallLogEvent` objects generated based on the results in the `Cursor` instance. First, the number, type and date columns from the `Cursor` are stored in variables. Next, each row in the `Cursor` is iterated over using the `moveToNext` method and the value of each column is extracted. The value under the date column undergoes further processing and is converted to a timestamp in the format `dd/MM/yyyy HH:mm`. Finally, the call direction, phone number associated with the call event and timestamp of the call are packaged in a `CallLogEvent` object and added to the `callLog` mutable list.

Once all the rows in the `Cursor` table have been converted to `CallLogEvent` objects, the `Cursor` is closed and the `callLog` list is assigned to the `CommunicationViewModel` view model's `callLog` variable. And with that, the complete list of `CallLogEvent` objects is now accessible to the `CallLogFragment`.

## Displaying the call log

To display the call history to the user, we need to return to the `CallLogAdapter` class (**Project > app > java > name of the project > ui > callLog**) and process the list of `CallLogEvent` objects. First, add the following variable to the top of the class above the `CallLogViewHolder` inner class:

```
var callLog = listOf<CallLogEvent>()
```

The `callLog` variable will store the list of `CallLogEvent` objects and allow its contents to be used elsewhere in the adapter. Next, add the following code below the `onCreateViewHolder` method:

```
override fun onBindViewHolder(holder: CallLogViewHolder, position: Int) {
    val current = callLog[position]

    val callDirection = holder.mDirection
    callDirection.visibility = View.VISIBLE
```

```

callDirection.setColorFilter(ContextCompat.getColor(activity, android.R.color.holo_green_dark))
when (current.direction) {
    "OUTGOING" -> callDirection.setImageResource(R.drawable.ic_outgoing)
    "INCOMING" -> callDirection.setImageResource(R.drawable.ic_incoming)
    "MISSED" -> {
        callDirection.setImageResource(R.drawable.ic_missed)
        callDirection.setColorFilter(ContextCompat.getColor(activity, android.R.color.holo_red_dark))
    }
    null -> callDirection.visibility = View.INVISIBLE
}
}

```

```

holder.mPhoneNumber.text = current.number
holder.mCallDate.text = current.date

```

```

holder.mCallback.setOnClickListener {
    if (current.number.isNotBlank()) activity.callNumber(current.number)
}
}

```

The `onBindViewHolder` method is another mandatory method for RecyclerView adapters. It helps manage the data for each item in the RecyclerView and handle user interactions. In the above code, the `CallLogEvent` object associated with the current position in the RecyclerView is retrieved from the `callLog` variable and used to populate the widgets in the `call_log_entry` layout.

First, the `callDirection` `ImageView` widget's visibility property is set to visible and a green tint is applied. A `when` block is then used to apply call direction-specific properties to the widget. For example, each call direction is associated with a different image, as shown below. Also, if the call direction is set to 'MISSED', the tint property is changed from green to red to emphasise the missed call. Finally, in the unlikely event that the call direction is null, then the `ImageView` widget will be hidden.

Outgoing      Incoming      Missed



Next, the phone number and the timestamp associated with the event are loaded into the number and date `TextView` widgets, respectively. Finally, an `onClick` listener is applied to the phone icon `ImageButton` widget. If the user clicks the button, then the phone number associated with the call event is sent to the `MainActivity` class's `callNumber` method, which will attempt to initiate a phone call.

There is one more mandatory method we need to add to the adapter and it is called `getItemCount`. The `getItemCount` method will calculate how many items are loaded into the adapter. In this instance, the item count will equal the number of objects in the `callLog` list, so define the `getItemCount` method by adding the following code below the `onBindViewHolder` method:

```

override fun getItemCount() = callLog.size

```

All the mandatory adapter methods are now in place. Let's return to the `CallLogFragment` class and write the code that retrieves the list of `CallLogEvent` objects from the view model and loads them into the adapter. Open the `CallLogFragment.kt` file and add the following variable to the list of variables at the top of the class:

```

private val communicationViewModel: CommunicationViewModel by activityViewModels()

```

Note you may need to add the following import statement to the top of the file:

```

import androidx.fragment.app.activityViewModels

```

The fragment can use this `communicationViewModel` variable to interact with the `CommunicationViewModel` class and access its data. To process the list of `CallLogEvent` objects held by the view model, add the following code below the `onCreateView` method:

```

@SuppressLint("NotifyDataSetChanged")
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {

```

```

super.onViewCreated(view, savedInstanceState)

binding.root.layoutManager = LinearLayoutManager(activity)
binding.root.adapter = callLogAdapter

callingActivity.getCallLogs()

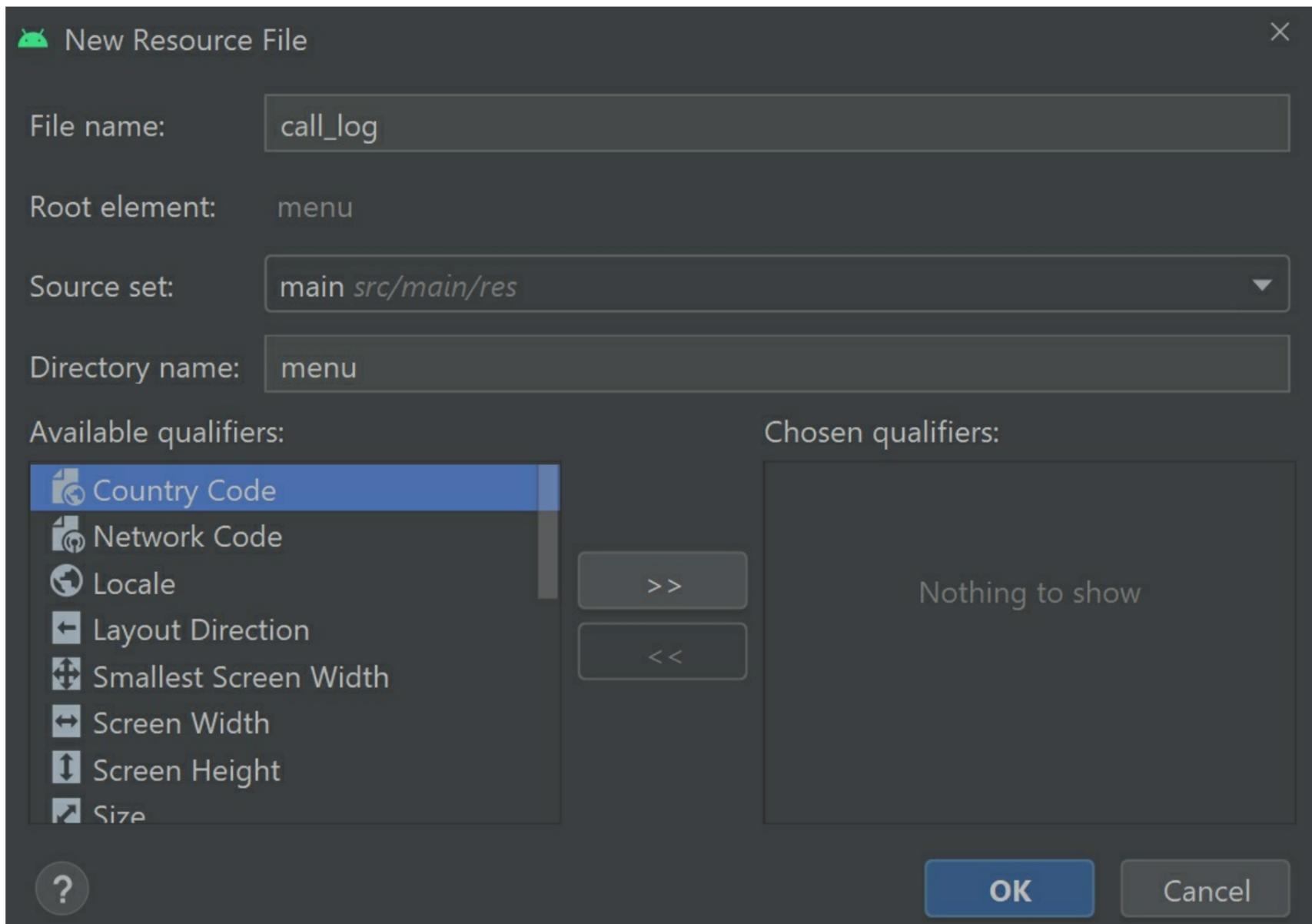
communicationViewModel.callLog.observe(viewLifecycleOwner, { log ->
    log?.let {
        // take most recent 10 calls
        callLogAdapter.callLog = it.take(10)
        callLogAdapter.notifyDataSetChanged()
    }
})
}

```

The above code applies a linear layout manager and the CallLogAdapter class to the RecyclerView widget from the fragment\_call\_log layout. The adapter and layout manager will work together to display the list of call log events vertically inside the RecyclerView. Next, the MainActivity class's getCallLogs method is used to retrieve the call log history from the device. Once the call log is available, the fragment will be notified via an observer that is registered to the CommunicationViewModel view model's callLog variable. The observer is notified whenever the underlying data changes. When the call log is updated, the above code sends the first ten items from the call log to the CallLogAdapter class's callLog variable. The adapter's inbuilt notifyDataSetChanged method is then used to refresh the RecyclerView widget's contents and display the new list of call log events. And with that, the fragment and adapter can now work together to retrieve the device's call log history and display the ten most recent events to the user.

## Handling user interactions with call log events

The user will be able to interact with call log events via an options menu. The options menu will allow the user to call or send an SMS message to the phone number associated with the event. The options menu will open as a popup window whenever the user long presses an item in the call log RecyclerView widget. To define the contents of the popup menu, locate the **menu** directory (**Project > app > res**) then select **New > Menu Resource File**. Name the file call\_log then click OK.



Open the **call\_log.xml** resource file in Code view and add the following two items inside the menu element:

```
<item android:id="@+id/make_call"
    android:title="@string/make_call" />
<item android:id="@+id/send_sms"
    android:title="@string/send_sms" />
```

Each menu item contains an ID, which we can use to refer to the menu item, and a title, which displays text to the user. In this case, the two menu items will invite the user to call or send an SMS message to the associated phone number, respectively. To make the popup menu operational, open the **MainActivity.kt** file (**Project > app > java > name of the project**) and add the following method below the `getCallLogs` method:

```
fun showCallLogPopup(view: View, phoneNumber: String) {
    PopupMenu(this, view).apply {
        inflate(R.menu.call_log)
        setOnMenuItemClickListener {
            when (it.itemId) {
                R.id.make_call -> {
                    callNumber(phoneNumber)
                    true
                }
                R.id.send_sms -> {
                    openDialog(SendSMS(phoneNumber))
                    true
                }
            }
            else -> super.onOptionsItemSelected(it)
        }
    }
    show()
}
```

Note you may need to add the following import statement to the top of the file:

```
import android.widget.PopupMenu
```

The showCallLogPopup method receives two arguments: the layout View that the popup menu will appear over (i.e. the RecyclerView item that the user has long pressed) and the phone number associated with the selected call log event. The showCallLogPopup method then uses an instance of the PopupMenu class to inflate the **call\_log.xml** menu resource file and applies an action to each menu item. If the make\_call menu item is clicked, then the selected phone number will be sent to the callNumber method, which will initiate the phone call. Meanwhile, the send\_sms menu item will open a dialog window called SendSMS. We will discuss the mechanics of the SendSMS dialog window in a later section. In brief, it will allow the user to type and send an SMS message. Once the menu is built, the PopupMenu class's show method will display the menu to the user.

As mentioned above, the application will use dialog windows to allow the user to view and reply to SMS messages. The dialog windows will be inflated (displayed) using a method called showDialog. To define the showDialog method, add the following code below the showCallLogPopup method:

```
fun showDialog(dialog: DialogFragment) = dialog.show(supportFragmentManager, "")
```

Note you may need to add the following import statement to the top of the file:

```
import androidx.fragment.app.DialogFragment
```

The showDialog method uses the DialogFragment class's show method and the activity's support fragment manager to load the fragment and display it to the user. This method will come in handy later as we proceed with the project.

The popup menu is now operational. To load the popup menu when a user long presses an item in the call log fragment's RecyclerView widget, open the **CallLogAdapter.kt** file (**Project > app > java > name of the project > ui > callLog**) and replace the TODO comment in the CallLogViewHolder inner class with the following code:

```
val phoneNumber = callLog[adapterPosition].number
if (phoneNumber.isNotBlank()) activity.showCallLogPopup(it, phoneNumber)
```

The above code retrieves the phone number associated with the user's selected call log event and checks whether the phone number is blank. If the phone number is not blank, then the MainActivity class's showSMSPopup method will load the popup menu so the user can call or message the phone number.

## Designing the SMS fragment layout

The third and final fragment in the app will display a list of recently received SMS messages. It will also feature a floating action button that the user can click to send new SMS messages. The SMS fragment will require a layout, so right-click the **layout** directory (**Project > app > res**) then select **New > Layout Resource File**. Name the layout **fragment\_sms** then press OK. Once the layout opens in the editor, switch to Code view and edit the file so it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/callLog"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <com.google.android.material.floatingactionbutton.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="16dp"
        android:contentDescription="@string/send_sms"
        app:srcCompat="@drawable/ic_new_sms"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

The root element of the fragment\_sms layout is a ConstraintLayout widget. Inside the ConstraintLayout, there is a

RecyclerView widget and a FloatingActionButton widget. The floating action button is constrained to the bottom right corner of the layout. The image source for the button is an SMS icon to indicate to the user that pressing the button will allow them to write and send an SMS message.

The RecyclerView widget will display a list of recent messages. The messages inside the RecyclerView will require a layout to display details including the phone number of the sender and a preview of the message body. To achieve this, create a new layout in the usual way and name it `sms_entry`. Once the `sms_entry.xml` layout file is open in the editor, switch the layout to Code view and edit the file so it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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="wrap_content"
    android:padding="16dp"
    android:background="?attr/selectableItemBackground">

    <TextView
        android:id="@+id/sender"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:singleLine="true"
        android:textColor="@color/material_on_surface_emphasis_high_type"
        android:textSize="16sp"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/body"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="8dp"
        android:maxLines="3"
        android:textSize="14sp"
        android:textColor="@color/material_on_surface_emphasis_medium"
        app:layout_constraintTop_toBottomOf="@id/sender" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

The above layout comprises a ConstraintLayout widget and two TextView widgets. The first TextView widget will display the phone number that sent the SMS message, while the second TextView widget will display up to three lines of the message body.

## Creating the SMS fragment and SMS adapter

Similar to the steps for creating the phone and call log fragments, we will repurpose one of the packages that were autogenerated by Android Studio and use it to handle SMS messages. Right-click the **dashboard** directory (**Project** > **app** > **java** > **name of the project** > **ui**) then select **Refactor** > **Rename**. Set the new name to `sms`. Next, explore the `sms` directory, delete the `DashboardViewModel` class and refactor the `DashboardFragment` class to `SMSFragment`. Open the newly refactored `SMSFragment` class in the editor and replace the `dashboardViewModel` and `_binding` variables with the following code:

```
private var _binding: FragmentSmsBinding? = null
private val communicationViewModel: CommunicationViewModel by activityViewModels()
private lateinit var callingActivity: MainActivity
private lateinit var smsAdapter: SMSAdapter
```

Note you may need to add the following import statement to the top of the file:

```
import androidx.fragment.app.activityViewModels
```

In the above code, the binding variable will store an instance of the `fragment_sms` layout's view binding class, which the fragment will use to access the different components of the layout; the `communicationViewModel` variable will provide access to the data stored in the `CommunicationViewModel` view model; the `callingActivity` variable will allow the fragment to use the `MainActivity` class's public methods and variables; and the `smsAdapter` variable will store an instance of a RecyclerView adapter class called `SMSAdapter` that we will create shortly. To

initialise the binding and callingActivity variables, edit the fragment's onCreateView method so it reads as follows:

```
override fun onCreateView(  
    inflater: LayoutInflater,  
    container: ViewGroup?,  
    savedInstanceState: Bundle?  
): View {  
    _binding = FragmentSmsBinding.inflate(inflater, container, false)  
    callingActivity = activity as MainActivity  
  
    // TODO: Initialise the smsAdapter variable here  
  
    return binding.root  
}
```

The binding and callingActivity variables have now been initialised and can be used elsewhere in the fragment. Moving on, let's create the adapter class that will handle the list of recently received SMS messages and load them into the RecyclerView widget from the fragment\_sms layout. Right-click the **sms** directory then select **New > Kotlin Class/File**. Name the file SMSAdapter and select Class from the list of options. Next, open the SMSAdapter class in the editor and modify its code so it reads as follows:

```
class SMSAdapter(private val activity: MainActivity) : RecyclerView.Adapter<SMSAdapter.SMSViewHolder> {  
  
    // TODO: Define the SMSViewHolder inner class here  
  
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): SMSAdapter.SMSViewHolder {  
        return SMSViewHolder(LayoutInflater.from(parent.context).inflate(R.layout.sms_entry, parent, false))  
    }  
}
```

The above code begins defining the mandatory methods for RecyclerView adapters, starting with the onCreateViewHolder method. The onCreateViewHolder method states that each item in the RecyclerView will be managed by a view holder class called SMSViewHolder and the view holder will use the **call\_log\_entry.xml** layout file. To define the SMSViewHolder inner class, replace the TODO comment at the top of the adapter with the following code:

```
inner class SMSViewHolder(itemView: View) :  
    RecyclerView.ViewHolder(itemView),  
    View.OnClickListener {  
  
    internal var mSender = itemView.findViewById<View>(R.id.sender) as TextView  
    internal var mBody = itemView.findViewById<View>(R.id.body) as TextView  
  
    init {  
        itemView.isClickable = true  
        itemView.setOnClickListener(this)  
        itemView.setOnLongClickListener {  
            // TODO: Open the SMS options dialog here  
            return@setOnLongClickListener true  
        }  
    }  
  
    override fun onClick(view: View) = activity.openDialog(ViewSMS(texts[adapterPosition]))  
}
```

The SMSViewHolder class creates variables for each TextView widget in the sms\_entry layout. The adapter can use the variables to access the TextView widgets and update their contents. An init block is also defined, which will handle user interactions. In this case, an onLongClick listener is applied to the entire view holder. Once implemented, the onLongClick listener will open a popup menu that allows the user to view and reply to the SMS message. The SMSViewHolder class also registers a regular onClick listener. If the user clicks the view holder, then the MainActivity class's openDialog method will load a dialog fragment called ViewSMS and display the full SMS message.

To implement the adapter, return to the **SMSFragment.kt** file and replace the TODO comment in the onCreateView method with the following code:

```
smsAdapter = SMSAdapter(callingActivity)
```

The SMSFragment class can now interact with the adapter.

## Retrieving the device's recent SMS messages

Information about an SMS message will be packaged in a Kotlin data class. The data class will act as a template and contain fields for the different bits of information such as the message sender and message body. Create a new Kotlin class in the usual way, navigate through **Project > app > java** then right-click the folder with the name of the project and select **New > Kotlin File/Class**. Name the file SMS and select Data Class from the list of options. A file called **SMS.kt** should then open in the editor. Modify the class so reads as follows:

```
data class SMS(  
    val sender: String,  
    val body: String)
```

The SMS data class will store information about a given SMS message on the user's device. The primary constructor of the data class contains two variables, each storing a different bit of information about the message. First, the sender variable will store the phone number that sent the message and the body variable will store the message's contents. Both variables will store data in string format and are initialised using the val keyword. The val keyword means the value of a variable is fixed and cannot be changed once set. It is used here because there will not be any occasion where we need to change the contents or sender of an existing SMS message.

When the user navigates to the SMS fragment, the fragment will request that the MainActivity class retrieve the 20 most recent SMS messages. The SMS messages will be converted to SMS data class objects and sent to the CommunicationViewModel view model. The fragment will monitor the list of SMS objects and use the adapter to display them to the user. To arrange this, we should first prepare the CommunicationViewModel view model to store the list of SMS objects. Locate and open the **CommunicationViewModel.kt** file (**Project > app > java > name of the project**) and add the following code below the callLog variable:

```
var texts = MutableLiveData<List<SMS>>()
```

The texts variable will store a MutableLiveData list of SMS objects. The SMSFragment class will observe the list and update the user interface whenever the list changes, such as when a new SMS message arrives.

Moving on, let's return to the MainActivity class and write the code that retrieves the SMS message history from the user's device. Open the **MainActivity.kt** file (**Project > app > java > name of the project**) and add the following method below the showCallLogPopup method to request the most recently received SMS messages:

```
fun get texts() {  
    if (ContextCompat.checkSelfPermission(this, Manifest.permission.READ_SMS)  
        != PackageManager.PERMISSION_GRANTED) {  
        ActivityCompat.requestPermissions(this, arrayOf(Manifest.permission.READ_SMS),  
            READ_SMS_REQUEST_CODE)  
    }  
    return  
}  
  
// TODO: Retrieve SMS messages here  
}
```

The get texts method begins by checking whether the user has granted permission for the app to read the SMS messages on the device. If permission has been granted, then the method will continue as normal; however, if permission has not been granted then a dialog window will appear requesting the user grant the relevant permission. The reference number for the permission request is READ\_SMS\_REQUEST\_CODE, which refers to a value that will be stored in the companion object at the top of the class. Add the following code below the READ\_STORAGE\_REQUEST\_CODE variable:

```
const val READ_SMS_REQUEST_CODE = 2
```

This reference code will allow the MainActivity class to track and respond to the permissions request. If the user grants the relevant permissions when prompted, then we will need to run the get texts method again. To handle the user's response to the permissions request, edit the when block in the onRequestPermissionsResult method so it reads as follows:

```
when (requestCode) {  
    READ_STORAGE_REQUEST_CODE -> if (grantResults.isNotEmpty() && grantResults[0] ==
```

```

PackageManager.PERMISSION_GRANTED) getCallLogs()
    READ_SMS_REQUEST_CODE -> if (grantResults.isNotEmpty() && grantResults[0] ==
PackageManager.PERMISSION_GRANTED) getTextures()
}

```

The above code expands the `onRequestPermissionsResult` method so that as well as responding to the permissions requests from the `getCallLogs` method, the method can also respond to the permissions request from the `getTexts` method. If the user grants permission for the app to read the SMS messages on the device, then the `getTexts` method is run again. Otherwise, nothing happens and the user will be prompted to grant permission again the next time they open the app.

Let's now return to the `getTexts` method and write the code that retrieves the device's recent SMS messages. Replace the `TODO` comment with the following code:

```

val cursor = application.contentResolver.query(Uri.parse("content://sms/inbox"), null, null, null, "date DESC")

val texts = mutableListOf<SMS>()

cursor?.use {
    val senderColumn = it.getColumnIndexOrThrow(Telephony.TextBasedSmsColumns.ADDRESS)
    val bodyColumn = it.getColumnIndexOrThrow(Telephony.TextBasedSmsColumns.BODY)

    while (it.moveToNext()) {
        val sender = it.getString(senderColumn) ?: getString(R.string.error_sender)
        val body = it.getString(bodyColumn) ?: getString(R.string.error_body)
        val sms = SMS(sender, body)
        texts.add(sms)
    }
}

cursor?.close()

communicationViewModel.texts.value = texts.take(20)

```

The above code queries the user's device for messages in the SMS inbox and transfers the results to a table using the `Cursor` interface. A `sortOrder` property of 'date DESC' is applied to the query, which means results will be sorted from newest to oldest. Next, the table of the results is processed and used to generate a list of SMS objects. First, the sender address (i.e. phone number) and body columns are isolated from the `Cursor`. Next, each row in the `Cursor` is iterated over using the `moveToNext` command and the value for each targeted column is retrieved. Elvis operators `?:` are used to define alternative values if a given field is null. For example, if the sender's address cannot be found for a given message, then the value on the right-hand side of the Elvis operator (i.e. the string "Could not load message sender") will be used instead. Finally, the sender's phone number and the message body are packaged in an SMS object and added to a list variable called `texts`. Once all the rows in the `Cursor` have been converted to SMS objects, the `Cursor` is closed and the first 20 items from the `texts` list are assigned to the `CommunicationViewModel` view model's `texts` variable.

## Displaying the SMS messages

To display the user's recently received SMS messages, we need to return to the `SMSAdapter` class (**Project > app > java > name of the project > ui > sms**) and configure it to support the list of SMS objects. First, add the following variable to the top of the class above the `SMSViewHolder` inner class:

```

var texts = listOf<SMS>()

```

The `texts` variable will store the list of SMS objects. To load the SMS objects into the `RecyclerView`, add the following code below the `onCreateViewHolder` method:

```

override fun onBindViewHolder(holder: SMSViewHolder, position: Int) {
    val current = texts[position]

    holder.mSender.text = current.sender
    holder.mBody.text = current.body
}

```

The `onBindViewHolder` method will manage the data that is displayed at each position in the `RecyclerView`. The SMS object associated with the current position is retrieved from the `texts` list and its values are used to populate the

sender and body TextView widgets from the sms\_entry layout. In the sms\_entry layout, the body TextView widget has a maxLines attribute set to 3. This means if the body string is too long then it will be cut off after three lines and only a preview will be displayed. The user can view the full message by clicking the RecyclerView item.

There is one more mandatory method that we need to add to the adapter and it is called getItemCount. The getItemCount method will calculate how many items are loaded into the RecyclerView. In this instance, the item count will equal the number of objects in the texts list, so define the getItemCount method by adding the following code below the onBindViewHolder method:

```
override fun getItemCount() = texts.size
```

All the mandatory adapter methods are now in place. Let's return to the SMSFragment class and write the code that retrieves the list of SMS objects from the CommunicationViewModel view model and loads them into the adapter. Open the **SMSFragment.kt** file and add the following code below the onCreateView method:

```
@SuppressWarnings("NotifyDataSetChanged")
override fun onCreateView(view: View, savedInstanceState: Bundle?) {
    super.onCreateView(view, savedInstanceState)

    binding.callLog.layoutManager = LinearLayoutManager(activity)
    binding.callLog.adapter = smsAdapter

    callingActivity.getTexts()

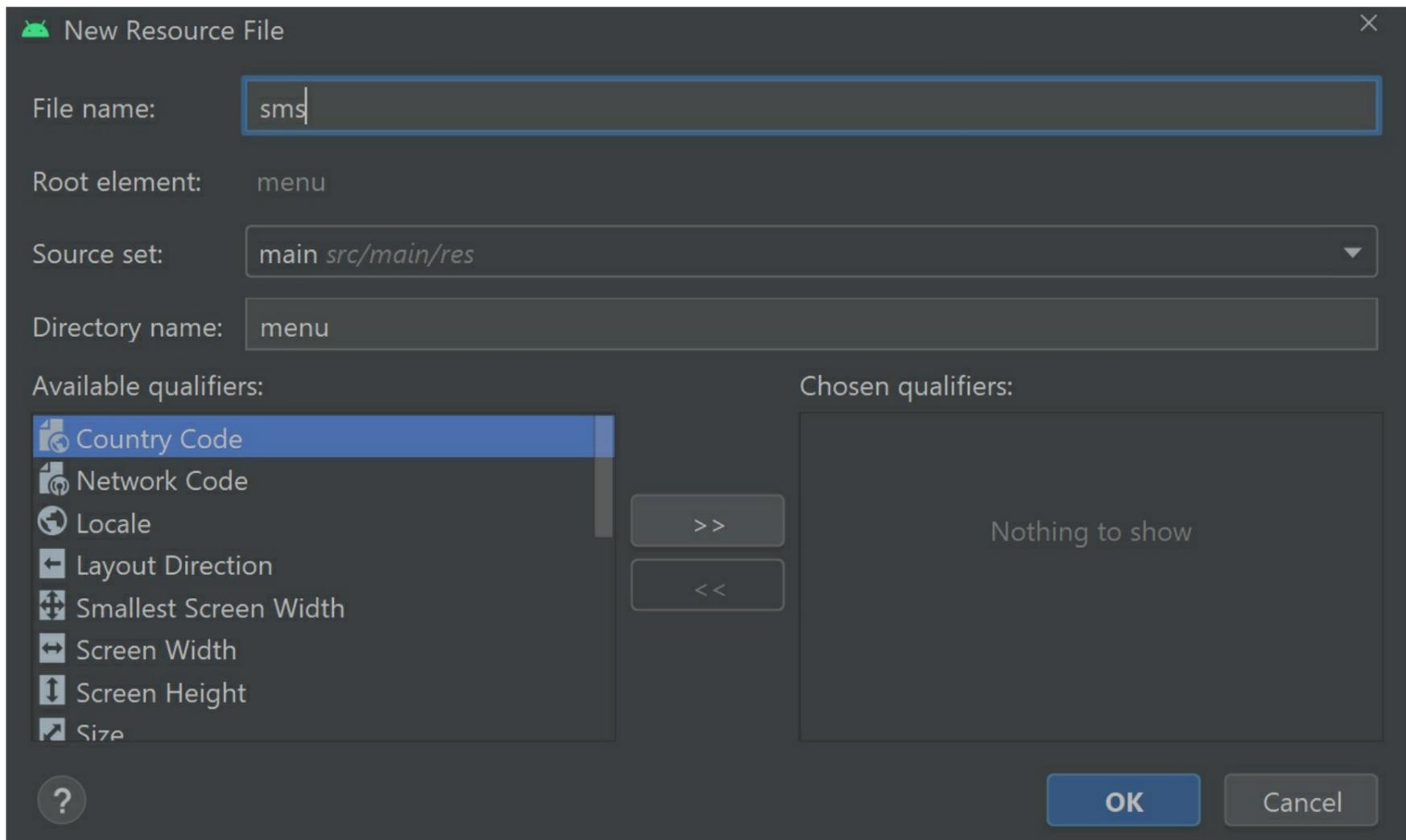
    communicationViewModel.texts.observe(viewLifecycleOwner, { texts ->
        texts?.let {
            smsAdapter.texts = it
            smsAdapter.notifyDataSetChanged()
        }
    })

    // TODO: Assign an action to the floating action button here
}
```

The onCreateView method directs the RecyclerView widget from the fragment\_sms layout to use a linear layout manager and the SMSAdapter adapter class. Next, the MainActivity class's getTexts method is run to retrieve the recently received SMS messages from the device. Once the list of SMS messages is available, the fragment will be notified via an observer it registers to the view model's texts variable. Each time the list of SMS messages is updated, the observer sends the list to the SMSAdapter adapter's texts variable. The adapter's notifyDataSetChanged method is used to refresh the RecyclerView widget with the new list of SMS objects and display them to the user.

## Handling user interactions with SMS messages

If the user long-presses an item in the SMS fragment's RecyclerView widget, a popup menu will invite the user to view or reply to the SMS message. To define the contents of the popup menu, locate the **menu** directory (**Project > app > res**) then select **New > Menu Resource File**. Name the file sms then press OK.



Open the **sms.xml** resource file in Code view and add the following two items inside the menu element:

```
<item android:id="@+id/view_message"
    android:title="@string/view_message" />
<item android:id="@+id/reply"
    android:title="@string/reply" />
```

Each menu item contains an ID, which we can use to refer to the menu item, and a title, which displays text to the user. In this case, the two menu items will invite the user to view or reply to the SMS message, respectively. To make the popup menu operational, open the **MainActivity.kt** file (**Project > app > java > name of the project**) and add the following method below the `getTexts` method:

```
fun showSMSPopup(view: View, text: SMS) {
    PopupMenu(this, view).apply {
        inflate(R.menu.sms)
        setOnMenuItemClickListener {
            when (it.itemId) {
                R.id.view_message -> {
                    openDialog(ViewSMS(text))
                    true
                }
                R.id.reply -> {
                    openDialog(SendSMS(text.sender))
                    true
                }
                else -> super.onOptionsItemSelected(it)
            }
        }
    }
    show()
}
```

The `showSMSPopup` method receives two arguments: the layout `View` that the popup menu will appear over (i.e. the `RecyclerView` item that the user has long pressed) and the `SMS` object associated with the selected message preview. The `showSMSPopup` menu then uses an instance of the `PopupMenu` class to inflate the **sms.xml** menu resource file and apply an action to each menu item. If the `view_message` menu item is clicked, then the full SMS message will be displayed by sending the full `SMS` object to a dialog fragment called `ViewSMS` that we will create

shortly. Meanwhile, if the reply menu item is clicked, then a dialog fragment called SendSMS will open and allow the user to reply to the SMS message. Once the menu is built, the PopupMenu class's show method will display the menu to the user.

To load the popup menu when a user long presses an item in the SMS fragment's RecyclerView, open the **SMSAdapter.kt** file (**Project > app > java > name of the project > ui > sms**) and replace the TODO comment in the SMSViewHolder inner class with the following code:

```
activity.showSMSPopup(it, texts[adapterPosition])
```

Now, if the user long presses an item in the RecyclerView, the adapter will run MainActivity's showSMSPopup method and load the popup menu so the user can view or reply to the message.

## Viewing full SMS messages

If the user wishes to view an SMS message then they can either click the message preview or select the view\_message item from the options popup menu. To display the full SMS message, we need to create a new layout file. Locate and right-click the **layout** directory (**Project > app > res**) then select **New > Layout Resource File**. Name the layout view\_sms then press OK. Once the **view\_sms.xml** layout opens in the editor, switch the file to Code view and edit its contents as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center">
    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="16dp">
        <TextView
            android:id="@+id/sender"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:singleLine="true"
            android:textColor="@color/material_on_surface_emphasis_high_type"
            android:textSize="18sp" />
        <TextView
            android:id="@+id/body"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginTop="8dp"
            android:justificationMode="inter_word"
            android:textSize="16sp"
            android:textColor="@color/material_on_surface_emphasis_medium"
            android:layout_below="@id/sender" />
        <Button
            android:id="@+id/replyBtn"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="8dp"
            android:text="@string/reply"
            android:layout_below="@id/body"
            android:layout_alignParentEnd="true"
            style="?android:attr/buttonBarButtonStyle"/>
        <Button
            android:id="@+id/cancelBtn"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
```

```

        android:layout_marginTop="8dp"
        android:layout_marginEnd="16dp"
        android:text="@string/cancel"
        android:layout_below="@id/body"
        android:layout_toStartOf="@id/replyBtn"
        style="?android:attr/buttonBarButtonStyle"/>
    </RelativeLayout>
</ScrollView>

```

The root element of the `view_sms` layout is a `ScrollView` widget. `ScrollView` widgets allow the user to scroll down if the layout's content is too large to fit on the screen. The `ScrollView` is useful here because the user could receive an SMS message that is too long to fit on the screen and requires scrolling. `ScrollView` widgets may only contain one direct child element, and so it is this child element that will coordinate the other widgets in the layout. In this case, we will use a `RelativeLayout` widget. The `RelativeLayout` widget defined above contains two `TextView` widgets, which will display the phone number that sent the SMS message and the body of the message, and two `Button` widgets. The buttons will allow the user to reply to the message or dismiss it, respectively. The `TextView` widget that displays the message body has a `justificationMode` attribute set to `inter_word`. Setting the justification mode attribute to `inter_word` means that the space between words will stretch so that the left and right edges of each line align. Using this justification will help the message look neat.

Let's now create the class that will make the `view_sms` layout operational. Right-click the `sms` folder in the `ui` directory and select **New > Kotlin Class/File**. Name the file `ViewSMS` and select `Class` from the list of options. Once the `ViewSMS.kt` file is open in the editor, modify its code so it reads as follows:

```

import android.app.AlertDialog
import androidx.fragment.app.DialogFragment

class ViewSMS(private val sms: SMS) : DialogFragment() {

    private var _binding: ViewSmsBinding? = null
    private val binding get() = _binding!!

    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
        val callingActivity = activity as MainActivity
        val inflater = callingActivity.layoutInflater
        _binding = ViewSmsBinding.inflate(inflater)

        val builder = AlertDialog.Builder(callingActivity)
            .setView(binding.root)

        binding.sender.text = sms.sender
        binding.body.text = sms.body

        binding.cancelBtn.setOnClickListener {
            dismiss()
        }

        binding.replyBtn.setOnClickListener {
            callingActivity.openDialog(SendSMS(sms.sender))
            dismiss()
        }

        return builder.create()
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}

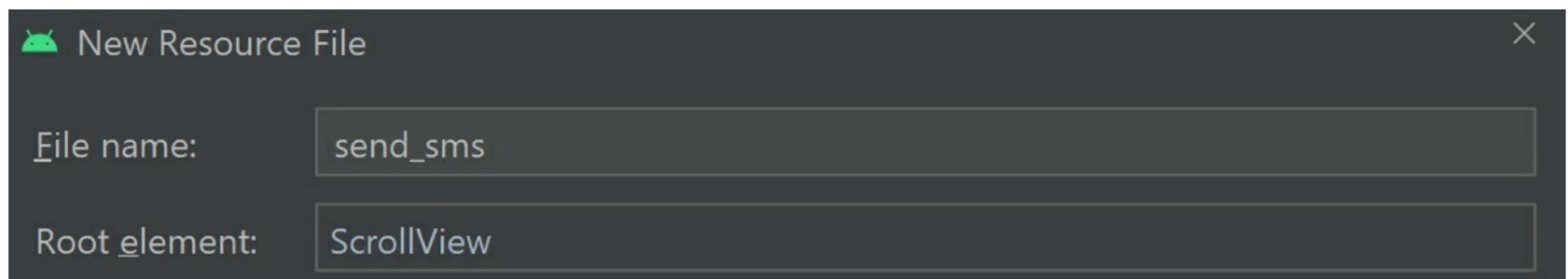
```

The `ViewSMS` class defined above has a parameter called `sms` in its primary constructor, which will store the user's selected SMS object. Next, the class's `onCreateDialog` method prepares the dialog window by initialising the `view_sms` layout's binding class and setting the root element of the layout as the main view for the dialog window. The remainder of the method assigns values and actions to each widget inside the `view_sms` layout. First, the sender

and body TextView widgets will display the phone number that sent the SMS message and the message's contents, respectively. Next, the cancel button is configured to dismiss the dialog window. The reply button will also dismiss the dialog window but not before opening another dialog fragment called SendSMS. The SendSMS dialog fragment will allow the user to write a reply to the SMS message they are currently viewing. Once all the options menu items have been populated with text and any necessary actions have been defined, the DialogFragment class's create method will compile the dialog window and display it to the user.

## Sending an SMS message

The app will allow the user to write and send SMS messages. New SMS messages will be drafted using a dialog fragment, and as always, the dialog fragment will require a user interface layout. Right-click the **layout** directory and select **New > Layout Resource File**. Name the layout send\_sms, enter ScrollView as the root element then click OK. The ScrollView widget will allow the user to scroll down the layout if the message they write is too large to fit in the screen.



Once the layout is open, add the following code inside the ScrollView element:

```
<RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="8dp" >

    <EditText
        android:id="@+id/number"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:inputType="number"
        android:singleLine="true"
        android:hint="@string/number"
        android:importantForAutofill="no" />

    <EditText
        android:id="@+id/body"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginVertical="8dp"
        android:hint="@string/message"
        android:importantForAutofill="no"
        android:inputType="text"
        android:layout_below="@+id/number" />

    <Button
        android:id="@+id/sendBtn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/body"
        android:layout_alignParentEnd="true"
        android:text="@string/send"
        style="?android:attr/buttonBarButtonStyle" />

    <Button
        android:id="@+id/cancelBtn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginEnd="12dp"
```

```

        android:layout_below="@id/body"
        android:layout_toStartOf="@id/sendBtn"
        android:text="@string/cancel"
        style="?android:attr/buttonBarButtonStyle" />
</RelativeLayout>

```

The above code adds a RelativeLayout widget inside the ScrollView widget to coordinate the other widgets in the layout. The first two widgets inside the RelativeLayout are EditText widgets that will allow the user to enter the phone number they would like to message and the body of the message, respectively. If the user is replying to an existing message then the value of the phone number EditText widget will be automatically populated. There are also two button widgets at the bottom of the layout. The first button will send the SMS message, while the second button will dismiss the dialog without sending the message.

Once the layout is in place, we can create the SendSMS dialog fragment class that will handle user interactions. Locate and right-click the **sms** directory (**Project** > **app** > **java** > **name of the project** > **ui**) then select **New** > **Kotlin Class/File**. Name the file SendSMS then select Class from the list of options. Once the **SendSMS.kt** file opens in the editor, modify its code so it reads as follows:

```

import android.app.AlertDialog
import androidx.fragment.app.DialogFragment

class SendSMS(private val number: String?) : DialogFragment() {
    private var _binding: SendSmsBinding? = null
    private val binding get() = _binding!!

    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
        val callingActivity = activity as MainActivity
        val inflater = callingActivity.layoutInflater
        _binding = SendSmsBinding.inflate(inflater)

        val builder = AlertDialog.Builder(callingActivity)
            .setView(binding.root)

        if (number != null) {
            val editable: Editable = SpannableStringBuilder(number)
            binding.number.text = editable
        }

        binding.cancelBtn.setOnClickListener {
            dismiss()
        }

        binding.sendBtn.setOnClickListener {
            if (callingActivity.sendSMS(binding.number.text.toString(), binding.body.text.toString())) dismiss()
        }

        return builder.create()
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}

```

The SendSMS class has a parameter called number in its primary constructor. If the user is replying to a previously received message then the value of the number parameter will be the phone number the user is replying to. Otherwise, the value of the number parameter will be null. If a phone number is supplied, then it is converted to an Editable before being assigned to the number EditText widget. This step is necessary for the user to be able to modify the phone number if they wish.

Next, the onCreateDialog method sets the send\_sms layout as the main view for the dialog window. The method also assigns actions to each widget inside the layout. For example, the cancel button will dismiss the dialog window without sending the message. Meanwhile, the send button will run a MainActivity method called sendSMS, which will attempt to dispatch the SMS message. If the message is sent successfully, then the sendSMS method will return a value of true and the dialog fragment will dismiss itself. Otherwise, the dialog fragment will remain active and the

MainActivity class will display a toast notification advising the user that the message could not be sent. Once all the necessary actions have been defined, the AlertDialog class's create method will compile the dialog window and display it to the user.

Let's now define the sendSMS method. The sendSMS method will attempt to dispatch the user's drafted SMS message. Open the **MainActivity.kt** file and add the following method below the showSMSPopup method:

```
fun sendSMS(number: String, message: String): Boolean {
    if (number.isEmpty() || message.isEmpty()) {
        Toast.makeText(this, getString(R.string.error_sending_sms), Toast.LENGTH_LONG).show()
        return false
    }
    return if (ContextCompat.checkSelfPermission(this, Manifest.permission.SEND_SMS)
        == PackageManager.PERMISSION_GRANTED) {
        val smsManager = SmsManager.getDefault()
        // 160 characters is typically the maximum size per message
        if (message.length > 160) {
            val messages: ArrayList<String> = smsManager.divideMessage(message)
            smsManager.sendMultipartTextMessage(number, null, messages, null, null)
        } else smsManager.sendTextMessage(number, null, message, null, null)
        true
    } else {
        ActivityCompat.requestPermissions(this, arrayOf(Manifest.permission.SEND_SMS), 0)
        false
    }
}
```

Note you may need to add the following import statement to the top of the file:

```
import android.telephony.SmsManager
```

The sendSMS method requires the recipient's phone number and the SMS message body to be supplied as arguments. If the value of either argument is empty, then a toast notification will inform the user that they need to provide both bits of information and a value of false will be returned. On the other hand, if both pieces of information have been provided then the method will check whether the app has permission to send SMS messages. If permission has not been granted, then the user will be prompted to provide permission. Meanwhile, if permission has been granted, then the length of the message body is calculated to check whether the message exceeds 160 characters. If the message body is longer than 160 characters then the message must be split into fragments using the device's SMS manager. Each message fragment is then sent using the SMS manager's sendMultiPartTextMessage method. These steps are not necessary if the message body is shorter than 160 characters because the SMS message can be sent in full using the SMS manager's sendTextMessage method. Once the message has been sent successfully, the method will return a value of true.

The app can now send SMS messages to a phone number of the user's choosing. The only step that remains is to enable the floating action button in the SMS fragment and allow the user to draft a new message. To do this, locate and open the **SMSFragment.kt** file (**Project > app > java > name of the project > ui > sms**) then replace the TODO comment in the onCreateView method with the following code:

```
binding.fab.setOnClickListener {
    callingActivity.openDialog(SendSMS(null))
}
```

The floating action button will now run the openDialog method and load the SendSMS dialog fragment when clicked. A value of null is passed as the number parameter in the SendSMS class constructor, which means the dialog fragment will not auto-populate the recipient's phone number and the user will need to provide this information.

## Detecting newly received SMS messages

The last feature we will add to this app is a broadcast receiver that will detect whenever new SMS messages are received. The broadcast receiver will require a class to operate, so create a new Kotlin class in the same folder as MainActivity called SMSBroadcastReceiver and edit its code so it reads as follows:

```
import android.telephony.SmsMessage
```

```

class SMSBroadcastReceiver: BroadcastReceiver() {

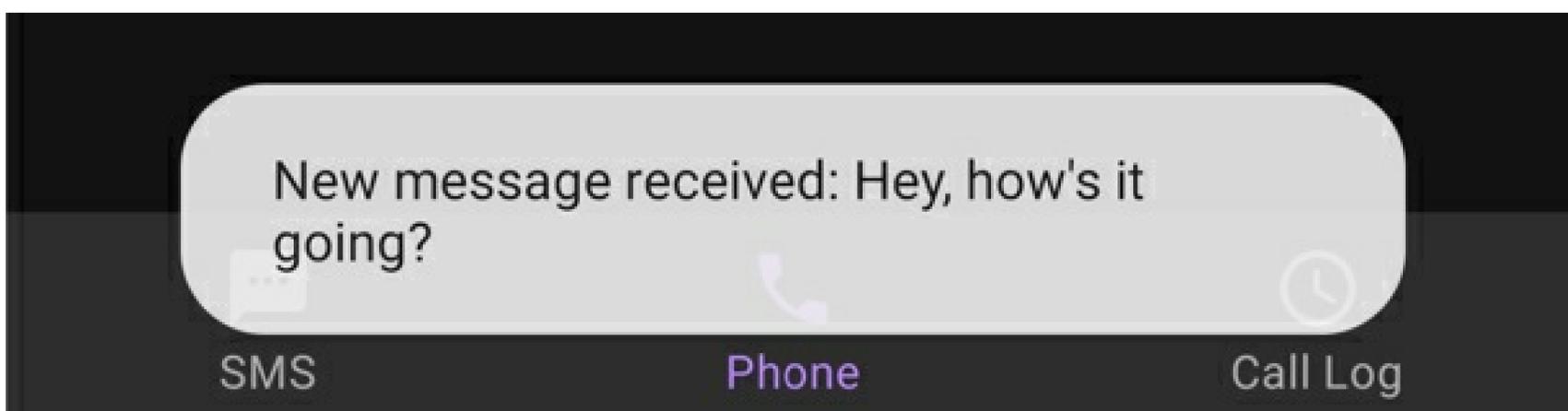
    companion object {
        const val SMS_RECEIVED = "android.provider.Telephony.SMS_RECEIVED"
    }

    override fun onReceive(context: Context, intent: Intent) {
        if (intent.action == SMS_RECEIVED) {
            val bundle = intent.extras
            if (bundle != null) {
                val pdus = bundle["pdus"] as Array<*>?
                val format = bundle.getString("format")
                val messages: Array<SmsMessage?> = arrayOfNulls(pdus!!.size)

                for (i in pdus.indices) {
                    messages[i] = SmsMessage.createFromPdu(pdus[i] as ByteArray, format)
                    if (messages.isNotEmpty()) Toast.makeText(context,
                        context.getString(R.string.new_sms_received, messages[0]?.messageBody),
                        Toast.LENGTH_SHORT).show()
                    context.sendBroadcast(Intent("SMS_RECEIVED"))
                }
            }
        }
    }
}

```

The broadcast receiver will monitor the broadcast intents that are emitted by the device. In this case, we are only interested in broadcast intents that involve newly received SMS messages. For this reason, the onReceive method is configured to check whether the broadcast intent action equals 'android.provider.Telephony.SMS\_RECEIVED'. When a new message is received, the onReceive method converts the Protocol Data Units (PDUs), which are the binary data blocks for SMS messages, into an array of SmsMessage objects. The first message in the array will be the most recently received, so the onReceive method locates the first SmsMessage in the array and displays its message body in a toast notification. Whenever a new SMS message arrives, the user will see a notification detailing the contents of that message.



Finally, the broadcast receiver uses the sendBroadcast method to dispatch an intent with a value of SMS\_RECEIVED to the context in which the receiver is running. In this app, the context will always be the MainActivity activity. The MainActivity activity should respond by refreshing the list of SMS objects loaded into the CommunicationViewModel view model to reflect the newly received message. For MainActivity to receive and respond to the intent, we will need to register a broadcast receiver in the MainActivity class. To do this, open MainActivity and add the following code below the communicationViewModel variable at the top of the class:

```

private val broadcastReceiver = object : BroadcastReceiver() {
    override fun onReceive(context: Context?, intent: Intent?) {
        getTextures()
    }
}

```

The broadcast receiver will detect incoming broadcast intents and run the getTextures method to refresh the list of SMS objects held by the CommunicationViewModel view model. To instruct the broadcast receiver to detect all broadcast intents with an action of 'SMS\_RECEIVED', add the following code to the bottom of the onCreate method.

```
registerReceiver(broadcastReceiver, IntentFilter("SMS_RECEIVED"))
```

The MainActivity class will now register the broadcast receiver, which will run the `getTexts` method whenever an intent with an action of 'SMS\_RECEIVED' arrives. An important consideration though is that we must unregister the broadcast receiver when the activity shuts down to prevent memory leaks and any unnecessary drains on the device's resources. To unregister the broadcast receiver when the activity is being destroyed, add the following code below the `onCreate` method:

```
override fun onDestroy() {  
    super.onDestroy()  
    unregisterReceiver(broadcastReceiver)  
}
```

The final consideration here is that for the `SMSBroadcastReceiver` class to successfully detect when SMS messages arrive, the app must have permission to monitor newly received messages. To request the necessary user permissions when the app is launched, add the following code to the `onCreate` method above where the broadcast receiver is registered:

```
if (ContextCompat.checkSelfPermission(this, Manifest.permission.RECEIVE_SMS)  
    != PackageManager.PERMISSION_GRANTED) {  
    ActivityCompat.requestPermissions(this, arrayOf(Manifest.permission.RECEIVE_SMS), 0)  
}
```

## The BottomNavigationView widget

The app is almost finished; the user can initiate phone calls and send and receive SMS messages. The last task is to allow the user to navigate around the app by configuring the `BottomNavigationView` widget. The `BottomNavigationView` widget was created automatically as part of the Bottom Navigation Activity project template. It will allow the user to navigate between the top-level destinations in the app, which in this case is the phone fragment, call log fragment and sms fragment.

Each fragment will be mapped by a navigation graph. The navigation graph will tell the app which fragment to load when the user clicks an item in the `BottomNavigationView`. Android Studio will have automatically generated a navigation graph when the project was created. To locate it, navigate through **Project** > **app** > **res** > **navigation** and open the file called `mobile_navigation.xml`. Edit the file as follows to define separate destinations and corresponding files for the phone, call log and sms fragments:

```
<?xml version="1.0" encoding="utf-8"?>  
<navigation xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:app="http://schemas.android.com/apk/res-auto"  
    xmlns:tools="http://schemas.android.com/tools"  
    android:id="@+id/mobile_navigation"  
    app:startDestination="@+id/navigation_phone">
```

```
    <fragment  
        android:id="@+id/navigation_phone"  
        android:name="com.example.communication.ui.phone.PhoneFragment"  
        android:label="@string/phone"  
        tools:layout="@layout/fragment_phone" />
```

```
    <fragment  
        android:id="@+id/navigation_sms"  
        android:name="com.example.communication.ui.sms.SMSFragment"  
        android:label="@string/sms"  
        tools:layout="@layout/fragment_sms" />
```

```
    <fragment  
        android:id="@+id/navigation_call_log"  
        android:name="com.example.communication.ui.callLog.CallLogFragment"  
        android:label="@string/call_log"  
        tools:layout="@layout/fragment_call_log" />
```

```
</navigation>
```

The navigation graph is now correctly configured to allow the user to navigate to different destinations within the app. To populate the contents of the `BottomNavigationView` widget, we must define the top-level destinations. To

do this, navigate through **Project > app > res > menu** and open the file called **bottom\_nav\_menu.xml**. Switch the file to Code view and replace the automatically generated menu items with the following code:

```
<item
    android:id="@+id/navigation_sms"
    android:icon="@drawable/ic_sms"
    android:title="@string/sms" />
```

```
<item
    android:id="@+id/navigation_phone"
    android:icon="@drawable/ic_phone"
    android:title="@string/phone" />
```

```
<item
    android:id="@+id/navigation_call_log"
    android:icon="@drawable/ic_time"
    android:title="@string/call_log" />
```

The menu items defined above will appear in the BottomNavigationView widget. The ID of each menu item should match the ID of the corresponding navigation destination in the **mobile\_navigation.xml** navigation graph. Each menu item also contains an icon and a title that will be displayed in the BottomNavigationView widget.



The BottomNavigationView widget can be found in the **activity\_main.xml** layout (**Project > app > res**). The activity\_main layout is the main layout of the activity and will load when the app is launched. It contains the BottomNavigationView widget and a fragment, which will display the content of the user's current destination in the app. It is now convention to use a FragmentContainerView widget rather than a regular fragment to display content, so replace the fragment with the following code:

```
<androidx.fragment.app.FragmentContainerView
    android:id="@+id/nav_host_fragment"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    app:defaultNavHost="true"
    app:navGraph="@navigation/mobile_navigation"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toTopOf="@id/nav_view" />
```

The above code defines a FragmentContainerView widget that will source its content from the **mobile\_navigation.xml** navigation graph. The height of the FragmentContainerView is set to 0dp, which means its height will occupy the maximum available space according to its constraints. In this case, the FragmentContainerView is constrained to the top of the parent layout and the BottomNavigationView at the bottom of the layout. These constraints mean the FragmentContainerView will occupy the maximum available space once it has left enough room for the BottomNavigationView.

Note the root ConstraintLayout in the **activity\_main.xml** layout may contain a paddingTop attribute `android:paddingTop="?attr/actionBarSize"` designed to leave space for an action bar. This attribute will not be necessary for this app and can be deleted.

To make the BottomNavigationView widget operational, return to the **MainActivity.kt** file and replace the navController and appBarConfiguration variables in the onCreate method with the following code:

```
val navHostFragment = supportFragmentManager.findFragmentById(R.id.nav_host_fragment) as
NavHostFragment
val navController = navHostFragment.navController
val appBarConfiguration = AppBarConfiguration(setOf(R.id.navigation_phone, R.id.navigation_call_log,
R.id.navigation_sms))
```

The onCreate method now initialises a NavHostFragment object, which will provide access to the FragmentContainerView widget from the **activity\_main.xml** layout. The FragmentContainerView will allow the user to navigate between the destinations defined in the **mobile\_navigation.xml** navigation graph via a NavController object. We also defined a variable called appBarConfiguration, which details the app's top-level destinations. A top-level destination is considered the origin of a navigation pathway. In the Communication app, the phone fragment, call log fragment and sms fragment are all top-level destinations.

## Summary

Congratulations on completing the Communications app! In creating this app, you have covered the following skills and topics:

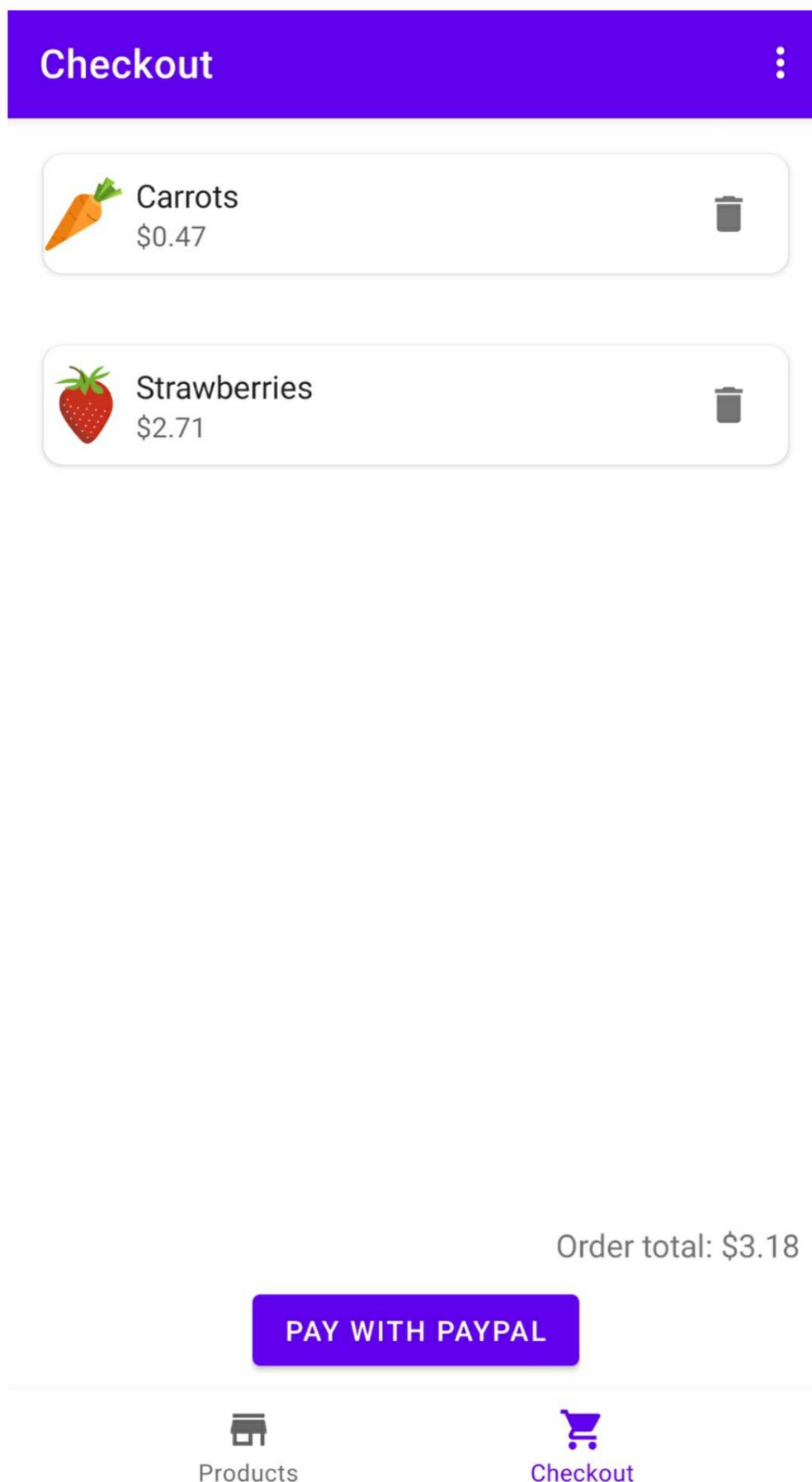
- Creating a new application using the Bottom Navigation Activity project template.
- Manage different fragments in an application.
- Use a content resolver query to retrieve the call log and SMS message history and store the results in a Cursor interface table.
- Use a variety of layout container Views including RelativeLayout, LinearLayout ConstraintLayout, ScrollView, HorizontalScrollView and GridLayout.
- Initiate phone calls.
- Display dialog fragments and popup menus.
- Use the device's SMS manager to send single or multi-part SMS messages.
- Register a broadcast receiver and respond to new broadcast events, such as the arrival of a new SMS message.

# How to create a mobile Store application

## Introduction

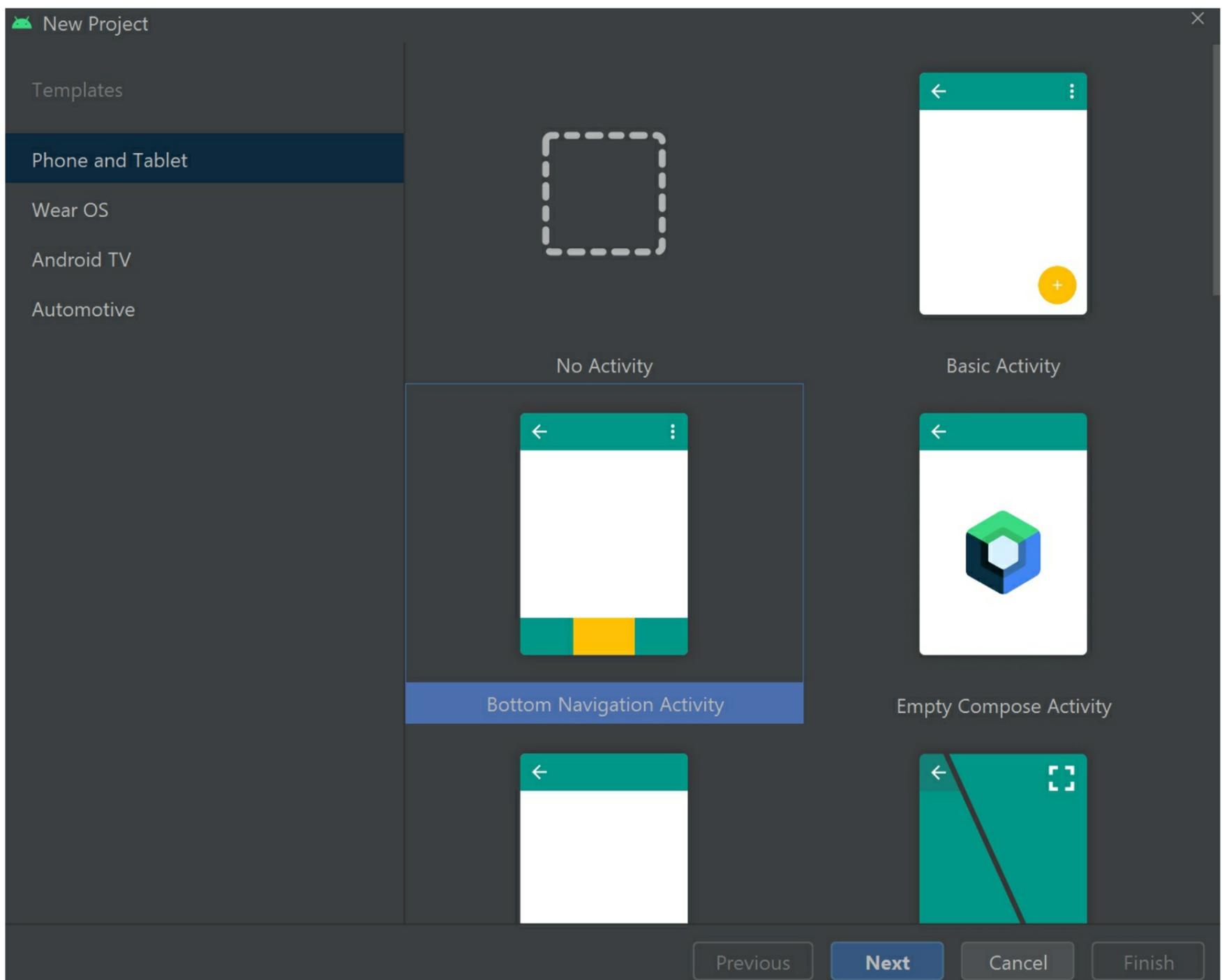
For this project, we will create a mobile store application that allows the user to shop for products and pay using debit/credit card, PayPal and more! Payments will be handled using the Braintree API, which is the recommended mobile payment processing system for PayPal (<https://www.braintreepayments.com/gb>) and is fully compatible with other digital wallets such as Apple Pay and Google Pay. After completing this project, you will know how to integrate the Braintree payment system into your app, process transactions and use a currency exchange API (<https://www.exchangerate-api.com/>) to allow the customer to shop in different currencies.

To complete this project and process transactions using the Braintree API, you will need to have a website domain and server hosting provider that you can upload files to. For a guide on how to register a website domain and choose a hosting provider, then see this tutorial: <https://codersguidebook.com/how-to-build-a-website/how-to-use-ftp-to-upload-website-files>



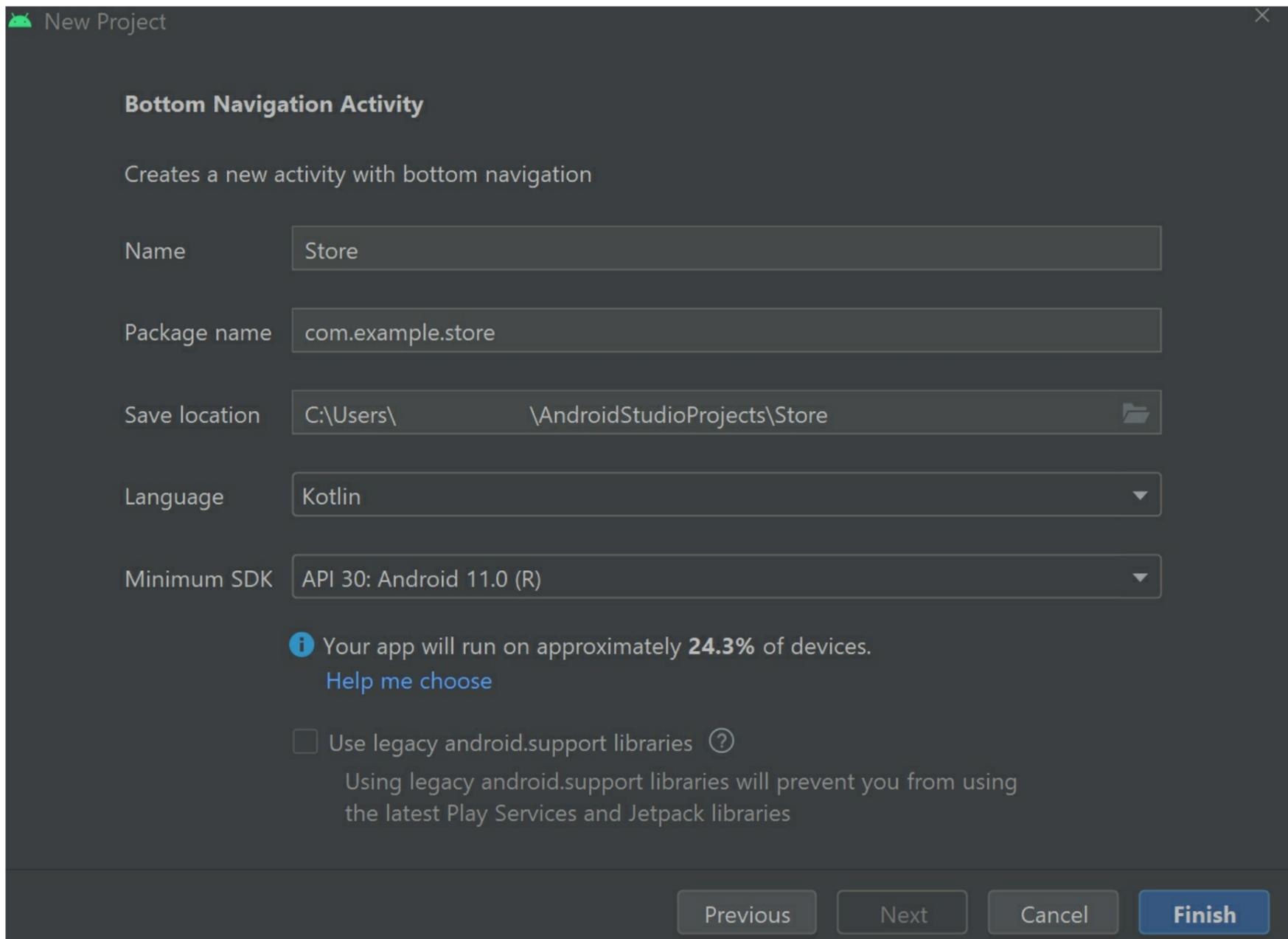
## Getting started

To begin, open Android Studio and create a new project using the Bottom Navigation Activity project template.



The Bottom Navigation Activity project template provides your app with a navigation bar at the bottom of the screen, as well as some readymade fragments. Each fragment represents a different destination in the app, which the user will be able to navigate using the navigation bar. In this app, there will be separate fragments for the products catalogue and checkout pages.

In the Create New Project window, name the project Store, set the language to Kotlin and select API level 30.

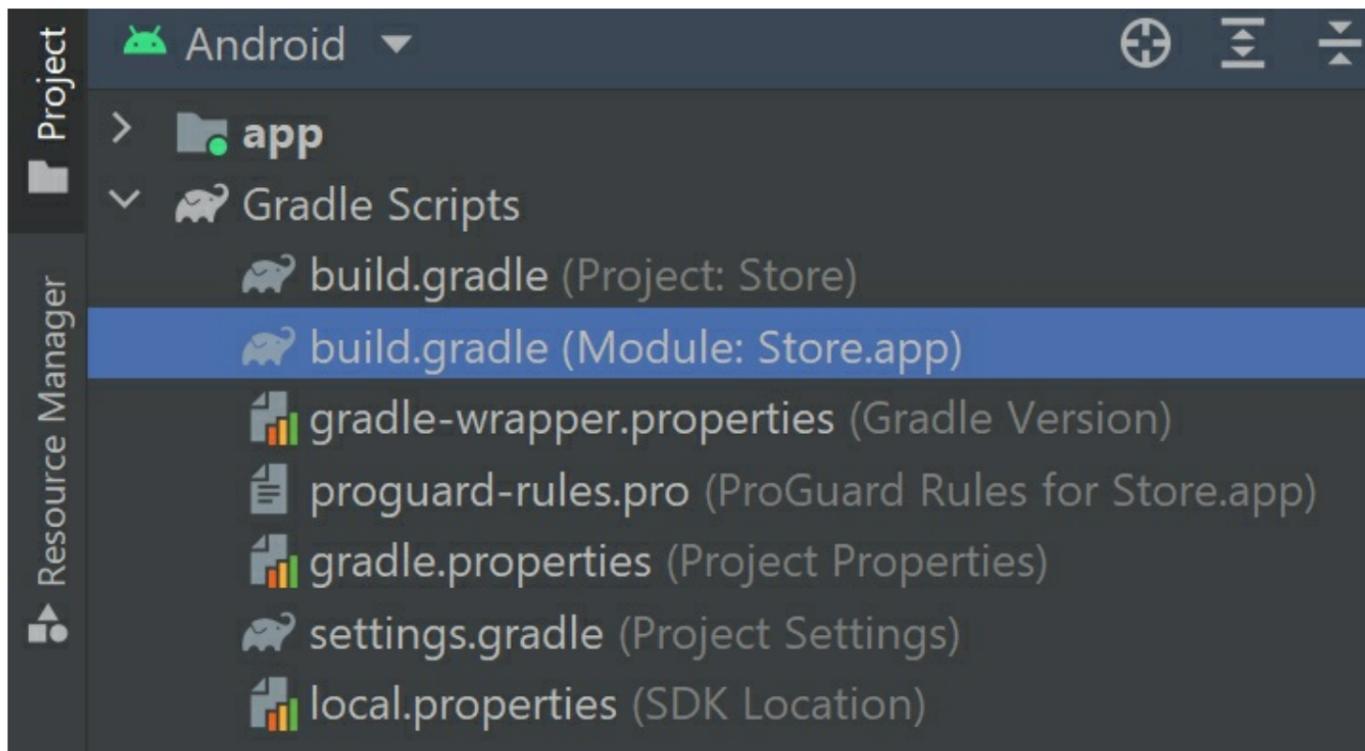


It is recommended you enable Auto Imports to direct Android Studio to add any necessary import statements to your Kotlin files as you code. These import statements are essential for incorporating the external classes and tools required for the app to run. To enable Auto Imports, open Android Studio's Settings window by clicking **File > Settings**. In the Settings window, navigate through **Editor > General > Auto Import** then select 'Add unambiguous imports on the fly' and 'Optimise imports on the fly' for both Java and Kotlin then press Apply and OK.

Android Studio should now add most of the necessary import statements to your Kotlin class files automatically. Sometimes there are multiple classes with the same name and the Auto Import feature will not work. In these instances, the requisite import statement(s) will be specified explicitly in the example code. You can also refer to the finished project code which accompanies this book to find the complete files including all import statements.

## Configuring the Gradle scripts

For the Store app to perform all the operations we want it to, we must manually import some external packages using a toolkit called Gradle. To do this, navigate through **Project > Gradle Scripts** and open the Module-level **build.gradle** file:



Next, refer to the dependencies element and add the following code to the list of implementation statements:

```
implementation 'com.github.bumptech.glide:glide:4.11.0'
implementation 'androidx.preference:preference-ktx:1.2.0'
implementation 'com.loopj.android:android-async-http:1.4.9'
implementation 'com.braintreepayments.api:paypal:4.4.1'
implementation 'com.braintreepayments.api:data-collector:4.7.0'
```

The above implementation statements enable the app to access an image rendering tool called Glide, a shared preferences feature that will allow the app to write data to a file that is readily accessible across every area of the app and persists when the app is closed, an asynchronous HTTP client that can interact with web pages, and several Braintree payments packages that process PayPal payments and handle user data.

We're now finished with the Gradle Scripts files. Don't forget to re-sync your project when prompted!

Gradle files have changed since last project sync. A project sync may be necessary for the IDE ... [Sync Now](#)

## Configuring the Manifest file

We'll now turn our attention to the application's manifest file, which contains an overview of the app's activities, as well as the user permissions the app requires to run. Open the **AndroidManifest.xml** file by navigating through **Project > app > manifests** then add the following line of code above the application element:

```
<uses-permission android:name="android.permission.INTERNET" />
```

The above uses-permission element signals to the device (and the Google Play store) that this app will require access to the internet.

Next, locate the activity element. You should notice an intent filter. Intent filters define actions that the parent component can respond to. In this case, the parent component is the MainActivity class. We need to define an additional intent filter for the MainActivity class so the user can return to the app after completing a payment using Braintree. For instance, PayPal, Venmo and 3D secure verification will require a browser window to open for payment authentication. To redirect users back to the app, add the following code below the existing intent filter:

```
<intent-filter>
  <action android:name="android.intent.action.VIEW" />
  <category android:name="android.intent.category.DEFAULT" />
  <category android:name="android.intent.category.BROWSABLE" />
  <data android:scheme="{applicationId}.braintree" />
</intent-filter>
```

The above code defines a URL scheme that the MainActivity class can use to return the user to the app once payment has been authenticated.

## Defining the string resources used in the app

Like the other projects covered in this book, the Store app will store all the strings of text used throughout the application in a resource file. To define the string resources, navigate through **Project > app > res** and open the file called **strings.xml**. Once the file opens in the editor, modify its contents so it reads as follows:

```
<resources>
  <string name="app_name">Store</string>
  <string name="id">Id</string>
  <string name="amount">Amount</string>
  <string name="status">Status</string>
  <string name="empty_basket">Your shopping basket is empty.</string>
  <string name="product_price">%1$s%2$s</string>
  <string name="pay_using_paypal">Pay with PayPal</string>
  <string name="paypal_error">A PayPal authentication error has occurred: %1$s</string>
  <string name="payment_error">Unfortunately there was an error processing the payment.</string>
  <string name="payment_successful">Congratulations! Your order has been placed successfully.</string>
  <string name="title_products">Products</string>
  <string name="title_checkout">Checkout</string>
  <string name="an_image_of_the_product">product_image</string>
  <string name="add_to_basket">Add to basket</string>
  <string name="remove_from_basket">Remove from basket</string>
  <string name="exchange_data_unavailable">Exchange rate data currently unavailable. Please try again shortly.
</string>
  <string name="select_currency">Select currency</string>
  <string name="order_total">Order total: %1$s</string>
  <string name="empty_cart">Your cart is empty.</string>

  <!-- TODO: Create a string title for each currency your store supports -->
  <string name="currency_gbp">Pound Sterling (£)</string>
  <string name="currency_usd">United States Dollar ($)</string>
  <string name="currency_eur">Euro (€)</string>
</resources>
```

Each string resource contains a name attribute, which we will use to reference the string elsewhere in the app. The text that will be displayed is input between the opening and closing string tags. You may notice that the `product_price` and `order_total` strings contain code in the format `'%1$s'`. These sections of code represent arguments that must be supplied when the string is created. The `'%1'` part represents the argument number, so will increase incrementally for each new argument that is added to the string e.g. `'%1'`, `'%2'`, `'%3'` etc. The second part of the argument indicates what data type is expected: `'$s'` represents a string, while `'$d'` represents a decimal integer and `'$.nf'` represents a floating number rounded to a certain number of decimal places (replace `'n'` with the number of decimal places e.g. `'$.2f'`). For example, in the above code, we can see the `product_price` string expects two string arguments. The first argument will display the currency symbol while the second argument will specify the price. To build the `product_price` string, you could write the following Kotlin code:

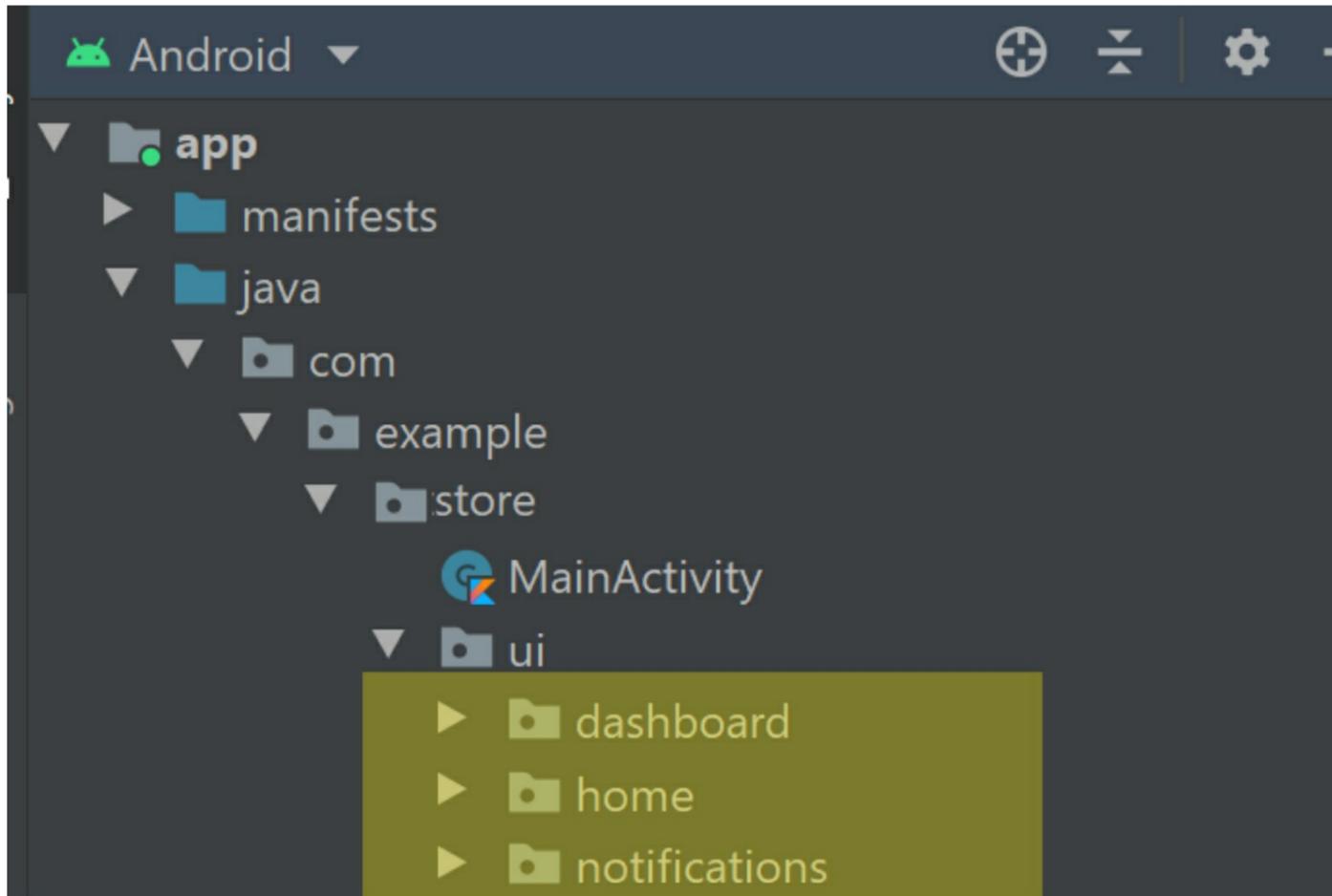
```
getString(R.string.product_price,"£", "17.99")
```

The above line imports the `product_price` string and supplies `"£"` as argument 1 and `"17.99"` as argument 2. The output string of the above code would be `"£17.99"`.

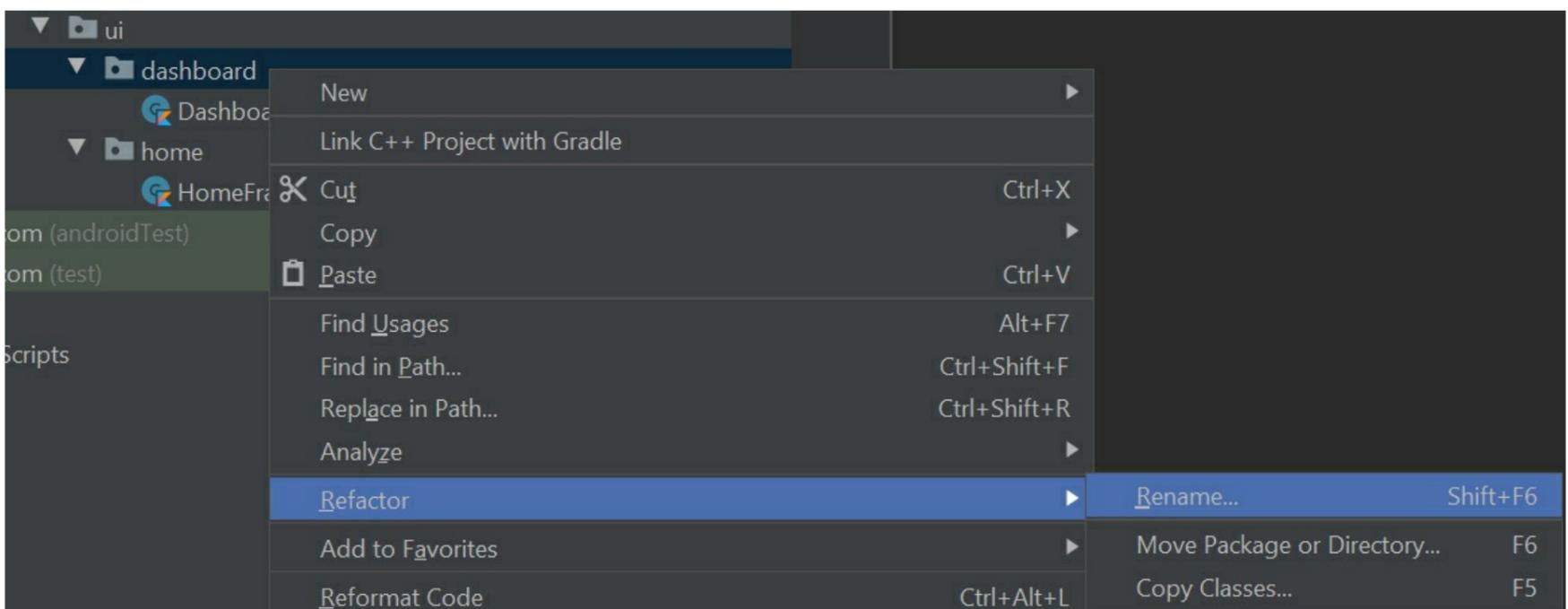
At the bottom of the **strings.xml** file, you should see a **TODO** comment that indicates where you should include a list of string resources detailing each currency that the app supports. These strings will populate the text in currency conversion buttons that the customer can use to change the currency that your products are priced. In this project, the app will support Pound Sterling, United States Dollar and Euro but you can add or remove currencies to suit your business requirements.

## Setting up the Products fragment and layout

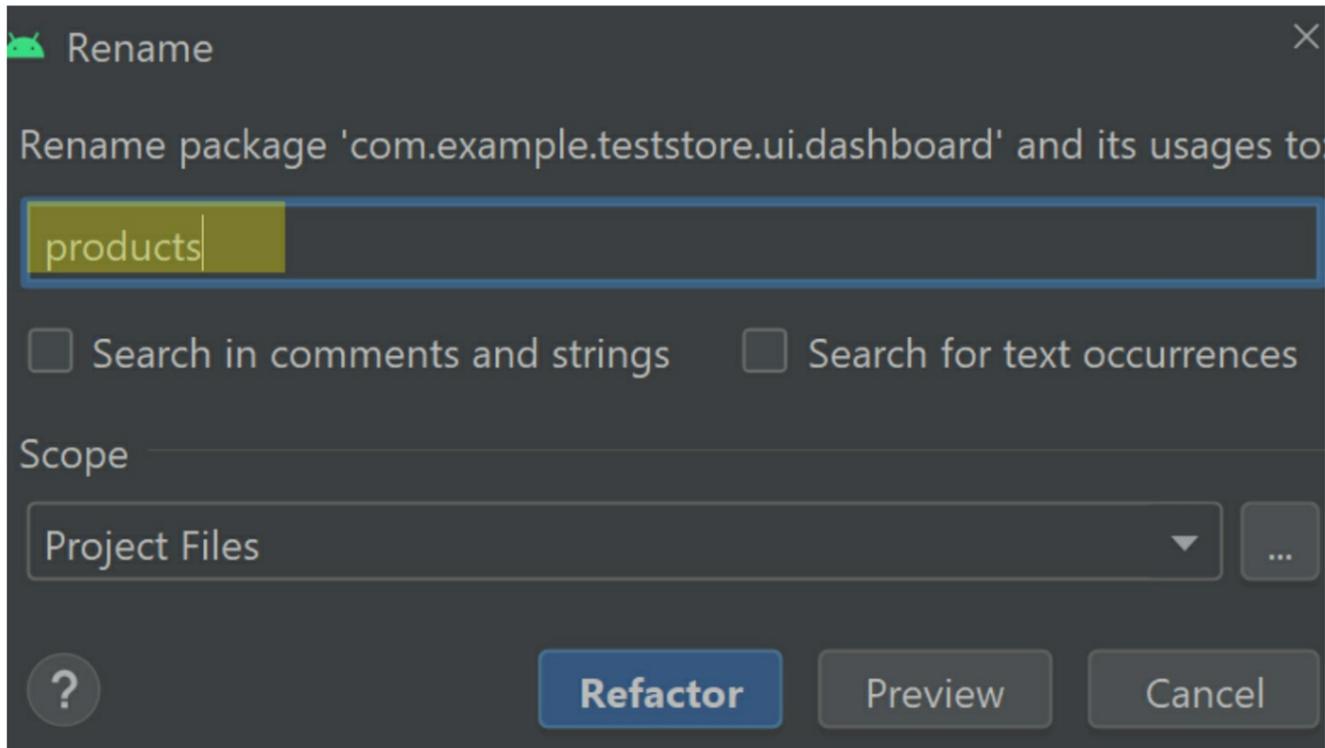
In this section, we will create a fragment that lists the catalogue of available products and allows the user to add items to their shopping basket. When we created the project using the Bottom Navigation Activity template, Android Studio automatically generated three fragment packages. You can locate them by navigating through **Project > app > java > name of project > ui**.



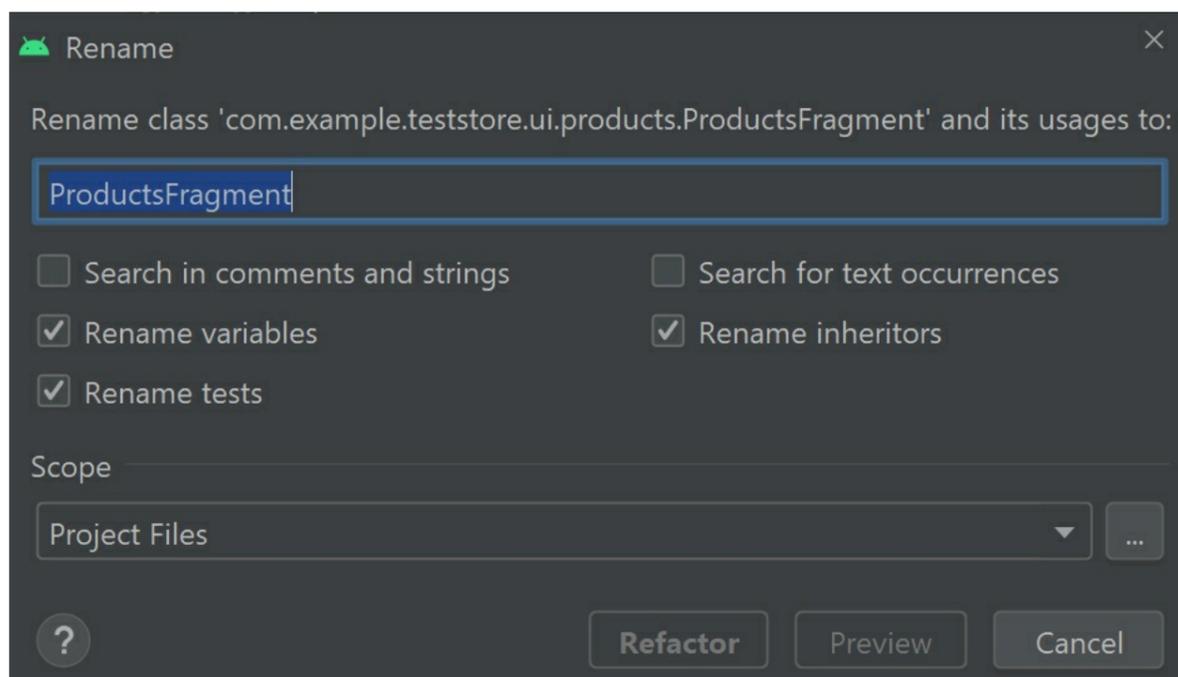
For this app, we will only require two fragments, so right-click one of the packages (e.g. notifications) and press Delete. Of the remaining two packages, we will not require either of the view models, so expand the **dashboard** and **home** directories, right-click the **DashboardViewModel.kt** and **HomeViewModel.kt** files and press Delete. Next, we need to rename the remaining fragment files to suit our application. Android Studio provides a Refactor tool that allows you to rename an item (e.g. a file, variable or class) and automatically update all in-code references to the item with the new information. To refactor the dashboard package and repurpose it for the products fragment, right-click the **dashboard** directory then press **Refactor > Rename**.



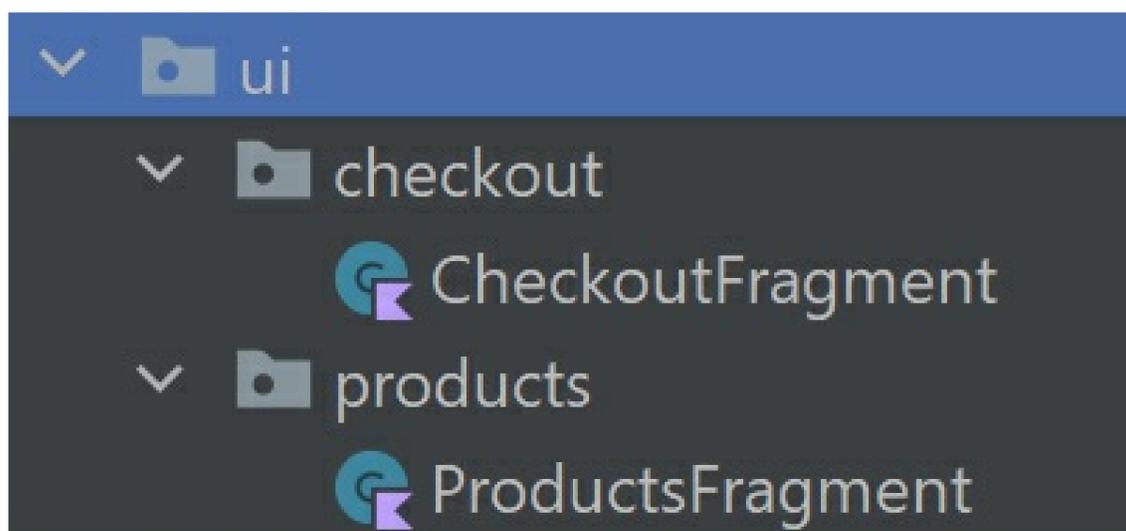
Change the name to products then press Refactor.



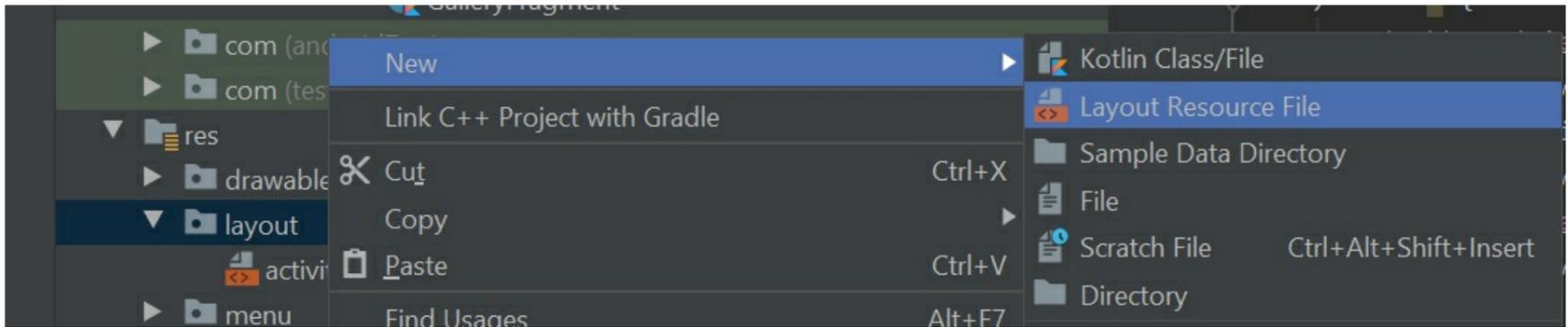
Next, right-click the **DashboardFragment.kt** file and again select **Refactor** > **Rename**. Set the new name to **ProductsFragment** then press Refactor.



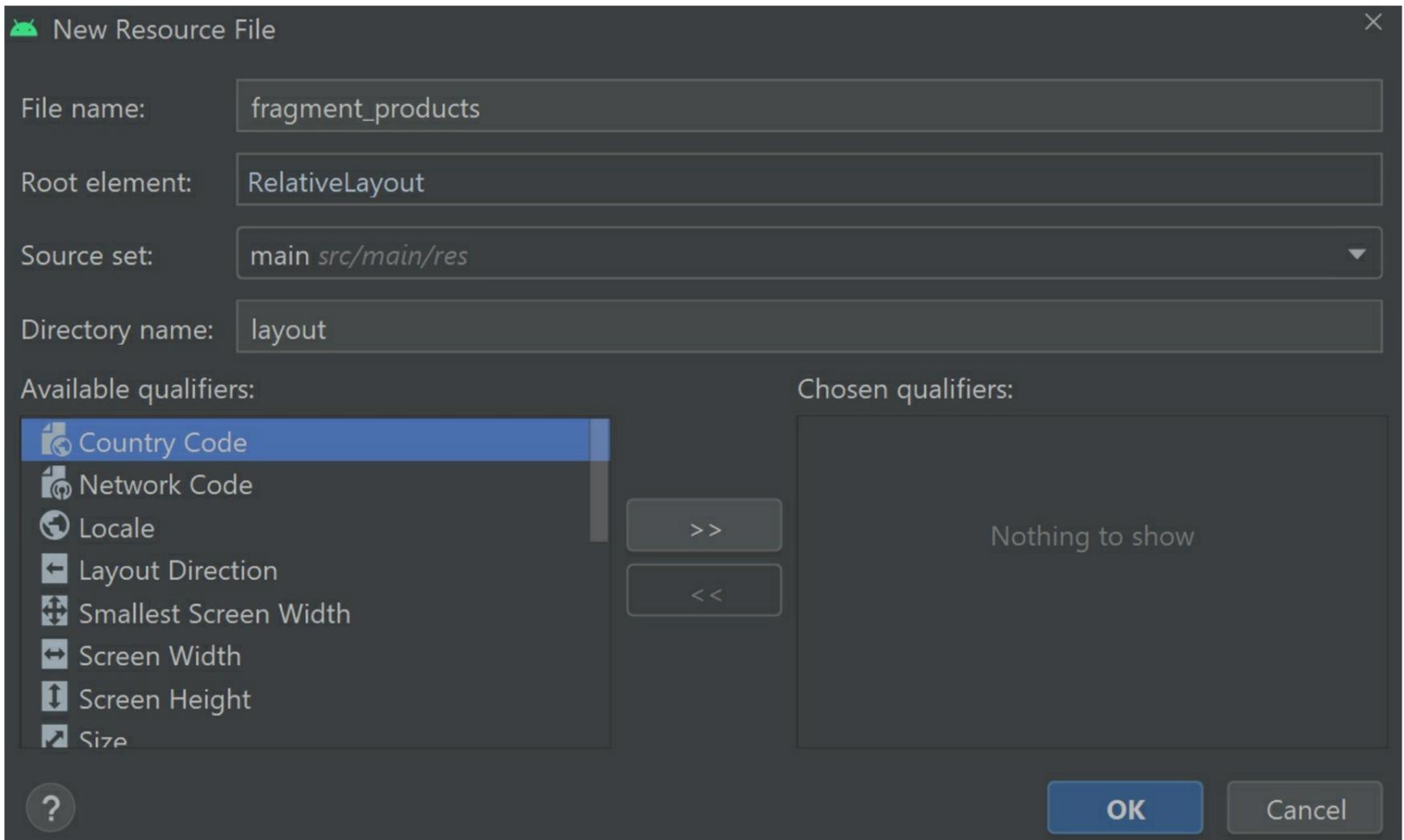
The products fragment and directory have now been successfully refactored. While we're here, it may be a good opportunity to also prepare the Checkout fragment. To do this, refactor the **home** directory to **checkout** and refactor the **HomeFragment** file to **CheckoutFragment**.



The products fragment will require a layout, so locate the **layout** folder by navigating through **Project** > **app** > **res**. The readymade **fragment\_dashboard.xml**, **fragment\_home.xml** and **fragment\_dashboard.xml** files can be deleted because we will not use them. Instead, create a new layout file by right-clicking the **layout** folder and selecting **New** > **Layout Resource File**.



Name the file **fragment\_products**, set the root element to RelativeLayout then press OK.



Once the **fragment\_products.xml** file opens in the editor, add the following code inside the RelativeLayout element:

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/productsRecyclerView"
    android:visibility="gone"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

```
<ProgressBar
    android:id="@+id/loadingProgress"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:indeterminate="true"
    android:layout_centerInParent="true" />
```

The above code defines a RecyclerView widget that will display the details of each available product. Initially, the RecyclerView will be hidden because its visibility is set to gone. The RecyclerView is hidden because the products fragment will not display the list of products until it has connected to the currency exchange API and calculated the price of each product. For this reason, the layout also contains a ProgressBar widget with an indeterminate attribute set to true. This means the progress bar will continuously loop like a loading symbol until the list of products is ready. Once the price of each product has been calculated, the ProgressBar will disappear and the RecyclerView will become visible. To position the ProgressBar widget in the center of the RelativeLayout container, we set the ProgressBar's centerInParent attribute to true.

The layout for the camera fragment is now complete. To integrate the layout with the fragment, open the

**ProductsFragment.kt** file and replace the `dashboardViewModel` and `_binding` variables at the top of the class with the following code:

```
private var _binding: FragmentProductsBinding? = null
private lateinit var callingActivity: MainActivity
```

The above variables allow the fragment to access the data and methods in the **fragment\_products.xml** layout's binding class and the `MainActivity` class, respectively. To initialise the variables, edit the `onCreateView` method so it reads as follows:

```
override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
): View {
    _binding = FragmentProductsBinding.inflate(inflater, container, false)

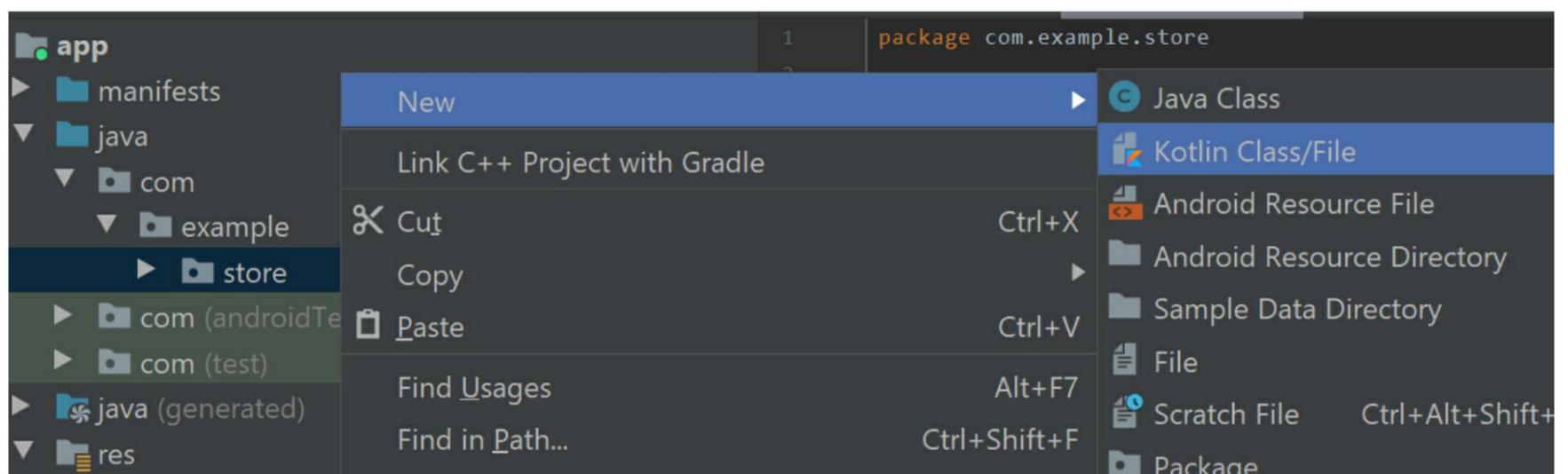
    callingActivity = activity as MainActivity

    return binding.root
}
```

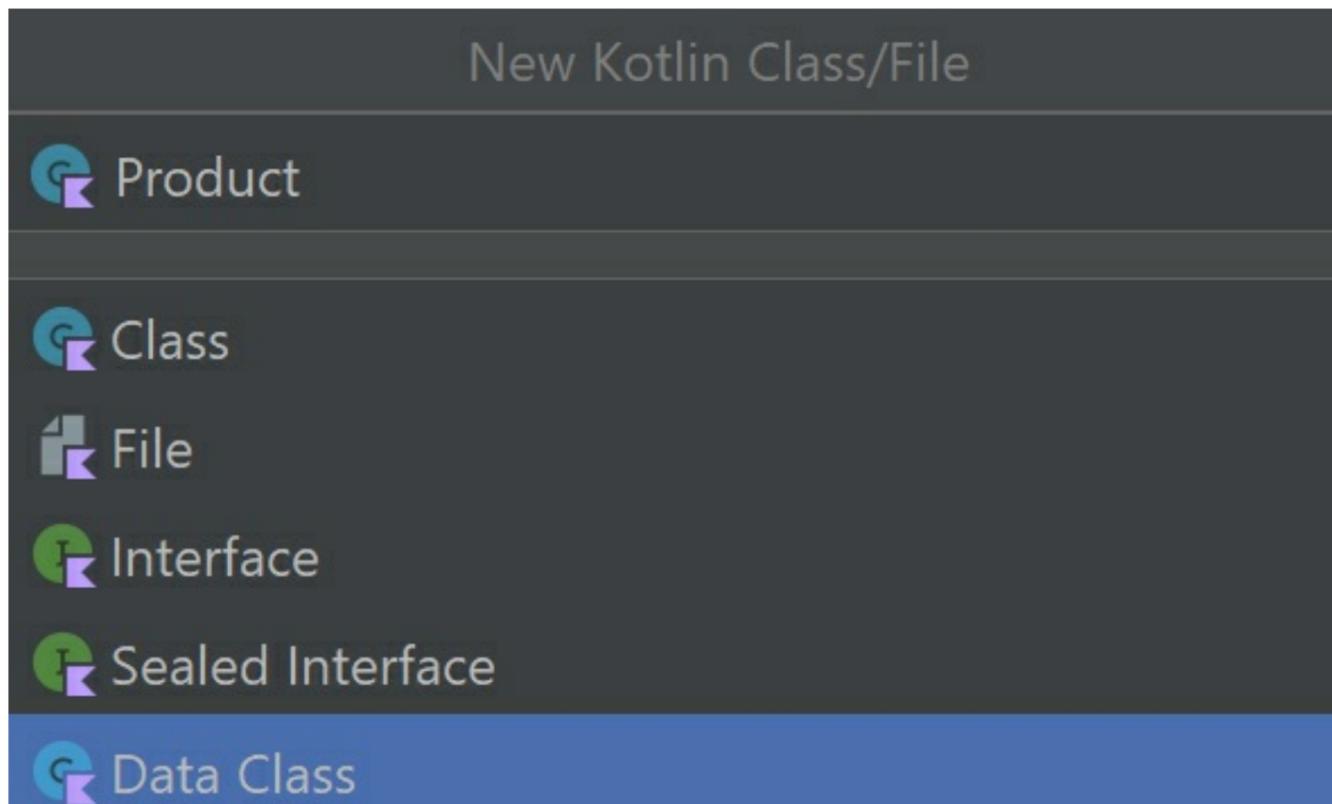
The above code creates a binding instance between the `ProductsFragment` and the **fragment\_products.xml** layout via the layout's `FragmentProductsBinding` binding class. The fragment can use the binding class to interact with the layout and its widgets.

## Storing product information

The details of each product will be stored as objects of a data class. The data class will be called `Product` and will have separate fields for each item of information such as the product's name, price and image. To create the `Product` data class, navigate through **Project > app > java** then right-click the folder with the name of the project and select **New > Kotlin File/Class**.



Name the file `Product` and select `Data Class` from the list of options.



A file called **Product.kt** should then open in the editor. Modify the class so its code reads as follows:

```
data class Product(  
    // In this app the images used are saved within the app as drawable resources.  
    // In a live app you may instead prefer to store a link (as a String) to an image stored online (e.g. on your website  
server)  
    var image: Int,  
    var name: String,  
    // Note the price of each product should be input in the base currency of the store  
    var price: Double,  
    var inCart: Boolean = false  
)
```

The above code defines a data class that will hold information about each available product. The data class's primary constructor contains four parameters, each storing a different bit of data about the product. First, the image variable will store an integer value that will identify the drawable resource file that will be used as the product's image. Referencing a drawable resource assumes that the image file is included within the app. If your store will be retrieving product images from the internet then it may be preferable to change the data type of the image variable to 'String' so you can store a link to the online image.

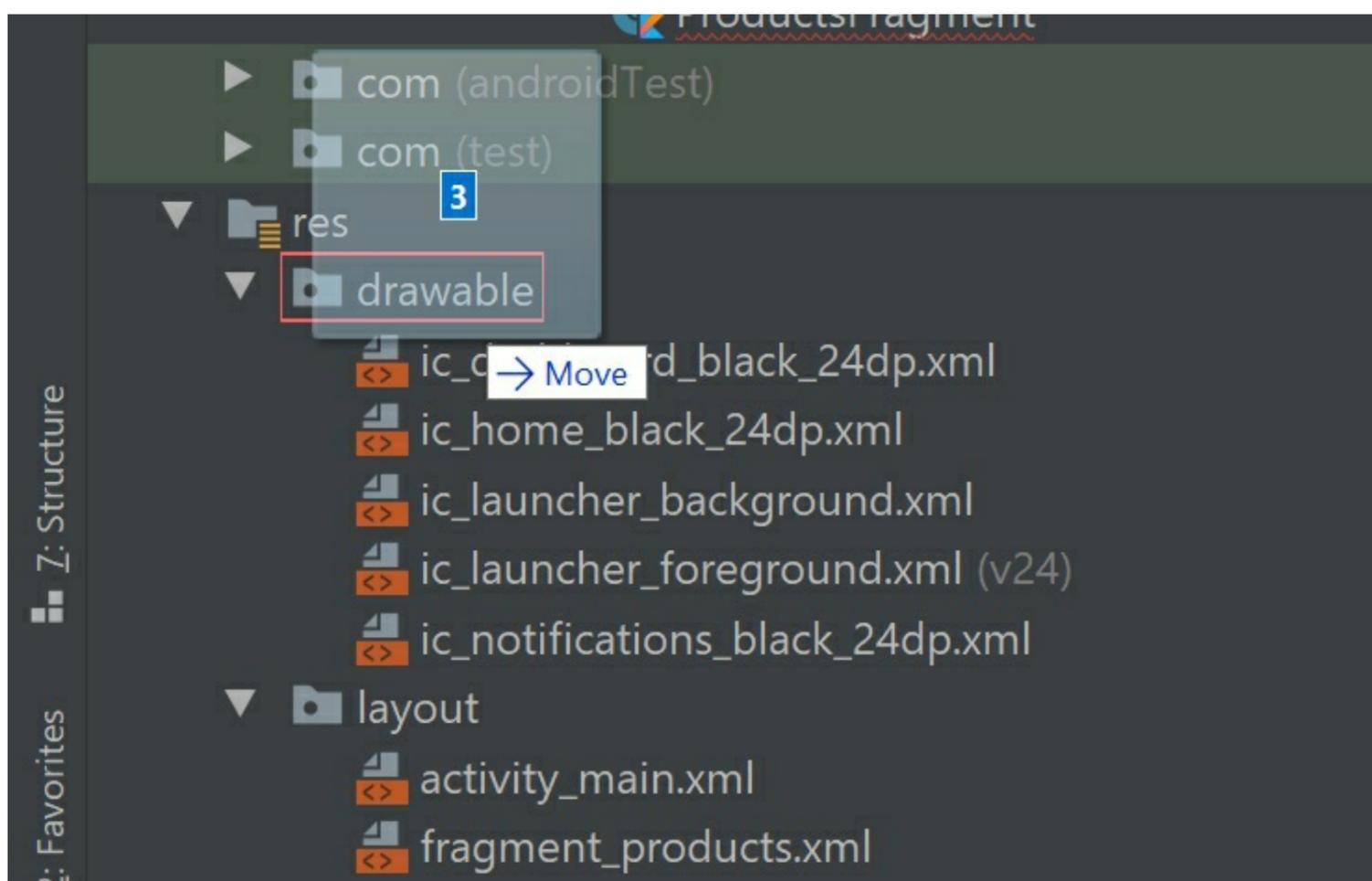
```
var image: String
```

Next, the name variable will store the name of the product, the price variable will contain the price of the product in the base currency of the store, and the inCart variable will store a boolean value indicating whether or not the product is currently in the user's shopping cart. By default, the inCart variable is set to false because products will not automatically be added to the customer's shopping cart.

As mentioned above, for this example project the images for each product will be stored as drawable resources within the app. Specifically, the example store will allow the user to shop for broccoli, carrots and strawberries. If you would like to add the drawable resources for these products into your project, then you will need to copy the source files from the example code into the **drawable** folder of your project. To locate the drawable files in the example code, navigate through the following folders: **app > src > main > res > drawable**. The files you should copy are highlighted below. You could also copy the unhighlighted files over; some are default drawable files created by Android Studio and others are drawable icon resources we will create later.

Name	Date modified	Type
<input checked="" type="checkbox"/> broccoli	07/01/2021 23:12	PNG File
<input checked="" type="checkbox"/> carrot	07/01/2021 23:13	PNG File
<input type="checkbox"/> ic_cart	31/10/2021 20:16	XML Document
<input type="checkbox"/> ic_dashboard_black_24dp	22/10/2021 21:53	XML Document
<input type="checkbox"/> ic_delete	27/10/2021 17:55	XML Document
<input type="checkbox"/> ic_home_black_24dp	22/10/2021 21:53	XML Document
<input type="checkbox"/> ic_launcher_background	22/10/2021 21:53	XML Document
<input type="checkbox"/> ic_notifications_black_24dp	22/10/2021 21:53	XML Document
<input type="checkbox"/> ic_store	31/10/2021 20:16	XML Document
<input checked="" type="checkbox"/> strawberry	07/01/2021 23:12	PNG File

To copy image files into the Android Studio project, simply drag and drop them into the **drawable** directory as shown below:



The broccoli, carrot and strawberry image source files were made by Icongreek26 from [www.flaticon.com](http://www.flaticon.com) and are free for commercial use with attribution.

The list of available products will be managed by a view model. The view model will store the list of products and distribute the list's contents to other areas of the app. To create the view model, right-click the folder that contains the MainActivity class (**Project > app > java > name of the project**) and select **New > Kotlin File/Class**. Name the file **StoreViewModel** and select **Class** from the list of options. Once the **StoreViewModel.kt** file opens in the editor, modify its code so it reads as follows:

```
class StoreViewModel : ViewModel() {
    var products = MutableLiveData<List<Product>>()
}
```

The above code defines a variable called **products** that will store a mutable list of **Product** objects. Other areas of the app can observe this list and monitor any changes. To initialise the view model in the MainActivity class, open the **MainActivity.kt** file (**Project > app > java > name of the project**) and add the following variable to the top of the

class:

```
private val storeViewModel: StoreViewModel by viewModels()
```

Note you may need to add the following import statement to the top of the file:

```
import androidx.activity.viewModels
```

Next, add the following code to the bottom of the onCreate method to populate the view model's products variable with Product objects:

```
/* FIXME: Here we manually define a list of products
   In reality, you may want to retrieve product information in real-time from your website. */
val broccoli = Product(R.drawable.broccoli, "Broccoli", 1.40)
val carrots = Product(R.drawable.carrot, "Carrots", 0.35)
val strawberries = Product(R.drawable.strawberry, "Strawberries", 2.00)
val items = listOf(broccoli, carrots, strawberries)
storeViewModel.products.value = items
```

The above code defines separate instances of the Product class for the broccoli, carrots and strawberries products. To construct each Product object, we must provide the ID of the drawable resource that will be used for the image (R.drawable.broccoli), the name of the product ("Broccoli") and the price of the product in the base currency of the store (1.40). We do not need to provide a value for the inCart parameter because it is false by default, as specified in the Product data class primary constructor. Once all the Product objects have been created, they are added to a list and assigned to the StoreViewModel class's products variable so they can be accessed elsewhere in the app.

For this example project, the details of each product are predefined in the MainActivity class. In reality, you will likely want to retrieve the product information from your website server. There are a variety of ways you can do this, including using an HTTP client to query a web page and retrieve product data. We will discuss how to use an HTTP client in upcoming sections with regards to querying the Braintree and currency exchange APIs, so you could apply this logic to query your website and retrieve product information. If you need assistance with this, then you can discuss your requirements with a web developer or email us at [hello@codersguidebook.com](mailto:hello@codersguidebook.com).

## Displaying products in the Products fragment

In this section, we will load the list of products from the view model into the RecyclerView widget in the **fragment\_products.xml** layout. To facilitate this, we need to create a layout that will display information about each product. Create a new layout resource file by right-clicking the **layout** directory (**Project > app > res**) then selecting **New > Layout Resource File**. Name the layout product then press OK. Once the layout opens in the editor, switch it to Code view and edit the file so it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.cardview.widget.CardView 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="wrap_content"
    android:layout_margin="16dp"
    app:cardCornerRadius="10dp"
    app:cardElevation="2dp">
    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_marginVertical="6dp"
        android:orientation="vertical" >
        <ImageView
            android:id="@+id/productImage"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_gravity="center_horizontal"
```

```
android:contentDescription="@string/an_image_of_the_product" />
```

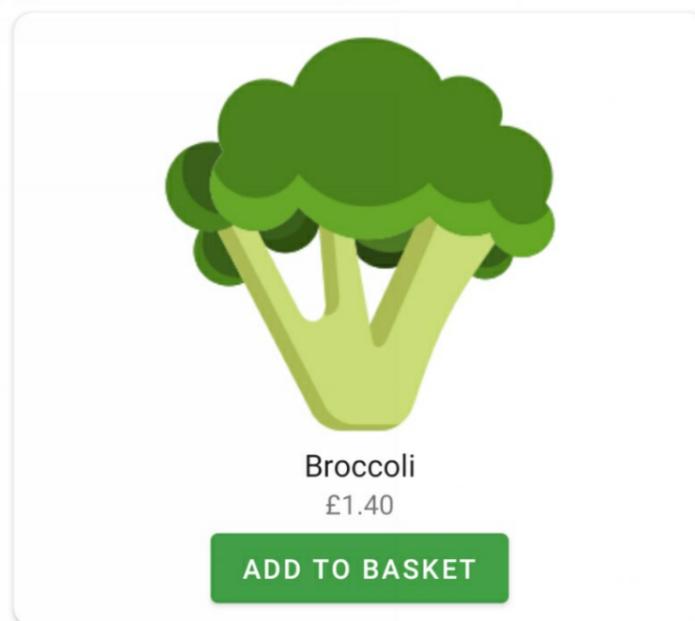
```
<TextView  
    android:id="@+id/productName"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_gravity="center_horizontal"  
    android:textColor="@color/material_on_surface_emphasis_high_type"  
    android:textSize="16sp" />
```

```
<TextView  
    android:id="@+id/productPrice"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_gravity="center_horizontal"  
    android:textColor="@color/material_on_surface_emphasis_medium"  
    android:textSize="14sp" />
```

```
<Button  
    android:id="@+id/addToBasketButton"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_gravity="center_horizontal"  
    android:text="@string/add_to_basket" />
```

```
</LinearLayout>  
</androidx.cardview.widget.CardView>
```

The root element of the product layout is a CardView widget. Card-based layouts appear slightly elevated above their containing view group (the RecyclerView in this instance) and allow you to display a list of items while keeping the style of each item consistent. In the above code, the CardView contains a LinearLayout widget that vertically aligns an ImageView widget, two TextView widgets and a Button widget. The ImageView widget will display an image of the product, while the two TextView widgets will display the product's name and price, respectively. Finally, the Button widget will enable the user to add or remove the product from their shopping basket.



Moving on, let's create an adapter that will coordinate the list of Product objects and load their information into the RecyclerView. To create the adapter class, right-click the **products** directory then select **New > Kotlin Class/File**. Name the file ProductsAdapter and select Class from the list of options. Once the **ProductsAdapter.kt** file opens in the editor, modify the class so it reads as follows:

```
class ProductsAdapter(private val activity: MainActivity, private val fragment: ProductsFragment) :  
    RecyclerView.Adapter<ProductsAdapter.ProductsViewHolder>() {  
    var products = mutableListOf<Product>()
```

```
    inner class ProductsViewHolder(itemView: View) :  
        RecyclerView.ViewHolder(itemView) {
```

```
        internal var mProductImage = itemView.findViewById<View>(R.id.productImage) as ImageView
```

```

internal var mProductName = itemView.findViewById<View>(R.id.productName) as TextView
internal var mProductPrice = itemView.findViewById<View>(R.id.productPrice) as TextView
internal var mAddToBasketButton = itemView.findViewById<View>(R.id.addToBasketButton) as Button
}
}

```

The above code adds parameters called activity and fragment to the ProductAdapter class's primary constructor. The parameters will store references to the MainActivity and ProductsFragment classes, respectively. In the body of the adapter, a variable called products will store the list of Product objects from the view model. Next, an inner class called ProductsViewHolder is established. This inner class will initialise the **product.xml** layout and store references to its widgets in variables so the adapter can interact with the widgets elsewhere.

To direct the adapter to use the **product.xml** layout, we must inflate the layout by overriding the Adapter class's onCreateView method. To do this, add the following code below the ProductsViewHolder inner class:

```

override fun onCreateView(parent: ViewGroup, viewType: Int): ProductsViewHolder {
return ProductsViewHolder(LayoutInflater.from(parent.context).inflate(R.layout.product, parent, false))
}

```

Next, to load the product data associated with each item in the RecyclerView, we need to override an Adapter class method called onBindViewHolder. To do this, add the following code below the onCreateView method:

```

override fun onBindViewHolder(holder: ProductsViewHolder, position: Int) {
val current = products[position]

Glide.with(activity)
.load(current.image)
.transition(DrawableTransitionOptions.withCrossFade())
.centerCrop()
.override(600, 600)
.into(holder.mProductImage)

holder.mProductName.text = current.name

// TODO: Set the price of the product here

if (current.inCart) {
holder.mAddToBasketButton.text = activity.resources.getString(R.string.remove_from_basket)
holder.mAddToBasketButton.setBackgroundColor(ContextCompat.getColor(activity,
android.R.color.holo_red_dark))
} else {
holder.mAddToBasketButton.text = activity.resources.getString(R.string.add_to_basket)
holder.mAddToBasketButton.setBackgroundColor(ContextCompat.getColor(activity,
android.R.color.holo_green_dark))
}

holder.mAddToBasketButton.setOnClickListener {
fragment.updateCart(position)
}
}
}

```

In the above code, the onBindViewHolder method loads the corresponding Product object from the products list for the current position in the RecyclerView. Next, an image loading framework called Glide retrieves the drawable resource referenced in the Product object's image parameter and displays it in the productImage ImageView. The above code also directs Glide to apply a fade animation when loading the image, crop the image if necessary and set the image's dimensions to 600 pixels \* 600 pixels. Next, the name of the product is loaded into the productName TextView and the text and colour of the addToBasketButton are modified based on whether the inCart parameter of the Product object is true or false. If the product is currently in the user's shopping basket, then the button will be red and display the text "Remove from basket". Meanwhile, if the product is not in the user's shopping basket, then the button will be green and display the text "Add to basket". If the add to basket button is clicked, then a method in the ProductsFragment class called updateCart will add or remove the product from the customer's shopping basket as appropriate.

To finalise the adapter, we need to define a method called getItemCount. The method will calculate how many items are loaded into RecyclerView. Invariably, the total number of items will equal the size of the products list, so

define the getItemCount method by adding the following code below the onBindViewHolder method:

```
override fun getItemCount() = products.size
```

Moving on, let's integrate the adapter with the products fragment and apply it to the RecyclerView. To do this, open the **ProductsFragment.kt** file and add the following variables to the list of variables at the top of the class:

```
private val storeViewModel: StoreViewModel by activityViewModels()
private lateinit var productsAdapter: ProductsAdapter
```

Note you may need to add the following import statement to the top of the file:

```
import androidx.fragment.app.activityViewModels
```

The above variables will allow the fragment to access and interact with the StoreViewModel and ProductsAdapter classes. Next, add the following code below the onCreate method to direct the onCreateView stage of the fragment lifecycle to initialise the ProductsAdapter class and apply it to the RecyclerView:

```
override fun onCreateView(view: View, savedInstanceState: Bundle?) {
    super.onCreateView(view, savedInstanceState)

    productsAdapter = ProductsAdapter(callingActivity, this)
    binding.productsRecyclerView.layoutManager = LinearLayoutManager(activity)
    binding.productsRecyclerView.itemAnimator = DefaultItemAnimator()
    binding.productsRecyclerView.adapter = productsAdapter
}
```

In the above code, the ProductsAdapter class is initialised and applied to the RecyclerView widget from the **fragment\_products.xml** layout. RecyclerView widgets use a layout manager to organise their content. This RecyclerView will display a list of products stacked linearly one by one and so the LinearLayoutManager is best. Next, an instance of the DefaultItemAnimator class is applied to the RecyclerView, which will provide some basic animations when items are added, removed or updated.

To load the list of products from the StoreViewModel view model into the adapter, add the following code to the bottom of the onCreateView method:

```
productsAdapter.products = storeViewModel.products.value?.toMutableList() ?: mutableListOf()
productsAdapter.notifyItemRangeInserted(0, productsAdapter.products.size)
```

The above code sets the ProductAdapter class's products variable to equal the contents of the StoreViewModel view model's products variable, or an empty list if the view model's products variable is null. It also calls the Adapter class's notifyItemRangeInserted method to notify the adapter that items have been added.

Finally, let's define a method called updateCart that the ProductsAdapter class will run whenever the user clicks the add to basket button for a given product. To define the updateCart method, add the following code below the onCreateView method:

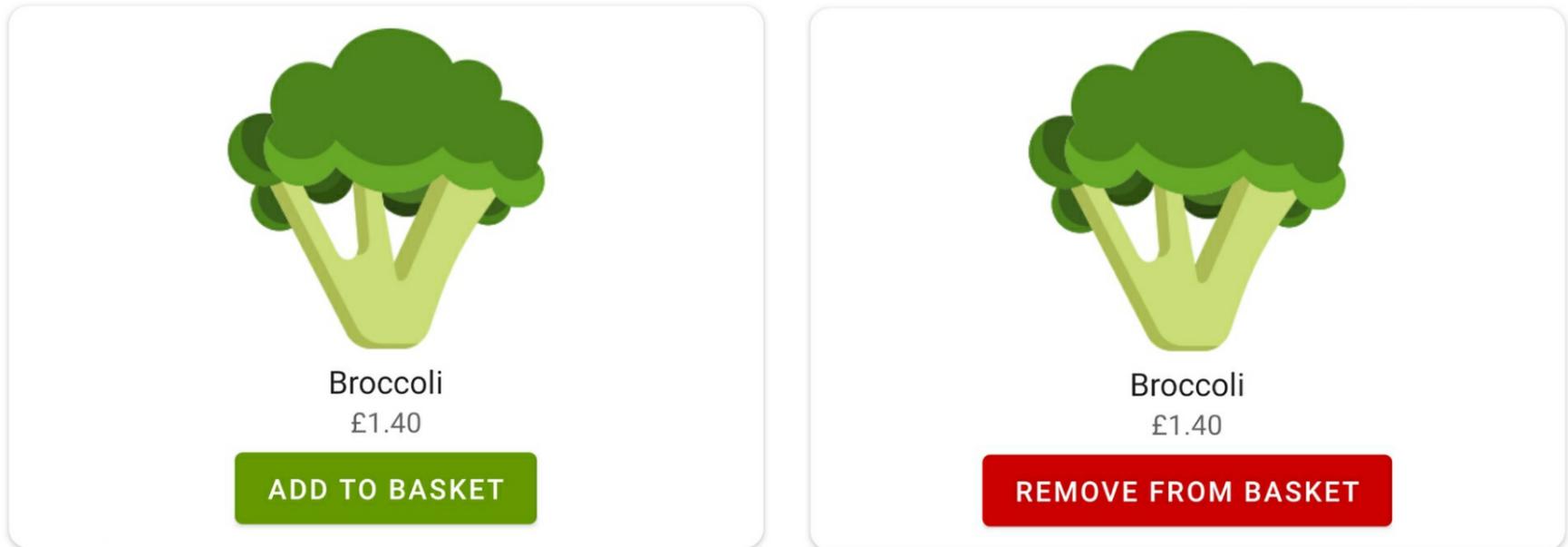
```
fun updateCart(index: Int) {
    val products = productsAdapter.products.toMutableList()
    products[index].inCart = !products[index].inCart
    productsAdapter.products = products

    // Call notifyItemChanged to update the add to basket button for that product
    productsAdapter.notifyItemChanged(index)
    storeViewModel.products.value = products
    storeViewModel.calculateOrderTotal()
}
```

The updateCart method requires an integer to be supplied as an argument. The integer will specify the index of the selected product within the overall list of products. Once the product index has been supplied, the updateCart method toggles the inCart field for the selected product from true to false or vice versa. This process effectively adds or removes the product from the customer's shopping cart. Next, the updated list of products is uploaded to the ProductsAdapter and StoreViewModel classes. A StoreViewModel method that we will define later called calculateOrderTotal will then calculate the new total price of the customer's order.

The above code uses the adapter's notifyItemChanged method to refresh the corresponding RecyclerView item for the user's selected product. Refreshing the RecyclerView item means the add to basket button will update to either

display “Add to basket” or “Remove from basket” as appropriate, to reflect the product having been added to or removed from the user’s shopping basket.



## Using the currency exchange rate API

The app will allow the customer to shop in different currencies. If the customer changes the currency, the app will retrieve the live exchange rate from the store’s base currency to their selected currency and update all the product prices accordingly. The currency exchange rate API we will use in this project can be found on this website: <https://api.exchangerate-api.com/>. To use the API, you must first create a free account by completing the sign-up form: <https://app.exchangerate-api.com/sign-up>. Free accounts can submit 1500 API calls a month, which is sufficient for this example project. Once you have signed up, log in to your account dashboard (<https://app.exchangerate-api.com/dashboard>) and make a note of your API key.

### API Access

Your API Key: **002**  
Example Request: <https://v6.exchangerate-api.com/v6/002>

Let’s now write the code that handles changes in the currency used in the store. Return to Android Studio and create a new Kotlin class by right-clicking the directory that contains MainActivity and selecting **New > Kotlin Class/File**. Name the file Currency and select Data Class from the list of options. Once the **currency.kt** file opens in the editor, modify its code so it reads as follows:

```
data class Currency(  
    var code: String,  
    var symbol: String,  
    // If the customer is using the base currency of the store then exchangeRate will be null  
    var exchangeRate: Double?  
)
```

The Currency data class will store information about the active currency. Specifically, it will store the three-digit ISO code that identifies the currency (see the following web page for a list of ISO codes supported by the currency exchange API <https://www.exchangerate-api.com/docs/supported-currencies>), the symbol that represents the currency (e.g. £ for GBP) and the exchange rate (if applicable).

Once the Currency data class is in place, open the **MainActivity.kt** file and add the following variables to the top of the class:

```
// TODO: put the ISO code for your store's base currency as the value of the defCurrency variable  
private val defCurrency = "GBP"  
private var exchangeData: JSONObject? = null  
private var selectedCurrency: Currency? = null  
private lateinit var sharedPreferences: SharedPreferences
```

The `defCurrency` variable in the above code will contain the three-digit ISO code for the base currency of the store. Currently, the base currency is Great British Pounds, but you may need to change the ISO code if your store will use a different base currency. A full list of accepted ISO codes can be found on the exchange rate API website: <https://www.exchangerate-api.com/docs/supported-currencies>. The other variables will store the JSON output the app receives following an exchange rate API request, the Currency object associated with the currently active currency, and a SharedPreferences class that we will use to record the user's currency preference.

To initialise the `sharedPreferences` variable and request exchange rate data when the app is launched, add the following code to the bottom of the `onCreate` method:

```
sharedPreferences = PreferenceManager.getDefaultSharedPreferences(this)
getCurrencyData()
```

Note you may need to add the following import statement to the top of the file:

```
import androidx.preference.PreferenceManager
```

Exchange rate data will be retrieved by a method called `getCurrencyData`. To define the `getCurrencyData` method, add the following code below the `onCreate` method:

```
private fun getCurrencyData(): JSONObject? {
    val client = AsyncHttpClient()

    // TODO: Replace YOUR-API-KEY-HERE with your exchange rate API key
    client.get("https://v6.exchangerate-api.com/v6/YOUR-API-KEY-HERE/latest/$defCurrency", object :
    TextHttpResponseHandler() {
        override fun onSuccess(statusCode: Int, headers: Array<out Header>?, responseString: String?) {
            if (responseString != null) {
                exchangeData = JSONObject(responseString)
                val currencyPreference = sharedPreferences.getString("currency", defCurrency) ?: defCurrency
                setCurrency(currencyPreference)
            }
        }

        override fun onFailure(statusCode: Int, headers: Array<out Header>?, responseString: String?, throwable:
        Throwable?) {
            Toast.makeText(this@MainActivity, resources.getString(R.string.exchange_data_unavailable),
            Toast.LENGTH_SHORT).show()
            setCurrency(defCurrency)
        }
    })

    return null
}
```

Note you may need to add the following import statement to the top of the file:

```
import cz.msebera.android.httpclient.Header
```

The `getCurrencyData` method builds an instance of the `AsyncHttpClient` class that will be used to communicate with the exchange rate API. The `AsyncHttpClient` object will request data via a URL that uses the ISO code associated with the store's base currency as the endpoint. Using the store's base currency ISO code as the endpoint instructs the exchange rate API to return all the exchange rates associated with your store's default currency. Note you will need to replace the `YOUR-API-KEY-HERE` part of the URL with your API key from the exchange rate API website dashboard: <https://app.exchangerate-api.com/dashboard>. The `AsyncHttpClient` object will request data from the URL asynchronously, which means the app will process the request without blocking other application processes while it awaits a response.

To define the `setCurrency` method, add the following code below the `getCurrencyData` method:

```
private fun setCurrency(isoCode: String) {
    val exchangeRate = exchangeData?.getJSONObject("conversion_rates")?.getDouble(isoCode)

    // TODO: Define the base currency here
    var currency = Currency(defCurrency, "£", null)
    if (exchangeRate != null) {
        when (isoCode) {
```

```
// TODO: Define each additional currency your store supports here
```

```
"USD" -> currency = Currency(isoCode, "$", exchangeRate)
```

```
"EUR" -> currency = Currency(isoCode, "€", exchangeRate)
```

```
}
```

```
sharedPreferences.edit().apply {
```

```
    putString("currency", isoCode)
```

```
    apply()
```

```
}
```

```
selectedCurrency = currency
```

```
storeViewModel.currency.value = currency
```

```
storeViewModel.calculateOrderTotal()
```

```
}
```

The setCurrency method begins by determining the exchange rate from the store's base currency to the user's selected currency. This is achieved by retrieving the conversion\_rates JSON object from the exchange rate API output. An example of what the exchange rate API call output looks like when USD is the base currency is included below. By way of illustration, to retrieve the exchange rate from USD to AUD, the setCurrency method would search the conversion\_rates JSON object to find the value assigned to the AUD key, which in the below example is 1.4817. If exchange rate data is not available, then the value of the exchangeRate variable will be null.

```
{
    "result": "success",
    "documentation": "https://www.exchangerate-api.com/docs",
    "terms_of_use": "https://www.exchangerate-api.com/terms",
    "time_last_update_unix": 1585267200,
    "time_last_update_utc": "Fri, 27 Mar 2020 00:00:00 +0000",
    "time_next_update_unix": 1585353700,
    "time_next_update_utc": "Sat, 28 Mar 2020 00:00:00 +0000",
    "base_code": "USD",
    "conversion_rates": {
        "USD": 1,
        "AUD": 1.4817,
        "BGN": 1.7741,
        "CAD": 1.3168,
        "CHF": 0.9774,
        "CNY": 6.9454,
        "EGP": 15.7361,
        "EUR": 0.9013,
        "GBP": 0.7679,
        "...": 7.8536,
        "...": 1.3127,
        "...": 7.4722, etc. etc.
    }
}
```

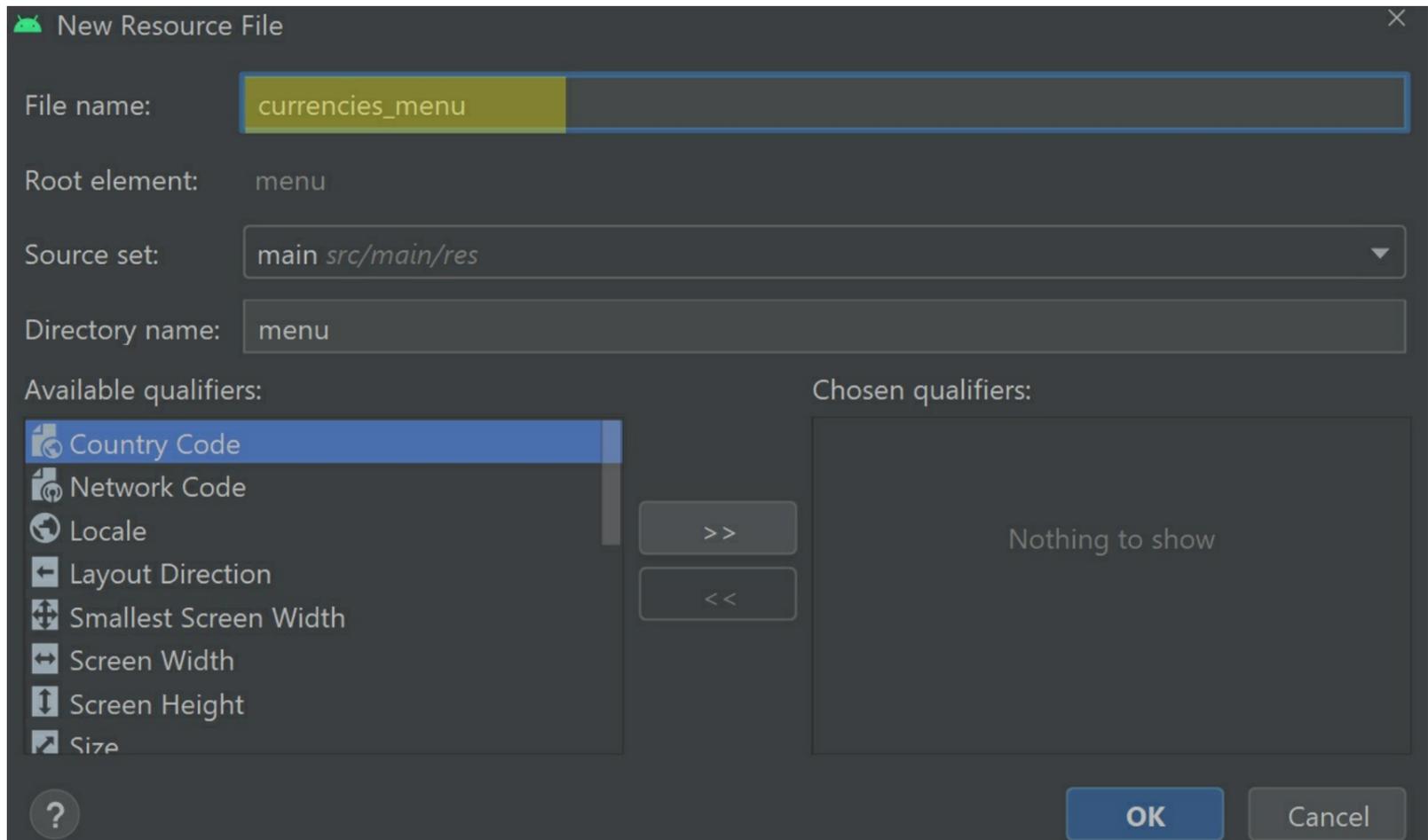
Once the exchange rate has been determined, the Currency object for the store's default currency is generated and assigned to a variable called currency. The value of the currency variable can be overwritten by a when block if the user has selected another currency and exchange rate data is available. The when block generates a corresponding Currency object for the user's selected currency based on its ISO code. You should add a separate option to the when block for each additional currency your store supports. For both the default and optional Currency objects, you should modify the currency symbol (e.g. '\$'), as appropriate.

The setCurrency method ends by writing the user's selected currency ISO code to the shared preferences file under a key called currency. In doing so, the user's currency preference can be retrieved whenever needed. Finally, the Currency object associated with the active currency is assigned to the MainActivity class's selectedCurrency variable and dispatched to the StoreViewModel view model so it can be used elsewhere in the app. To configure the view model to accept the Currency object, open the **StoreViewModel.kt** file and add the following variable to the top of the class:

```
var currency = MutableLiveData<Currency>()
```

## Changing the active currency

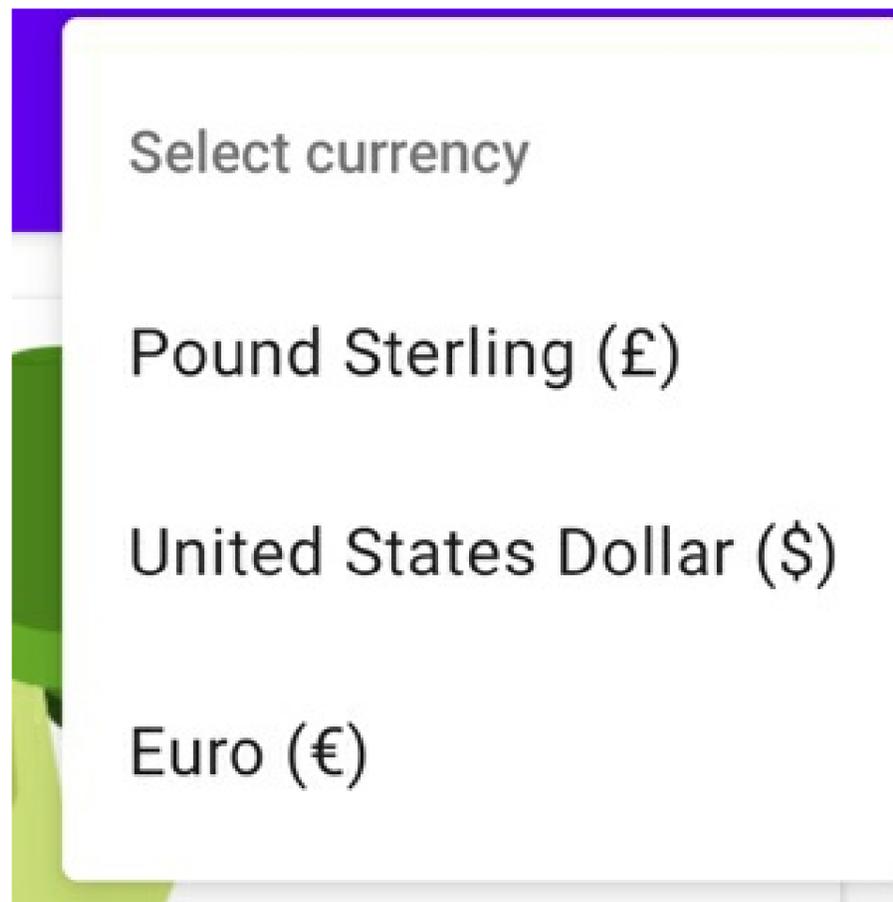
The user can change the currency using an overflow menu in the app toolbar. To design the overflow menu, create a menu resource file by right-clicking the **menu** folder (**Project** > **app** > **res**) and selecting **New** > **Menu Resource File**. Name the file `currencies_menu` then press OK.



Open the `currencies_menu.xml` file in Code view and add the following code inside the menu element:

```
<item android:title="@string/select_currency" >
  <menu>
    <!-- TODO: Create a menu item for each currency that your store supports -->
    <item
      android:id="@+id/currency_gbp"
      android:title="@string/currency_gbp" />
    <item
      android:id="@+id/currency_usd"
      android:title="@string/currency_usd" />
    <item
      android:id="@+id/currency_eur"
      android:title="@string/currency_eur" />
  </menu>
</item>
```

The above code defines a menu item that will display the text 'Select currency'. If the user clicks the item, then a submenu will expand showing all the available currencies. For each currency your store supports, you will need to define a separate menu item in the submenu. The above example code shows you how to create menu items for three currencies: GBP, USD and EUR.



To handle user interactions, return to the MainActivity class and add the following code below the onCreate method:

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {  
    menuInflater.inflate(R.menu.currencies_menu, menu)  
    return super.onCreateOptionsMenu(menu)  
}
```

The onCreateOptionsMenu method defined above uses the MenuInflater class to load the contents of the **currencies\_menu.xml** resource file and use it as an overflow menu for the app toolbar. The menu will be accessible across all fragments in the app (unless directed otherwise). To define what action should occur when a menu item is clicked, add the following code below the onCreateOptionsMenu method:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {  
    if (exchangeData == null) {  
        Toast.makeText(this, resources.getString(R.string.exchange_data_unavailable),  
            Toast.LENGTH_SHORT).show()  
        getCurrencyData()  
    } else {  
        when (item.itemId) {  
            // TODO: Configure each currency exchange menu item here  
            R.id.currency_gbp -> setCurrency("GBP")  
            R.id.currency_usd -> setCurrency("USD")  
            R.id.currency_eur -> setCurrency("EUR")  
        }  
    }  
    return super.onOptionsItemSelected(item)  
}
```

The above code defines a method called onOptionsItemSelected, which first checks whether the app has up to date exchange rate data. If the data is not available, then a toast notification will ask the user to try again later. The getCurrencyData method will then attempt to request data from the exchange rate API. On the other hand, if exchange rate data is available, then a when block will send the corresponding ISO code for the user's selected currency to the setCurrency method. The setCurrency method will then generate the appropriate Currency object and upload it to the StoreViewModel view model. Remember to add a separate entry to the when block for each currency your store supports, as shown above.

## Updating the prices of products

The application will need to update the product prices whenever the user changes the currency. To handle the price updates, we will instruct the ProductsFragment class to register an observer on the StoreViewModel view model's

currency variable so that the fragment will be notified whenever the active Currency object updates. The ProductsFragment class can use the value of the Currency object's exchangeRate field to update the price of each product. Open the **ProductsAdapter.kt** file (**Project > app > java > name of the project > ui > products**) and add the following variable below the products variable at the top of the class:

```
var currency: Currency? = null
```

Note you may need to add the following import statement to the top of the file, replacing the com.example.store part if your app uses a different package name. The package name of your app can be found in line 1 of the file:

```
import com.example.store.Currency
```

The currency variable will store the Currency object that the adapter will extract the exchange rate from. To calculate any necessary price adjustments, locate the adapter's onBindViewHolder method and replace the TODO comment with the following code:

```
val price = if (currency?.exchangeRate == null) current.price  
else current.price * currency?.exchangeRate!!
```

```
holder.mProductPrice.text = activity.resources.getString(R.string.product_price, currency?.symbol,  
String.format("%.2f", price))
```

The above code multiplies the price of each product by the exchange rate of the Currency object and assigns it to a variable called price. If the exchange rate is null then no price adjustment will occur. The updated price and symbol from the Currency object are then loaded into the productPrice TextView widget and displayed to the user.

To transfer the Currency object from the view model to the ProductsAdapter class, open the ProductsFragment class and add the following code to the bottom of the onCreateView method:

```
storeViewModel.currency.observe(viewLifecycleOwner, { currency ->  
    currency?.let {  
        binding.productsRecyclerView.visibility = View.VISIBLE  
        binding.loadingProgress.visibility = View.GONE  
        // Detect whether the selected currency different than the currency currently being used  
        if (productsAdapter.currency == null || it.symbol != productsAdapter.currency?.symbol) {  
            productsAdapter.currency = it  
            productsAdapter.notifyItemRangeChanged(0, productsAdapter.itemCount)  
        }  
    }  
})
```

The above code registers an observer on the StoreViewModel view model's currency variable. The observer will detect whenever the value of the currency variable changes. When the app is first launched, the Currency object will be null. Also, the RecyclerView will not be visible and a loading symbol will appear instead. However, once a Currency object has been constructed, either for the store's base currency or the user's selected currency, the observer will retrieve the Currency object, hide the loading symbol and reveal the RecyclerView. The RecyclerView will contain the catalogue of products available in the store and the adapter will adjust their price according to the customer's selected currency. Whenever the observer transfers a new Currency object to the adapter, it calls the adapter's notifyItemRangeChanged method to refresh the products in the RecyclerView and prompt the adapter to recalculate their prices.

## Setting up the Checkout fragment and layout

The user will be able to view all the products in their shopping basket and pay for their order in a dedicated checkout fragment. The checkout fragment will require a layout. To create the layout navigate through **Project > app > res** then right-click the **layout** folder and select **New > Layout Resource File**. Name the file **fragment\_checkout** then click OK. Edit the code in the layout file so it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>  
<androidx.constraintlayout.widget.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" >  
  
<androidx.recyclerview.widget.RecyclerView
```

```

    android:id="@+id/cartRecyclerView"
    android:layout_height="0dp"
    android:layout_width="match_parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toTopOf="@id/orderTotal" />

```

```

<TextView
    android:id="@+id/emptyCart"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/empty_cart"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintBottom_toBottomOf="parent" />

```

```

<TextView
    android:id="@+id/orderTotal"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="12dp"
    android:layout_marginBottom="8dp"
    android:textSize="16sp"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintBottom_toTopOf="@id/payButton" />

```

```

<Button
    android:id="@+id/payButton"
    android:text="@string/pay_using_paypal"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="6dp"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintBottom_toBottomOf="parent" />

```

```

</androidx.constraintlayout.widget.ConstraintLayout>

```

The root element of the `fragment_checkout` layout is a `ConstraintLayout` widget. The `ConstraintLayout` coordinates the following widgets: a `RecyclerView`, which will display a breakdown of the user's shopping basket; a `TextView`, that will inform the user if their shopping basket is empty; another `TextView`, that will display the total price of the order; and a `Button`, that will allow the user to pay using PayPal. The `RecyclerView` widget will display a summary of each product in the user's shopping basket. To initialise the `RecyclerView` widget, locate and open the **CheckoutFragment.kt** file by navigating through **Project > app > java > name of the project > ui > checkout**. Edit the list of variables at the top of the class so they read as follows:

```

private var _binding: FragmentCheckoutBinding? = null
private val storeViewModel: StoreViewModel by activityViewModels()
private lateinit var callingActivity: MainActivity

```

Note you may need to add the following import statement to the top of the file:

```

import androidx.fragment.app.activityViewModels

```

The above variables will allow the fragment to access the contents of the **fragment\_checkout.xml** layout via the layout's binding class, and interact with the `StoreViewModel` and `MainActivity` classes. To initialise the variables, edit the `onCreateView` method so it reads as follows:

```

override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
): View {
    _binding = FragmentCheckoutBinding.inflate(inflater, container, false)
}

```

```
callingActivity = activity as MainActivity
```

```
return binding.root
```

The above code creates a binding instance between the CheckoutFragment class and the **fragment\_checkout.xml** layout via the layout's FragmentCheckoutBinding binding class. The fragment can use the binding class to interact with the layout and its widgets.

## Preparing the Checkout adapter

In this section, we will load the contents of the user's shopping basket into the RecyclerView widget from the **fragment\_checkout.xml** layout. To facilitate this, we first need to create a layout that will hold information about each product. Create a new layout resource file by right-clicking the **layout** directory (found by navigating **Project** > **app** > **res**) and selecting **New** > **Layout Resource File**. Name the layout `basket_product` then press OK. Once the layout opens in the editor, switch it to Code view and edit the file so it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<androidx.cardview.widget.CardView 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="wrap_content"
    android:layout_margin="18dp"
    app:cardCornerRadius="10dp"
    app:cardElevation="2dp">
```

```
<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center">
```

```
<ImageView
    android:id="@+id/productImage"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:contentDescription="@string/an_image_of_the_product"
    app:layout_constraintDimensionRatio="1:1"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="@id/productDetails"
    app:layout_constraintBottom_toBottomOf="@id/productDetails" />
```

```
<LinearLayout
    android:id="@+id/productDetails"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginHorizontal="6dp"
    android:orientation="vertical"
    app:layout_constraintEnd_toStartOf="@id/removeFromBasketButton"
    app:layout_constraintStart_toEndOf="@id/productImage"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toBottomOf="parent">
```

```
<TextView
    android:id="@+id/productName"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textColor="@color/material_on_surface_emphasis_high_type"
    android:textSize="16sp" />
```

```
<TextView
    android:id="@+id/productPrice"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textColor="@color/material_on_surface_emphasis_medium"
```

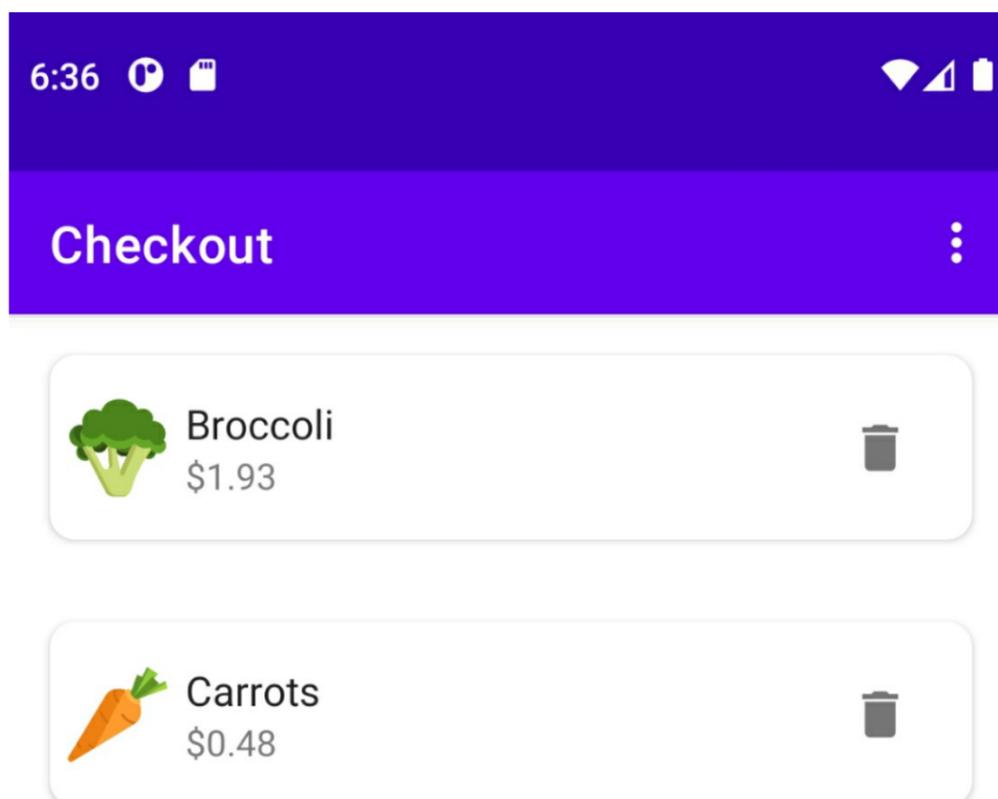
```

        android:textSize="14sp" />
    </LinearLayout>

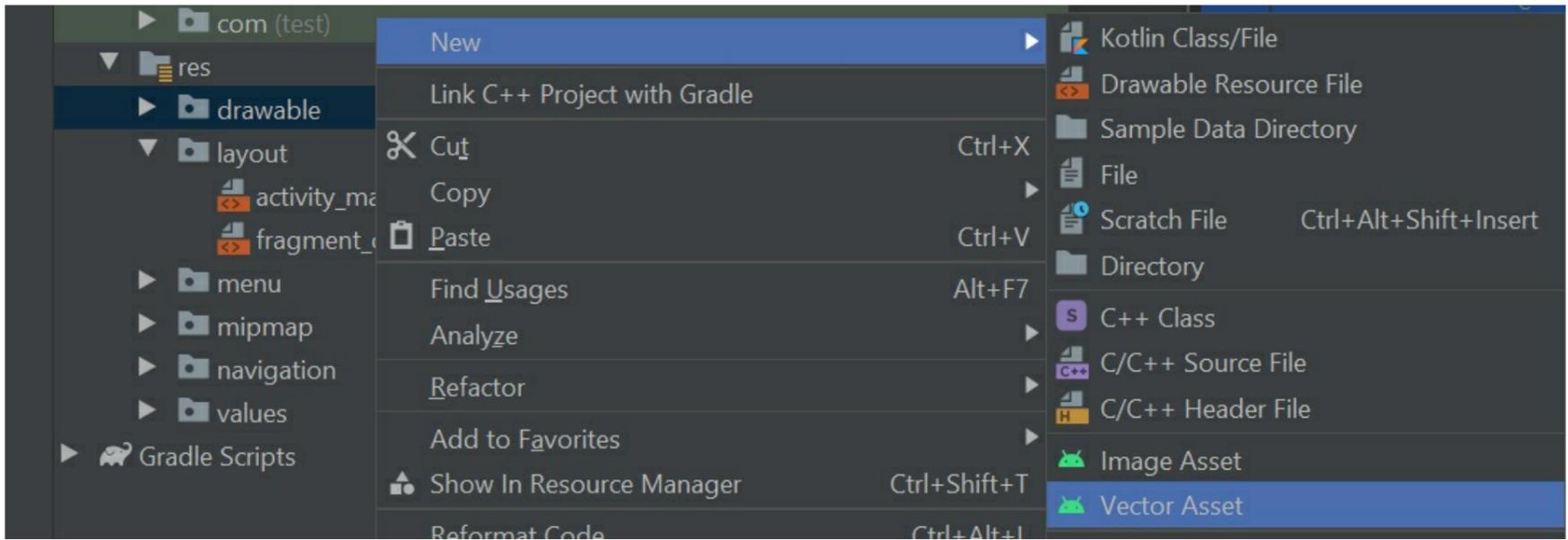
    <ImageButton
        android:id="@+id/removeFromBasketButton"
        android:layout_width="60dp"
        android:layout_height="60dp"
        android:src="@drawable/ic_delete"
        android:contentDescription="@string/remove_from_basket"
        android:backgroundTint="@android:color/transparent"
        android:foreground="?attr/selectableItemBackground"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
</androidx.cardview.widget.CardView>

```

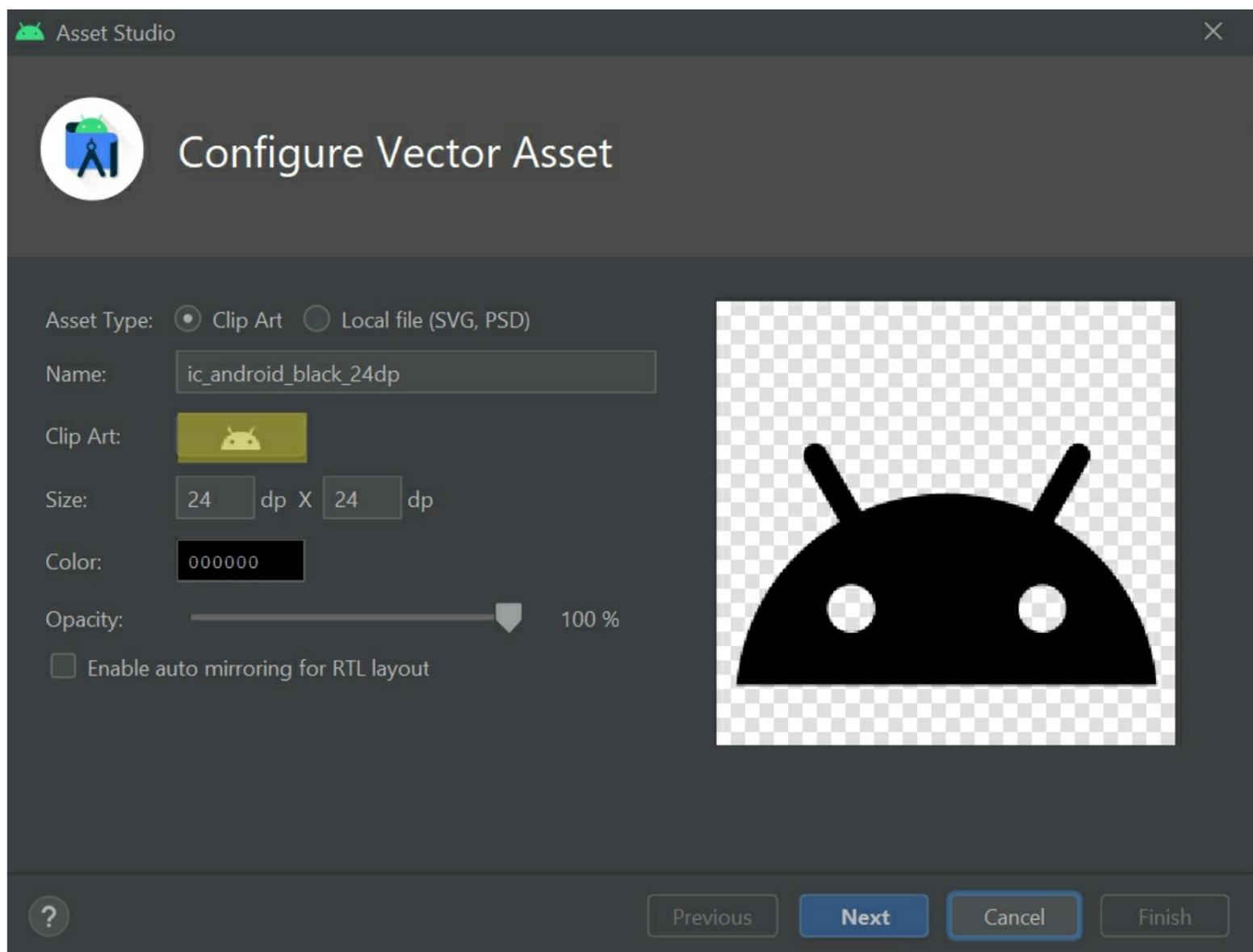
The root element of the `basket_product` layout is a `CardView` widget that will elevate the layout above the backdrop of the `RecyclerView`. The `CardView` widget also features a 10dp corner radius that will round the corners of the layout. Inside the `CardView` widget, a `ConstraintLayout` widget will coordinate the other widgets in the layout. The first widget is an `ImageView` that will display an image of the product. The `ImageView` widget has a `constraintDimensionRatio` of 1:1, which means the width of the image will equal its height and the image will appear as a square.



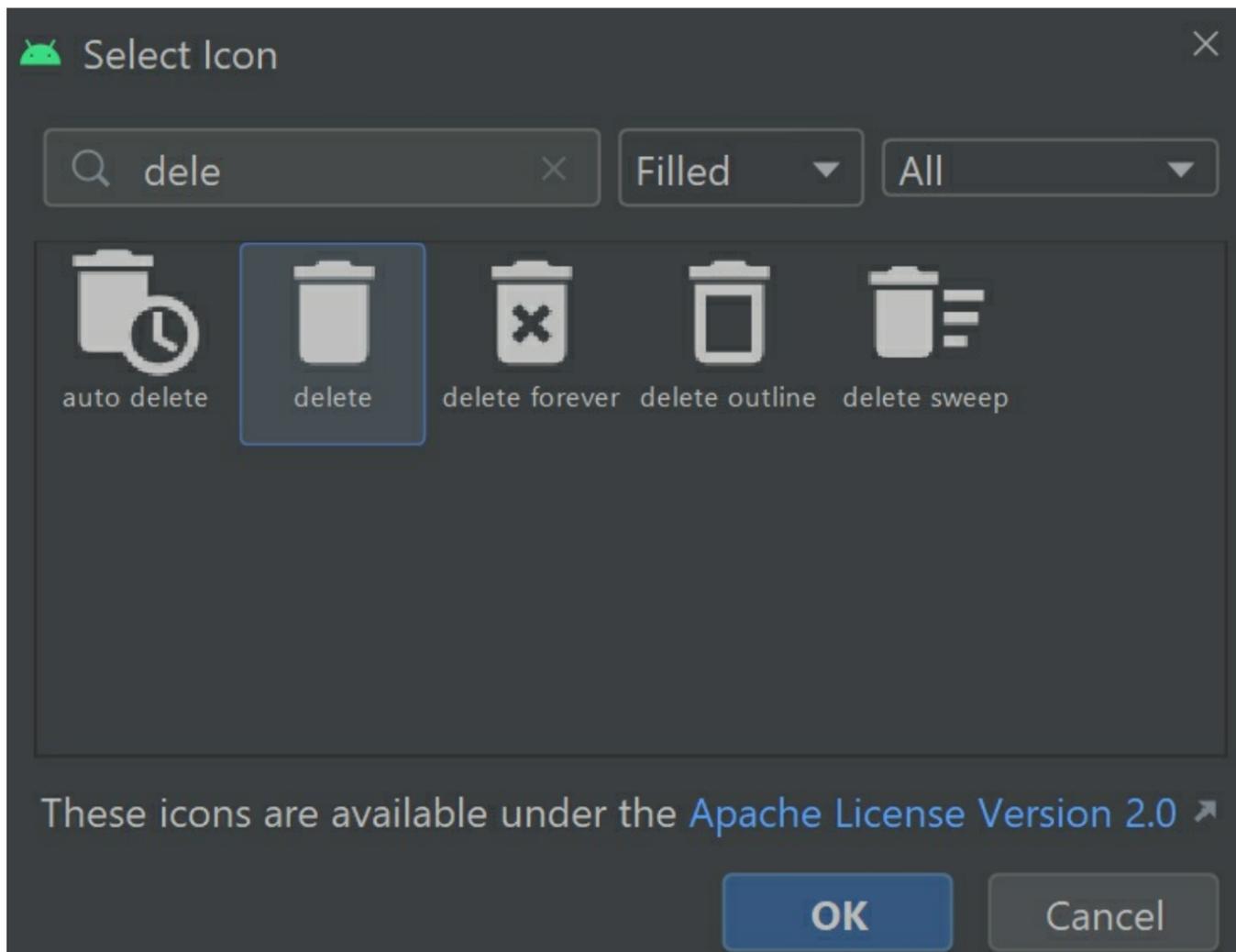
To the right-hand side of the product image, there is a vertically oriented `LinearLayout` widget containing two `TextView` widgets. The `TextView` widgets will display information about the name and price of the product. The `LinearLayout` is constrained so that it will sit between the product image on the left side of the layout and an `ImageButton` widget on the right side of the layout. The `ImageButton` will contain an image of a bin icon that will allow the user to remove the product from their shopping basket. We need to manually create the icon that will be used for the button. To do this, right-click the **drawable** folder (**project** > **app** > **res**) then select **New** > **Vector Asset**.



In the Asset Studio window, click the image of the Android next to the phrase Clip Art.



In the Select Icon window, search for and select the delete icon then click OK.



When you return to the Asset Studio window, set the name to `ic_delete` then press Next followed by Finish to save the icon.

Moving on, let's create the adapter class that will coordinate the product summaries in the checkout fragment. Right-click the **checkout** directory then select **New > Kotlin Class/File**. Name the file `CheckoutAdapter` and select Class from the list of options. Once the **CheckoutAdapter.kt** file opens in the editor, edit the class's code so it reads as follows:

```
// TODO: Note you may need to replace the 'com.example.store' part of the below import statement to reflect your
app's project declaration as specified on line 1
import com.example.store.Currency

class CheckoutAdapter(private val activity: MainActivity, private val fragment: CheckoutFragment) :
RecyclerView.Adapter<CheckoutAdapter.ProductsViewHolder>() {
    var products = mutableListOf<Product>()
    var currency: Currency? = null

    inner class ProductsViewHolder(itemView: View) :
        RecyclerView.ViewHolder(itemView) {

        internal var mProductImage = itemView.findViewById<View>(R.id.productImage) as ImageView
        internal var mProductName = itemView.findViewById<View>(R.id.productName) as TextView
        internal var mProductPrice = itemView.findViewById<View>(R.id.productPrice) as TextView
        internal var mRemoveFromBasketButton = itemView.findViewById<View>(R.id.removeFromBasketButton)
as ImageButton
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ProductsViewHolder {
        return ProductsViewHolder(LayoutInflater.from(parent.context).inflate(R.layout.basket_product, parent, false))
    }

    override fun onBindViewHolder(holder: ProductsViewHolder, position: Int) {
        val current = products[position]

        Glide.with(activity)
            .load(current.image)
            .transition(DrawableTransitionOptions.withCrossFade())
            .centerCrop()
            .override(400, 400)
    }
}
```

```

        .into(holder.mProductImage)

        holder.mProductName.text = current.name
        val price = if (currency?.exchangeRate == null) current.price
        else current.price * currency?.exchangeRate!!

        holder.mProductPrice.text = activity.resources.getString(R.string.product_price, currency?.symbol,
String.format("%.2f", price))

        holder.mRemoveFromBasketButton.setOnClickListener {
            fragment.removeProduct(current)
        }
    }
}

override fun getItemCount() = products.size
}

```

The CheckoutAdapter class defined above contains parameters called activity and fragment in its primary constructor. The parameters will allow the adapter to access the data and methods in the MainActivity and CheckoutFragment classes, respectively. In the body of the adapter, two variables called products and currency are defined, which will be populated with the list of Product objects and the currently active Currency object from the StoreViewModel class. Next, an inner class called ProductsViewHolder is established. The inner class will assign the **basket\_product.xml** layout's widgets to variables so they can be accessed elsewhere in the adapter. The adapter knows to use the **basket\_product.xml** layout to display items in the RecyclerView because this is the layout that is inflated by the onCreateViewHolder method.

The next method in the adapter class is called onBindViewHolder, which determines how data is displayed at each position in the RecyclerView. It does this by finding the corresponding Product object in the products list for the current position in the RecyclerView. Next, an image loading framework called Glide retrieves the drawable resource referenced in the Product object's image parameter and displays it in the productImage ImageView. The code directs Glide to use a fade animation when loading the image into the ImageView, crop the image if necessary, and set the image's dimensions to 400 pixels \* 400 pixels.

The name of the product is loaded into the productName TextView. Next, a variable called price is defined that will contain the result of the price of the product multiplied by the exchange rate stored in the Currency object. If the exchange rate is null then no price adjustment will occur. The output price and the currency symbol specified in the Currency object are then loaded into the productPrice TextView widget to display the price to the user. Finally, an onClick listener is attached to the removeFromBasket button. The listener will run a CheckoutFragment method called removeProduct (which we'll define shortly) and remove the product from the user's shopping basket. At the end of the adapter, there is a method called getItemCount, which calculates how many items will be loaded into RecyclerView. This value will equal the size of the products list.

The adapter is now set up, so let's integrate it with the CheckoutFragment class and apply it to the RecyclerView widget. To do this, open the **CheckoutFragment.kt** file and add the following variable to the list of variables at the top of the class to allow the fragment to access and interact with the CheckoutAdapter class:

```
private lateinit var checkoutAdapter: CheckoutAdapter
```

Next, add the following code below the onCreate method to direct the onCreateView stage of the fragment lifecycle to initialise the CheckoutAdapter instance and apply it to the RecyclerView:

```

override fun onCreateView(view: View, savedInstanceState: Bundle?) {
    super.onCreateView(view, savedInstanceState)

    checkoutAdapter = CheckoutAdapter(callingActivity, this)
    binding.cartRecyclerView.layoutManager = LinearLayoutManager(activity)
    binding.cartRecyclerView.itemAnimator = DefaultItemAnimator()
    binding.cartRecyclerView.adapter = checkoutAdapter
}

```

The above code initialises the checkoutAdapter variable and supplies an instance of the MainActivity class and 'this' (in this case 'this' refers to the CheckoutFragment class) for the activity and fragment parameters in the CheckoutAdapter class's primary constructor. Next, the RecyclerView widget is configured to use a linear layout manager to stack its constituent items vertically one by one. Also, a default item animator is applied to the RecyclerView to provide some basic animations when items are added, removed or updated. Finally, the CheckoutAdapter instance is assigned to the RecyclerView to help manage its content.

## Displaying the user's shopping basket

The checkout fragment will display a summary of each Product object that has an `inCart` parameter set to true, which indicates that the user has added the product to their shopping cart. To facilitate this, add the following code to the bottom of the CheckoutFragment class's `onViewCreated` method:

```
storeViewModel.products.observe(viewLifecycleOwner, { products ->
    products?.let {
        val basket = it.filter { p ->
            p.inCart
        }

        if (basket.isEmpty()) binding.emptyCart.visibility = View.VISIBLE
        else binding.emptyCart.visibility = View.GONE

        val adapterBasket = checkoutAdapter.products
        when {
            basket.size > adapterBasket.size -> {
                val newProducts = basket - adapterBasket
                for (p in newProducts) {
                    checkoutAdapter.products.add(p)
                    checkoutAdapter.notifyItemInserted(checkoutAdapter.products.size - 1)
                }
            }
            basket.size < adapterBasket.size -> {
                val removedProducts = adapterBasket - basket
                for (p in removedProducts) {
                    val index = checkoutAdapter.products.indexOf(p)
                    checkoutAdapter.products.removeAt(index)
                    checkoutAdapter.notifyItemRemoved(index)
                }
            }
            adapterBasket.isEmpty() && basket.isNotEmpty() -> {
                checkoutAdapter.products = basket.toMutableList()
                checkoutAdapter.notifyItemRangeInserted(0, basket.size)
            }
            basket.isEmpty() -> {
                checkoutAdapter.notifyItemRangeRemoved(0, basket.size)
                checkoutAdapter.products = mutableListOf()
            }
        }
        updateOrderTotal()
    }
})
```

The above code registers an observer on the StoreViewModel view model's `products` variable, which means the fragment will be notified each time the list of Product objects changes. Whenever there is a change, the code creates a variable called `basket` that contains a filtered list of all Product objects that have an `inCart` property set to true. If the filtered list of products is empty, then a TextView widget featuring the text "Your cart is empty" will be displayed. Otherwise, the TextView widget will be hidden.

Next, a `when` block is used to determine how best to update the adapter and RecyclerView. If the new basket is larger than the existing basket, then the newly added Product objects are isolated and added to the end of the adapter's `products` list. The adapter's `notifyItemInserted` command is then used to prompt the RecyclerView to display the new product(s) to the user. In contrast, if the new basket is smaller than the existing basket, then the Product object(s) that the user removed from their basket are deleted from the adapter. The RecyclerView widget is informed of this change using the adapter's `notifyItemRemoved` method. If the adapter is empty when the list of products in the basket is received, then this suggests the fragment is newly opened. In which case, the full list of products in the basket is transferred to the adapter and the RecyclerView is updated using the `notifyItemRangeInserted` method, which displays the entire basket. Finally, if the customer's shopping basket is empty, then the `notifyItemRangeRemoved` method will clear any products that had been loaded into the RecyclerView and the adapter's `products` variable is set to an empty list.

If the user presses the remove from basket button for an item in their shopping basket, then a method in the CheckoutFragment class called removeProduct will run. To define the removeProduct method, add the following code below the onViewCreated method:

```
fun removeProduct(product: Product) {
    product.inCart = !product.inCart
    val products = storeViewModel.products.value?.toMutableList() ?: mutableListOf()
    val position = products.indexOf(product)
    if (position != -1) {
        products[position] = product
        storeViewModel.products.value = products
        storeViewModel.calculateOrderTotal()
    }
}
```

The removeProduct method features an argument called product that will contain the Product object that the user is attempting to remove from their shopping basket. The method switches the Product object's inCart property from true to false, then updates the list of Product objects held by the StoreViewModel view model accordingly. The observer that the fragment has registered on the view model's products list will detect the change and update the RecyclerView to reflect the removed item.

## Calculating the total price of the user's order

The total price of the user's order will be calculated by the MainActivity class and uploaded to the StoreViewModel view model so it can be accessed by the checkout fragment. To arrange this, open the **StoreViewModel.kt** file and add a variable called orderTotal to the top of the class. The orderTotal variable will store a MutableLiveData number in Double format and will equal 0.00 by default.

```
var orderTotal = MutableLiveData(0.00)
```

Next, let's define a method called calculateOrderTotal, which will recalculate the total price of the customer's order whenever the active Currency object (and hence exchange rate) changes, or the customer adds or removes items in their shopping basket. To define the calculateOrderTotal method, add the following code below the variables in the StoreViewModel class:

```
fun calculateOrderTotal() {
    val basket = products.value?.filter { p ->
        p.inCart
    } ?: listOf()

    var total = 0.00
    for (p in basket) total += p.price

    if (currency.value != null) total *= currency.value?.exchangeRate ?: 1.00
    // Use BigDecimal to round the orderTotal value to two decimal places
    orderTotal.value = BigDecimal(total).setScale(2, RoundingMode.HALF_EVEN).toDouble()
}
```

Note you may need to add the following import statement to the top of the file:

```
import java.math.BigDecimal
```

The calculateOrderTotal method filters the total list of Product objects to select the products that have an inCart value of true and have been added to the user's shopping basket. Next, a variable called total is defined. Initially, the total variable will be set to 0.00 but will increase as the price of each product in the user's shopping basket is added. Once the price of every product has been added together, the total price is multiplied by the exchange rate of the user's selected currency (if applicable). The value of the view model's orderTotal variable must be two decimal digits long to be accepted by the Braintree Payments API. To achieve this, the value of the total variable, which contains the new order total, is processed by the BigDecimal class's setScale method with the scale set to two decimal places and the rounding preference set to HALF\_EVEN. The fully processed total price is then assigned to the view model's orderTotal variable so it can be accessed from elsewhere in the app.

The HALF\_EVEN rounding preference rounds excess digits to the nearest even neighbour. For example, 21.343 will become 21.34 because 21.34 is the nearest even number. Alternative rounding strategies include ROUND\_UP and ROUND\_DOWN, which always round the

number up or down, respectively; however, these rounding strategies are not ideal when calculating prices because they are biased towards one direction. The ROUND\_EVEN strategy helps eliminate this bias because the price could go up or down depending on where the nearest even number is.

To configure the checkout fragment to receive the order total amount, return to the CheckoutFragment class and add the following variables to the top of the class:

```
private var amount: Double? = null
private var currency: Currency? = null
```

Note you may need to import the Currency data class:

```
import com.example.store.Currency
```

The amount variable defined above will store the total price of the order and the currency variable will store the active Currency object. To update these variables based on the data in the view model, add the following code to the bottom of the onCreateView method:

```
storeViewModel.orderTotal.observe(viewLifecycleOwner, {
    amount = it
    updateOrderTotal()
})

storeViewModel.currency.observe(viewLifecycleOwner, { c ->
    c?.let {
        currency = it
        // Detect whether the selected currency different than the currency currently being used
        if (checkoutAdapter.currency == null || it.symbol != checkoutAdapter.currency?.symbol) {
            checkoutAdapter.currency = it
            checkoutAdapter.notifyItemRangeChanged(0, checkoutAdapter.itemCount)
        }
        updateOrderTotal()
    }
})
```

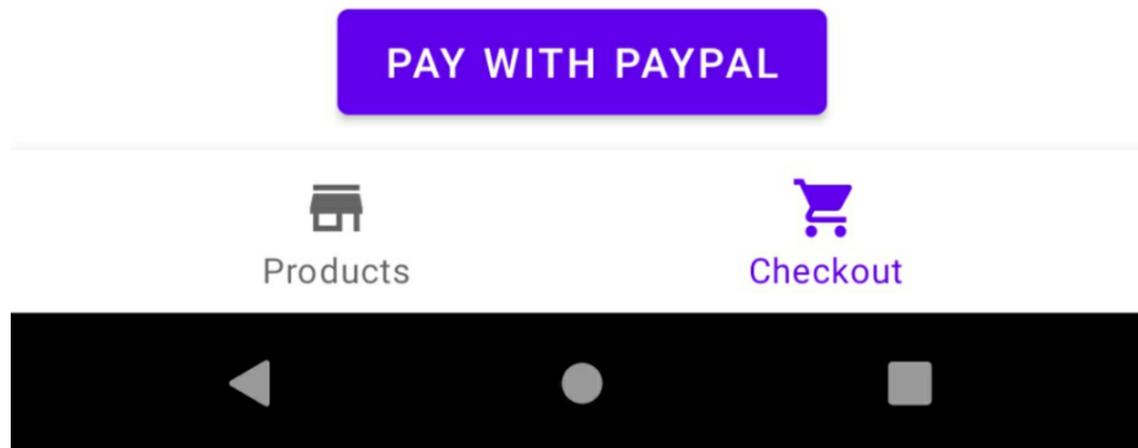
The above code registers observers on the StoreViewModel class's orderTotal and currency variables. The orderTotal observer will set the fragment's amount variable to the total price of the customer's order. It will also run a method called updateOrderTotal to update the price that is displayed in the orderTotal TextView in the fragment\_checkout layout. Meanwhile, the currency observer will transfer the active Currency object to the currency variable. The Currency object will also be delivered to the CheckoutAdapter class if it is different to the Currency object currently held by the adapter. Once that is done, the updateOrderTotal method will update the user interface because the new Currency object may require a different currency symbol to be displayed next to prices.

To define the updateOrderTotal method, add the following code below the removeProduct method:

```
private fun updateOrderTotal() {
    if (currency == null || amount == null) return
    val total = currency!!.symbol + String.format("%.2f", amount)
    binding.orderTotal.text = resources.getString(R.string.order_total, total)
}
```

The updateOrderTotal method starts by checking if either the currency or amount variables are null. The method requires both variables to have a value, so if the variables have not yet been initialised then a return command is used to exit the method. If both the currency and amount variables have been set, then a variable called total is defined that builds a string comprising the symbol for the active currency and the total order amount. The total order amount will be converted to a string using the "%.2f" format, which outputs a floating-point number with two digits after the decimal point. This is a necessary condition because otherwise trailing zeroes on prices such as £23.00 may be omitted. The total variable is input into the order\_total string resource to form a message such as "Order total: £23.43". Next, the compiled string is loaded into the fragment\_checkout layout's orderTotal TextView and displayed to the user.

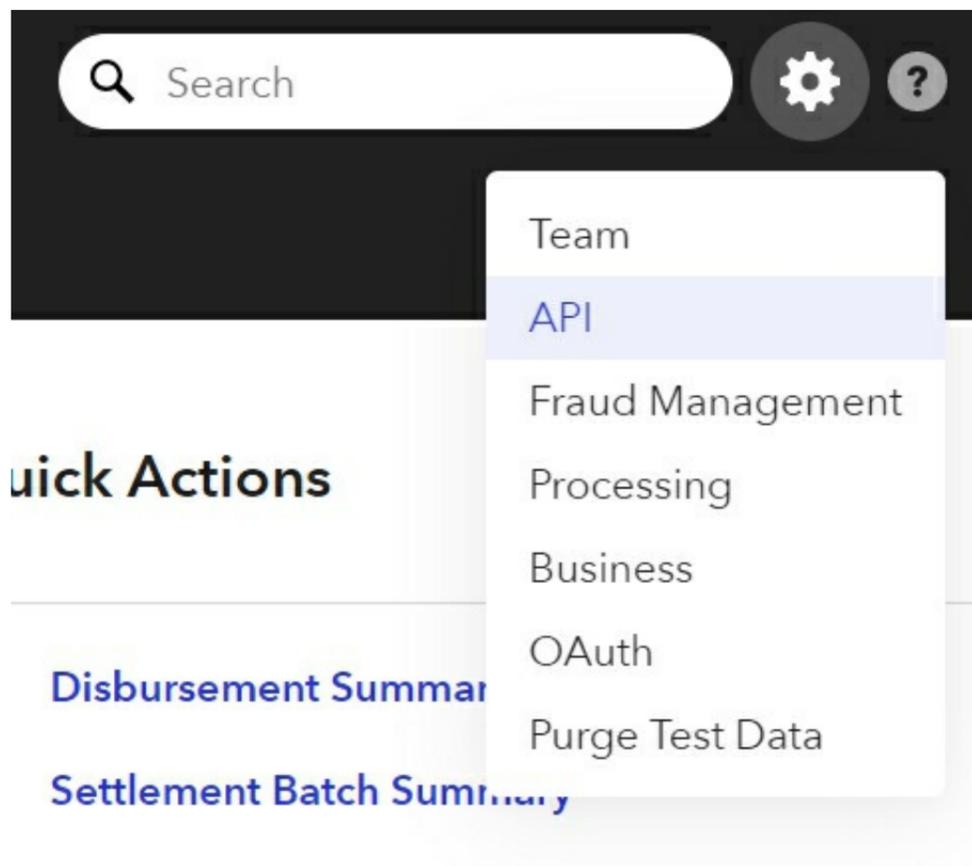
Order total: \$5.16



## Setting up the Braintree payments API

The app will use the Braintree payments API to authenticate payments and process transactions. To communicate with Braintree, you will need to create a Braintree account and register a tokenization key that will identify your application. There are two different types of Braintree accounts: Sandbox, which is used for testing your application and running mock transactions; and Production, which is used to process genuine transactions and handle real funds. Both accounts are distinct and use separate API keys. For now, you only need to create a Sandbox account. If you ultimately decide to release the app and process genuine payments then you can create a Production account also (more on this in the section on Going Live).

To create a Braintree Sandbox account, visit the following URL and complete the sign-up form: <https://www.braintreepayments.com/sandbox>. Once you have logged in, click the settings cog in the top-right corner and select API.



Next, locate and copy the tokenization key. Note you may need to press the Generate New Tokenization Key button if a tokenization key does not already exist.

### Tokenization Keys

For use in Braintree SDKs

Key	Label	Last Used On	Status	Actions
sandbox_8ht6		2021-01-09	Active	<a href="#">Edit</a>

[+ Generate New Tokenization Key](#)

To insert the tokenization key to the project, return to Android Studio and open the **MainActivity.kt** file (**Project > app > java > name of your project**). Add the following companion object above the onCreate method:

```
companion object {  
    // TODO: Replace the value of the below variable with your Sandbox/Production Braintree tokenization key  
    private const val TOKENIZATION_KEY = "YOUR-TOKENIZATION-KEY"  
}
```

Replace YOUR-TOKENIZATION-KEY with the tokenization key you copied from your Sandbox Braintree account. The tokenization key provides a useful mechanism for processing one-off transactions; however, Braintree restricts certain payment pathways and customer preferences because the tokenization key is static and can be used across an indefinite number of installations of the app (see the Braintree documentation for more details: <https://developer.paypal.com/braintree/docs/guides/authorization/tokenization-key>). For this reason, we will also discuss an alternative payment authentication method involving generating a new client token on a session-by-session basis. The client token authentication method has access to more of Braintree's client API capabilities than the tokenization key authentication method and so will be the preferred method for this app. It is still useful to have the tokenization key method available though in case there is a problem generating a client token.

Next, let's use the tokenization key to authenticate payments with Braintree and PayPal. To do this, add the following variables to the top of the MainActivity class:

```
private lateinit var braintreeClient: BraintreeClient  
private lateinit var paypalClient: PayPalClient
```

The above variables will contain instances of classes required to handle network requests with the Braintree Payments API and process PayPal transactions. To initialise the variables, add the following code to the bottom of the onCreate method:

```
braintreeClient = BraintreeClient(this, TOKENIZATION_KEY)  
paypalClient = PayPalClient(braintreeClient)
```

The above code initialises the BraintreeClient instance using the tokenization key that authorises your app to process payments. The BraintreeClient object is then applied to the PayPalClient object to enable the PayPal payment processing pathway.

## Linking your Braintree account with your PayPal business account

Braintree offers you the opportunity to link both its Sandbox and Production accounts with a PayPal business account that will receive funds following mock (Sandbox) or genuine (Production) PayPal transactions. To arrange this, you will require a PayPal business account: <https://www.paypal.com/business/getting-started>. Once you have set up a PayPal business account details, log in to the PayPal developer dashboard <https://developer.paypal.com/>. Next, navigate to the My Apps & Credentials page. This is where you'll create a PayPal app for your store so PayPal knows where to direct the funds when a customer makes a payment. Make sure you're viewing the page in Sandbox mode, then click 'Create app' in the 'REST API apps' section.

### My Apps & Credentials

Sandbox

Live

#### REST API apps

Get started quickly by using the Default Application credentials for testing PayPal REST APIs on the Sandbox environment.

App Name	Type	Actions
Default Application	REST	System generated, no actions available.

Create App

Name the app 'store' and leave the business account it is linked with unchanged. Once you submit those details you should be redirected to a page where you can view the credentials for the store app. Make a note of the Client ID and Secret and keep it somewhere safe because you will need those details again shortly. The Client ID and Secret are what we will use to communicate with PayPal's server and receive the order details following a purchase.

store

App display name: store 

SANDBOX API CREDENTIALS

Sandbox account

Client ID

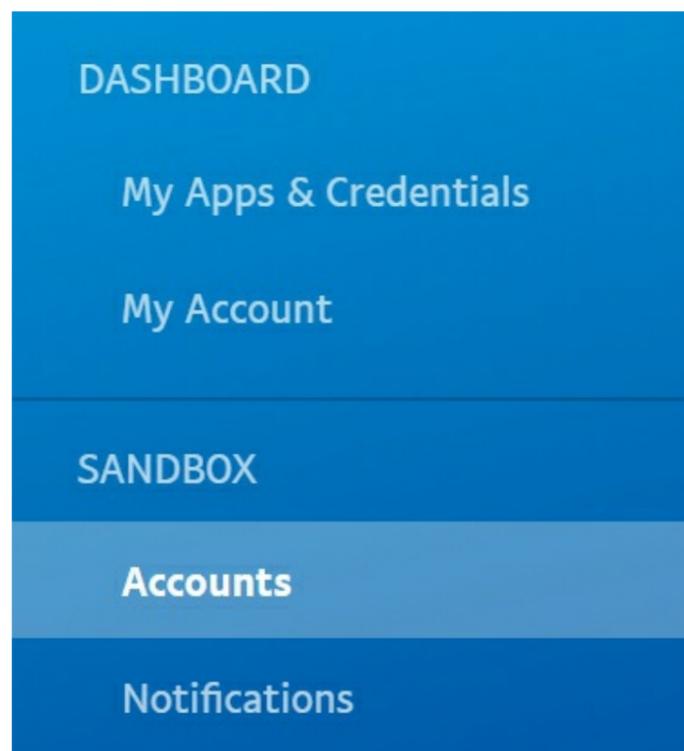
Secret  
[Hide](#)

**Note:** When you generate a new secret, you still maintain the original secret. The maximum number of client secrets is two. A client secret is either in enabled or disabled state.

Created	Secret	Status	Action
Feb 09, 2020		Enabled	

[Generate New Secret](#)

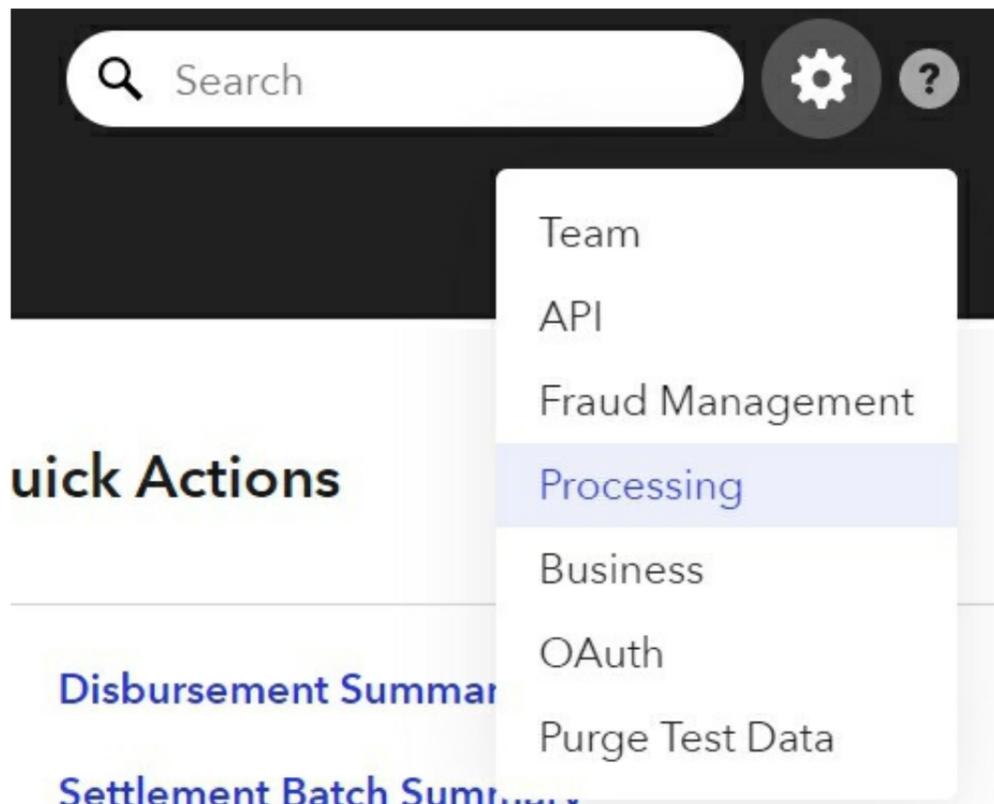
Next, navigate to the Accounts section.



In this section, you should see two sandbox accounts: one business and one personal. Under the 'Manage Accounts' column select 'View/Edit Account' for the personal account. An Account Details window should open that allows you to view the email address and system-generated password for the account. These are the details you will use to log in to PayPal when you are running a test transaction. It may also be a good idea to add lots of money to the account so you can make as many trial purchases as you like. To do this, click the Funding tab then press the Edit button. Set the balance to a large amount such as £100k then press Save.

You can log in to sandbox business and personal accounts as if they were real PayPal accounts. Simply visit <https://www.sandbox.paypal.com/signin>. This is a great way of checking how your store's/the customer's PayPal accounts will look following a transaction.

Once you have created the necessary accounts, make a note of your Sandbox PayPal app client ID and secret and return to your sandbox Braintree account: <https://sandbox.braintreegateway.com/>. Click the settings cog in the top right corner of the screen and navigate to the Processing section.



Ensure the PayPal item in the Payment Methods section is switched on then press 'Link Sandbox' to add the details of your Sandbox PayPal app.

## Payment Methods

**PayPal**

Enable customers to pay with PayPal

[Link Sandbox](#)

A **mocked PayPal flow** is enabled by default in the sandbox.

[Legal Agreements for PayPal Services](#)

Fill in the details of your PayPal app including the mock business email address linked with your sandbox PayPal app and the app's client ID and secret. Also, remember to press the Link PayPal Sandbox button to save any changes.

## PayPal Sandbox Credentials

By default, a mocked PayPal flow is used in the Braintree sandbox environment. It k

If you prefer to test the full functionality of your PayPal integration (transaction repc  
[testing guide](#) for full instructions on how to link the two accounts.

PayPal Email

Required

PayPal Client Id

Required

PayPal Client Secret

Required

Manage PayPal Disputes in the Braintree Control Panel

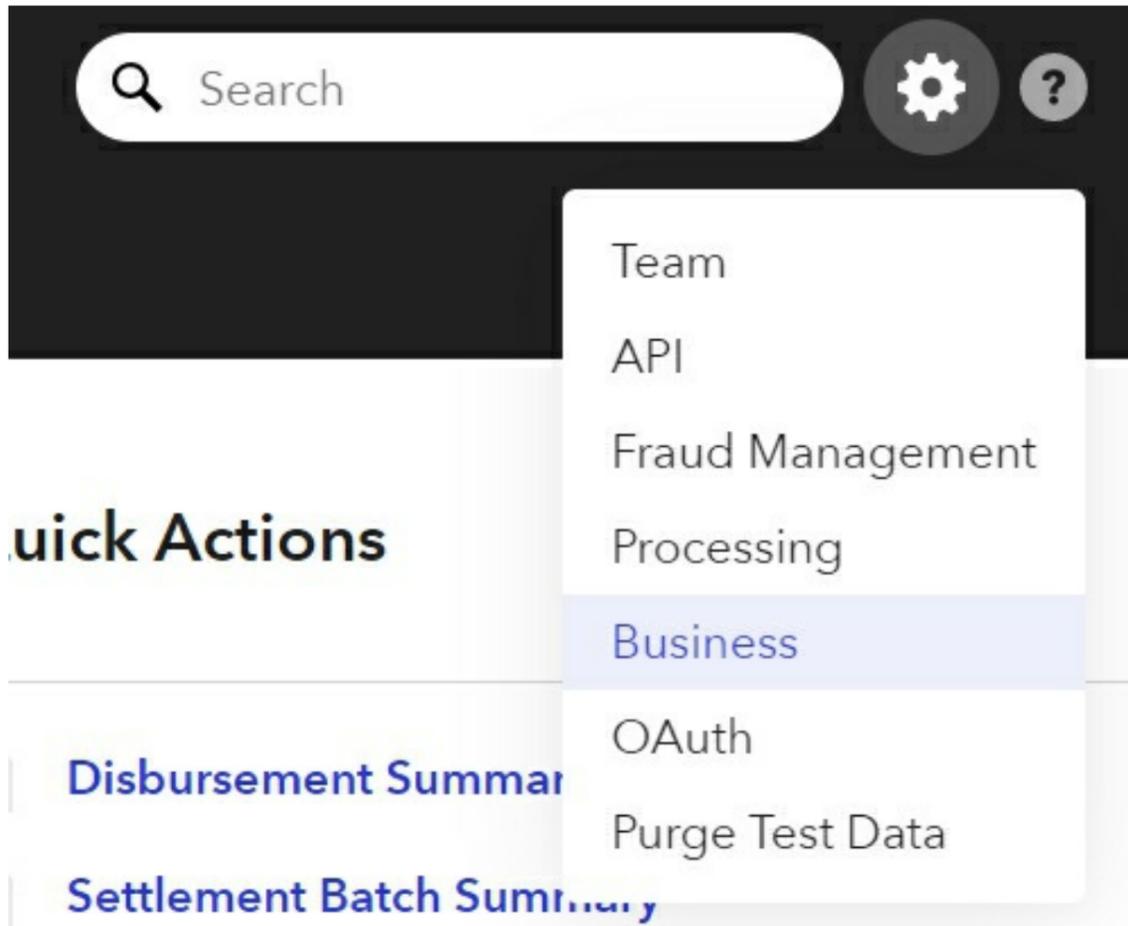
[See our support article on PayPal disputes.](#)

[Link PayPal Sandbox](#)

The Sandbox Braintree account and PayPal app are now linked and the funds from any transactions that are processed using PayPal will be sent to the business account linked with the PayPal app. These steps will need to be repeated for the Production Braintree account and Production PayPal app when you wish to begin processing genuine payments (more on this later!).

## Configuring your Braintree account to accept payment in multiple currencies

By default, your Braintree account will only accept payments in a single currency. This can lead to errors though if the user changes the store's currency. For example, they could be charged the incorrect amount and some payment methods such as PayPal may fail. To resolve this, you must manually configure your Braintree account to accept each additional currency your store will support. Log in to your Braintree account, click the cog in the top right corner and navigate to the Business section.



In the Merchant Accounts section, you should find your default Merchant Account ID and its associated currency. You must create a new Merchant Account for each additional currency your store will support. For example, if your default merchant account was associated with Great British Pounds (GBP) but you also wanted to process transactions in United States Dollars and Euros, then you must create Merchant Accounts that will process USD and EUR.

## Merchant ID

Your merchant ID is a unique identifier for your entire gateway account. This value is required to connect your API calls to the Braintree gateway.

## Merchant Accounts

Below is a list of payment methods and currencies you are currently accepting. The merchant account ID is a unique identifier for a specific merchant account in your gateway, and is used to specify which merchant account to use when creating a transaction.

[+ New Sandbox Merchant Account](#)

Merchant Account ID	Currency
<a href="#">codersguidebook</a> (default)	GBP

To create a new Merchant Account, click the New Sandbox Merchant Account button. Next, complete the New Sandbox Merchant Account form by assigning the merchant account an ID (such as `currency_usd`), leave the 'Make this my default merchant account' option unchecked, but check the 'Accept PayPal...' option so the merchant account can accept PayPal payments. Finally, select the appropriate underlying currency from the dropdown menu. For the `currency_usd` merchant account, select USD - United States Dollar.

## New Sandbox Merchant Account

Here you can add a new merchant account to test other currencies in the Sandbox. available in every region. You can read more about testing currencies in the [Sandbox API](#).

Merchant Account ID

Required

Make this my default merchant account.

Accept PayPal. Accepting PayPal limits the number of currencies you can choose from.

Save

Repeat this process to create another Merchant Account with an ID of `currency_eur` that accepts the currency EUR - Euro. Remember to make a note of all of the merchant account IDs (including the default ID) because you will need to refer to them later.

## Generating a client token

To access the full capabilities of the Braintree Payments API, the app will generate a client token each time it is launched. The client token is a JSON Web Token that provides access to Braintree's payment tools and authorises the app to prepare transactions. Your website is responsible for generating client tokens, and each token is valid for 24 hours. To begin the process of generating a client token, add the following line of code to the bottom of the MainActivity class's onCreate method:

```
getClientToken()
```

The above line of code runs a method called `getClientToken`, which will submit the request to your website server for the client token. To define the `getClientToken` method, add the following code below the `setCurrency` method:

```
private fun getClientToken() {  
    val client = AsyncHttpClient()  
    // TODO: Replace YOUR-DOMAIN.com with your website domain  
    client.get("https://YOUR-DOMAIN.com/store/client_token.php", object : TextHttpResponseHandler() {  
        override fun onSuccess(statusCode: Int, headers: Array<out Header>?, responseString: String?) {  
            braintreeClient = BraintreeClient(this@MainActivity, responseString ?: TOKENIZATION_KEY)  
            paypalClient = PayPalClient(braintreeClient)  
        }  
    })  
}
```

```
        override fun onFailure(statusCode: Int, headers: Array<out Header>?, responseString: String?, throwable:  
Throwable?) {  
            braintreeClient = BraintreeClient(this@MainActivity, TOKENIZATION_KEY)  
            paypalClient = PayPalClient(braintreeClient)  
        }  
    })  
}
```

The `getClientToken` method generates an instance of the `AsyncHttpClient` class, which is what we used to query the

currency exchange API. This time, however, we will query your website. Your website will attempt to generate a client token. If the request is successful, then the client token will be returned to the app in the onSuccess callback method as the responseString parameter. The above code uses the responseToken to reinitialise the BraintreeClient and PayPalClient objects with the client token instead of the tokenization key. However, if the responseString parameter is null or the request fails, then the tokenization key will be used. While it is preferable to initialise the BraintreeClient and PayPalClient objects using the client token because of the enhanced payment options and functionality such as remembering returning customer's details, the tokenization key is a useful fallback option. For a full comparison between the functionality offered by client tokens and tokenization keys, then see the authorisation overview on Braintree's website:

<https://developer.paypal.com/braintree/docs/guides/authorization/overview>

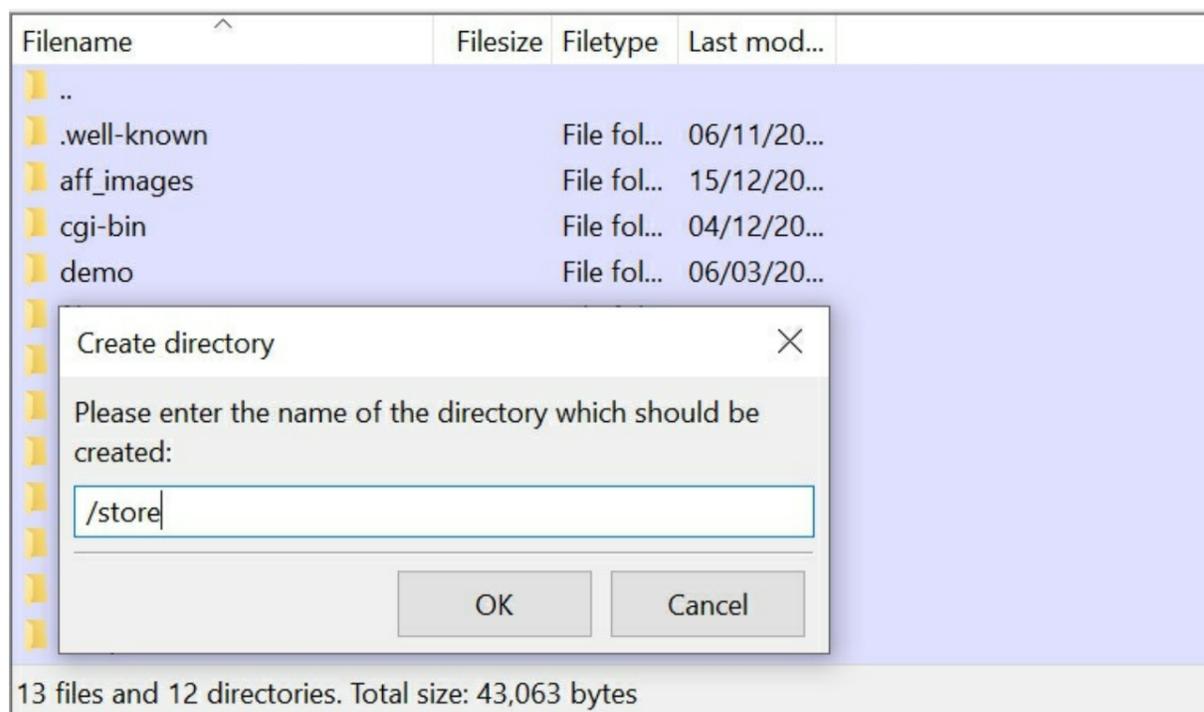
The client token will be generated by your website. If you do not have a website, then read this tutorial to get started: <https://codersguidebook.com/how-to-build-a-website/how-to-use-ftp-to-upload-website-files>. Next, edit the **YOUR-DOMAIN.com** part of the URL in the AsyncHttpClient object's get command to reflect your website's address. It is recommended to leave the **/store/client\_token.php** part of the URL unchanged because later tasks will advise you to upload files to a subdirectory called store. If you want to upload the files to an alternative folder and are confident doing so then feel free.

Client tokens will be generated by a file that you will upload to your website called **client\_token.php**. This file will use the Braintree PHP software development kit (SDK) to communicate with the Braintree Gateway and retrieve a client token. Your website will then deliver the client token to the Android app via the AsyncHttpClient object, which will assign the token to the MainActivity class's clientToken. If the website is unable to retrieve a client token then the value of the MainActivity class's clientToken variable will remain null and the app will attempt to use the tokenization key we discussed in an earlier section to authorise transactions instead; however, this is not preferable because the tokenization key has reduced privileges compared to the client token.

To upload files to your web server and create directories then you will need to use a file transfer protocol (FTP) client such as FileZilla (<https://filezilla-project.org/download.php?type=client>). If you have not used FileZilla before then this tutorial on our website will teach you everything you need to know:

<https://codersguidebook.com/how-to-build-a-website/how-to-use-ftp-to-upload-website-files>

First, log in to your web server through FileZilla (or your preferred FTP client) and create a new folder in the root directory of your website called store.



The store directory will contain the files required to interact with Braintree and process transactions. Backend server processing will be handled using a programming language called PHP. For the PHP code to communicate with the Braintree Gateway, you will need to upload the Braintree PHP SDK to your web server. The Braintree PHP SDK can be found in the lib folder in this Github repository [https://github.com/braintree/braintree\\_php](https://github.com/braintree/braintree_php). You are welcome to download the SDK from Github; however, for your convenience, the lib folder is also included in the example code that accompanies this book. To upload the SDK to your website, use your FTP client to navigate to the store directory you just created and upload the lib folder.

The **store** directory in the example code also contains a file called **client\_token.php** and its contents read as follows:

```
<?php
```

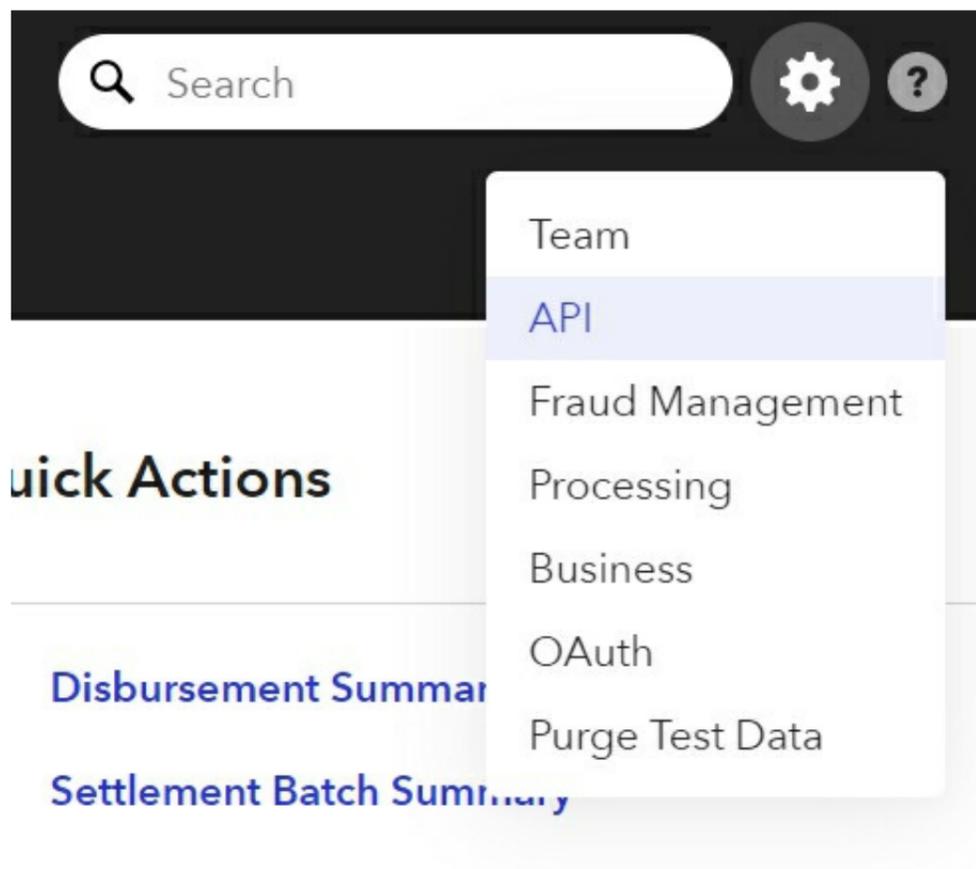
```

// Import the PHP library
require __DIR__ . '/lib/autoload.php';
use Braintree\Gateway;

$gateway = new Gateway([
    'environment' => 'sandbox',
    'merchantId' => 'MERCHANT-ID',
    'publicKey' => 'PUBLIC-KEY',
    'privateKey' => 'PRIVATE-KEY'
]);
$clientToken = $gateway->clientToken()->generate();
echo($clientToken);
?>

```

The above PHP code imports the contents of Braintree PHP SDK then creates an instance of the Gateway class. The Gateway class will authenticate your application with Braintree and request information such as the client token. For this reason, you will need to provide the information highlighted in blue above. The environment field can be set to either sandbox (for mock transactions) or production (for genuine transactions). The remainder of the information (merchantId, publicKey and privateKey) can be found by logging in to your sandbox or production Braintree account, clicking the settings cog in the top-right corner and selecting API.



The information you need to supply for the merchantId, publicKey and privateKey fields in the **client\_token.php** file can be found in the API Keys Client-Side Encryption Keys sections as shown in the screenshots below. Note, for the private key you will need to press the View button to reveal the key.

#### API Keys

Server use only, should not be shared.

Public Key	Private Key	Last Used On	Actions
	View	01/09/2021	Delete

+ Generate New API Key

## Client-Side Encryption Keys

For use in client SDKs prior to Braintree SDKs

Environment:	<input type="text" value="sandbox"/>	
Merchant ID:	<input type="text"/>	
CSE Key:	<input type="text"/>	
Never Used		

Once you have input the information required to build an instance of the Gateway object in the **client\_token.php** file, upload the completed file to the **store** directory alongside the lib folder.

Filename	Filesize	Filetype	Last mod...
..			
lib		File fol...	06/01/20...
 client_token.php	445	PHP File	07/01/20...

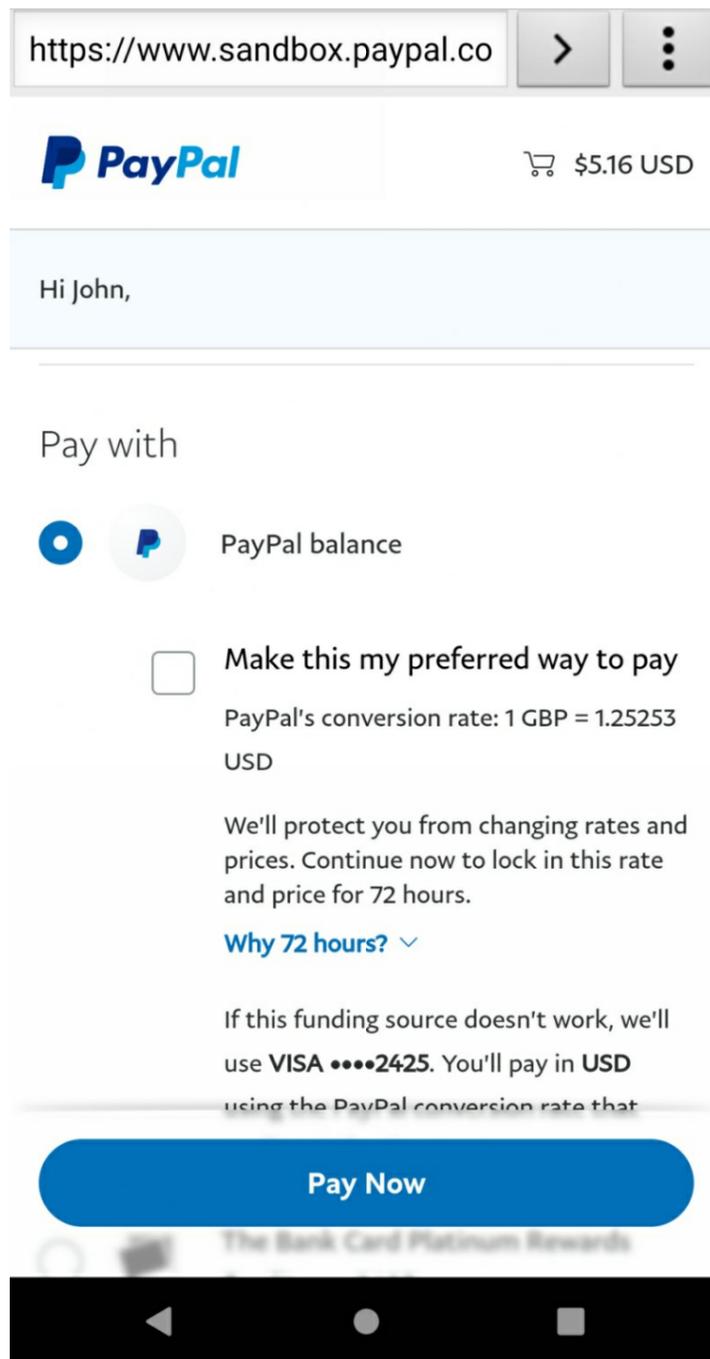
Your web server is now fully equipped to generate client tokens. The Store app simply needs to contact the **client\_token.php** file, which will use the Gateway object with your Braintree account's credentials to generate a client token and transmit the token back to the app using PHP's echo command.

## Initiating a PayPal transaction using Braintree

The user can pay for their order by clicking the payButton button in the **fragment\_checkout.xml** layout. To make the button active, add the following code to the bottom of the onCreateView method in the **CheckoutFragment.kt** file (**Project > app > java > name of the project > ui > checkout**):

```
binding.payButton.setOnClickListener {  
    callingActivity.initiatePayment()  
}
```

The onClick listener defined above runs a method in the MainActivity class called initiatePayment. The initiatePayment method will launch a web browser window that allows the user to pay using PayPal.



To define the `initiatePayment` method, open the `MainActivity.kt` file and add the following code below the `getClientToken` method:

```
fun initiatePayment() {
    if (storeViewModel.orderTotal.value == 0.00) return

    val orderTotal = storeViewModel.orderTotal.value.toString()
    // TODO - Save the total price to the share preferences file here

    val request = PayPalCheckoutRequest(orderTotal)
    request.currencyCode = selectedCurrency?.code ?: defCurrency
    request.userAction = USER_ACTION_COMMIT

    paypalClient.tokenizePayPalAccount(this, request) { error ->
        error?.let {
            Toast.makeText(this, getString(R.string.paypal_error, it.message), Toast.LENGTH_LONG).show()
        }
    }
}
```

Note you may need to add the following import statement to the top of the file:

```
import com.braintreepayments.api.PayPalCheckoutRequest.USER_ACTION_COMMIT
```

If the order total is 0.00, then the `initiatePayment` method will exit early using the `return` command because no payment is necessary and the user's shopping basket is likely empty. Otherwise, the method proceeds to construct a `PayPalCheckoutRequest` object that will enable checkout via PayPal. In constructing the `PayPalCheckoutRequest` object, the above code provides the order total amount, the ISO code for the currency that should be used in the transaction, and the user action. The user action dictates the call-to-action message when the user authorises a payment through PayPal. By default, PayPal will advise the user that there will be a further order confirmation for the customer to review their order before finalising the payment. However, in our app, the customer will already

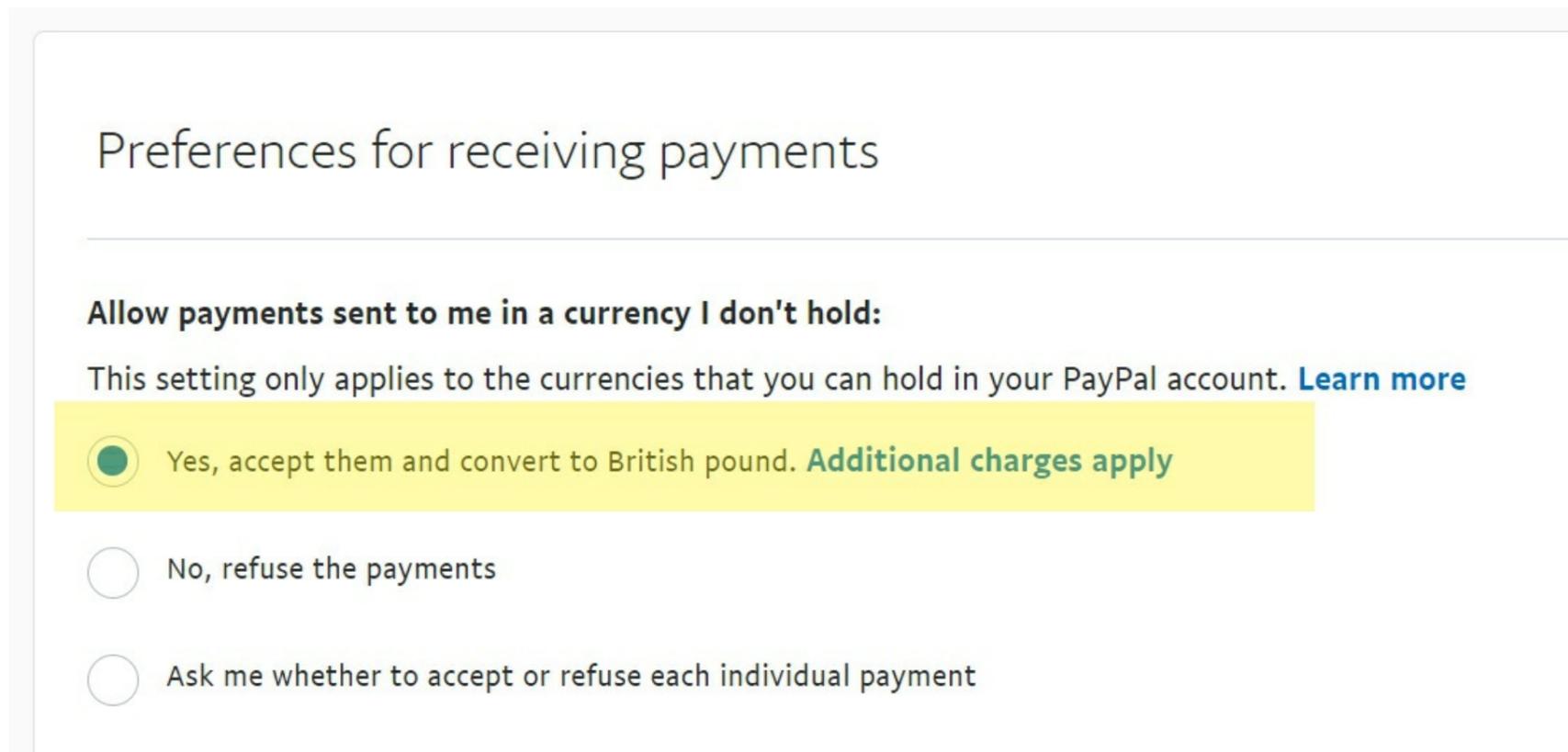
have had the chance to review their order in the checkout fragment. For this reason, we will process their payment immediately upon authorisation. Setting the userAction property of the PayPalCheckoutRequest object to USER\_ACTION\_COMMIT changes the wording of the PayPal authorisation window so that the user is aware that their payment will be processed immediately.

When testing your app, you may find that PayPal rejects transactions that use a foreign currency. To fix this, you need to log in to your Sandbox or Production PayPal account and access the payment preferences using one of the following links:

**Sandbox** - <https://www.sandbox.paypal.com/businessmanage/preferences/payments>

**Production** - <https://www.paypal.com/businessmanage/preferences/payments>

Next, locate the 'Allow payments sent to me in a currency I don't hold:' preference and set it to 'Yes, accept them and convert to British pound.'. Note British Pound may be a different currency based on the country your PayPal account is associated with.



Your PayPal account is now configured to accept payments in any currency.

Once the PayPalCheckoutRequest object has been constructed, the PayPalClient object's tokenizePayPalAccount method will prompt the customer to authorise the payment. In the initiatePayment method, you may notice a TODO comment that mentions saving the order total in the shared preferences file. This measure is there because after authorising the payment with PayPal in a popup browser window, the user will return to the app. Many components such as the MainActivity class and view model will restart and data including the total order price will be lost. Losing this data is not ideal because we need to refer to the total order price when finalising the transaction with Braintree. For this reason, replace the TODO comment with the following code to run a method called saveOrderTotal that will save the total order price to the app's shared preferences file. Data in the shared preferences file persists even when the app is shut down or restarted.

```
saveOrderTotal(orderTotal)
```

Next, define the saveOrderTotal method by adding the following code below the initiatePayment method:

```
private fun saveOrderTotal(total: String?) = sharedPreferences.edit().apply {  
    putString("orderTotal", total)  
    apply()  
}
```

The saveOrderTotal method will write the total order price to the shared preferences file under a key named orderTotal. Once the transaction is finalised, the value associated with the orderTotal key will be set to null. This is why the method has an argument called total, which features a question mark next to its type declaration (String?). The question mark signals that the argument is nullable. It can equal a string or a null value.

The outcome of the PayPal authentication request will be processed when the customer is redirected back to the app by PayPal. When the user returns to the app, the onResume stage of the activity lifecycle will run. To evaluate the outcome of the PayPal authentication process in the onResume stage, add the following code below the onCreate

method in the MainActivity class:

```
override fun onResume() {
    super.onResume()
    val browserSwitchResult = braintreeClient.deliverBrowserSwitchResult(this)
    if (browserSwitchResult != null) {
        paypalClient.onBrowserSwitchResult(browserSwitchResult) { paypalAccountNonce, error ->
            if (error != null) {
                Toast.makeText(this, resources.getString(R.string.payment_error), Toast.LENGTH_SHORT).show()
            } else postNonceToServer(payPalAccountNonce?.string)
        }
    }
}
```

Once the user has authorised the transaction through PayPal, Braintree will create a PayPalAccountNonce object that contains information about the payment. Our app can use the PayPalAccountNonce object to finalise the payment. In the above code, we use the PayPalClient object's onBrowserSwitchResult method to retrieve the outcome of the PayPal authentication request. If an exception has been thrown, then a toast notification will advise the customer that there was an error processing their payment. You can change the logic here to handle errors differently if you wish. Meanwhile, if no error has occurred, then the PayPalAccountNonce object is sent to a method called postNonceToServer. The postNonceToServer method will evaluate the PayPalAccountNonce object, process the payment and finalise the customer's order.

Once the user has authorised the payment and a PayPalAccountNonce object has been created, you can retrieve additional information from the PayPalAccountNonce. For example, you could retrieve the customer's contact details and billing address from PayPal by adding the following commands to the onPaymentMethodNonceCreated method:

```
// Access additional information
val email = paymentMethodNonce.email
val firstName = paymentMethodNonce.firstName
val lastName = paymentMethodNonce.lastName
val phone = paymentMethodNonce.phone
val billingAddress = paymentMethodNonce.billingAddress
val shippingAddress = paymentMethodNonce.shippingAddress
```

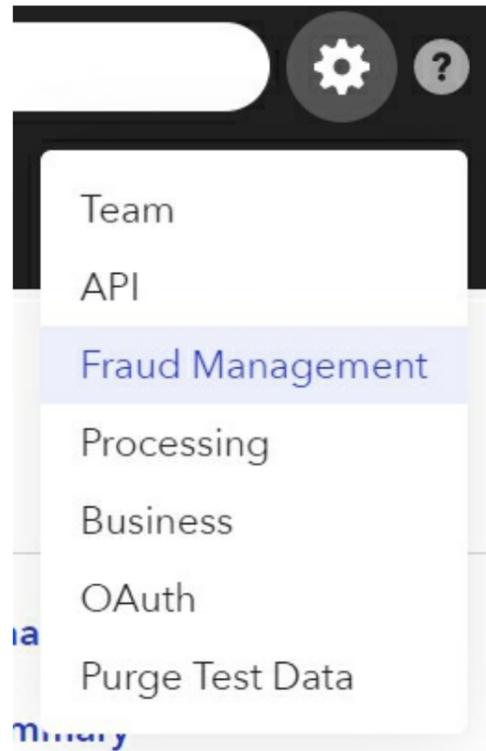
For a full list of details you can extract from the PayPalAccountNonce class, refer to the official Braintree documentation <https://www.javadoc.io/static/com.braintreepayments.api/braintree/2.13.0/com/braintreepayments/api/models/PayPalAccountNonce.html>

In this section, we have covered how to process payments using PayPal. Braintree also supports other payment methods though such as Venmo, Google Pay, Android Pay and more. You can incorporate these payment methods into your app and provide a wider range of payment options to your customers. To find the integration instructions for the additional payment methods then visit Braintree's website:

<https://developers.braintreepayments.com/guides/payment-method-types-overview>.

## Processing transactions

Once the user has authorised a payment and a PayPalAccountNonce object has been created, you must send the PayPalAccountNonce to your website to finalise the transaction. As well as sending the PayPalAccountNonce, it would also be beneficial to include device data for analysis by Braintree's Premium Fraud Management Tools (see <https://developer.paypal.com/braintree/docs/guides/premium-fraud-management-tools/configuration>). Braintree's Premium Fraud Management Tools attempt to detect and prevent fraud by following a set of rules involving geolocation and device data. Enabling the fraud management tools is relatively simple. First, log into your Braintree account, click the cog icon and navigate to the Fraud Management section.



Next, enable either the regular or premium fraud prevention tools depending on which tools best suit your business needs. Note that using the premium fraud prevention tools may incur additional fees.

 **Fraud Protection** **Enable**

*Included with standard pricing.*

Leverage intelligence from the PayPal and Braintree network plus state-of-the-art machine learning technology to detect fraudulent transactions and manage risk.

- Intuitive dashboard and at-a-glance metrics
- Real-time rule testing to assess impact
- Ongoing recommendations for optimization
- **Transaction Risk Score** turned on by default

**OR**

 **Fraud Protection Advanced** **Enable**

*Additional fees apply.*

In addition to the features offered by Fraud Protection, you'll also get features to help you investigate high-risk transactions and develop fraud prevention strategies based on your unique business needs. Find out if you're eligible at [Premium Fraud Tools](#).

- Review and investigate transactions
- Create new custom filters
- Manage block and allow lists
- See how transactions are linked through shared attributes

Once you have enabled either the regular or premium fraud prevention tools, return to Android Studio and add the following variable to the top of the MainActivity class:

```
private var deviceData = ""
```

The deviceData variable will store the data that Braintree will use to assess whether the transaction could be fraudulent. The data will be gathered using Braintree's DataCollector class. To define a method called collectDeviceData that gathers the data when required, add the following code below the saveOrderTotal method:

```
private fun collectDeviceData() {
    DataCollector(braintreeClient).collectDeviceData(this) { data, _ ->
        deviceData = data ?: ""
    }
}
```

Note you may need to add the following import statement to the top of the file:

```
import com.braintreepayments.api.DataCollector
```

The collectDeviceData method will be run whenever the app requests a payment. Moving on, let's discuss the PHP code that will allow your website to process PaymentMethodNonce objects and the fraud detection device data. In the example code, you should find a file called **process\_transaction.php** that reads as follows:

```
<?php
// import the PHP library
require __DIR__ . '/lib/autoload.php';
use Braintree\Gateway;

$gateway = new Gateway([
    'environment' => 'sandbox',
    'merchantId' => 'MERCHANT-ID',
    'publicKey' => 'PUBLIC-KEY',
    'privateKey' => 'PRIVATE-KEY'
]);
$amount = $_POST["amount"];
$currency = $_POST["currency_iso_code"];
$nonceFromTheClient = $_POST["payment_method_nonce"];
$deviceDataFromTheClient = $_POST["client_device_data"];

// Define a separate case in the switch for each additional currency your store supports
switch ($currency) {
    case "USD":
        $merchantAccount = "currency_usd";
        break;
    case "EUR":
        $merchantAccount = "currency_eur";
        break;
    default:
        $merchantAccount = "DEFAULT-MERCHANT-ID";
}
$result = $gateway->transaction()->sale([
    'amount' => $amount,
    'paymentMethodNonce' => $nonceFromTheClient,
    'deviceData' => $deviceDataFromTheClient,
    'merchantAccountId' => $merchantAccount,
    'options' => [
        'submitForSettlement' => True
    ]
]);
if ($result->success) {
    // See $result->transaction for details
    echo ($outcome = "SUCCESSFUL");
} else {
    // see $result->message for the error message
    echo ($outcome = "UNSUCCESSFUL");
}
?>
```

The above PHP code begins similar to the **client\_token.php** file we created earlier by importing the Braintree PHP SDK and creating an instance of the Gateway class. To construct the Gateway class, you will need to fill in the same fields as in the **client\_token.php** file and provide your sandbox Braintree account's merchantId, publicKey and privateKey (see the 'Generating a client token' section for instructions on how to find these details). When the app contacts the **process\_transaction.php** file, it will supply the total order amount, the PaymentMethodNonce

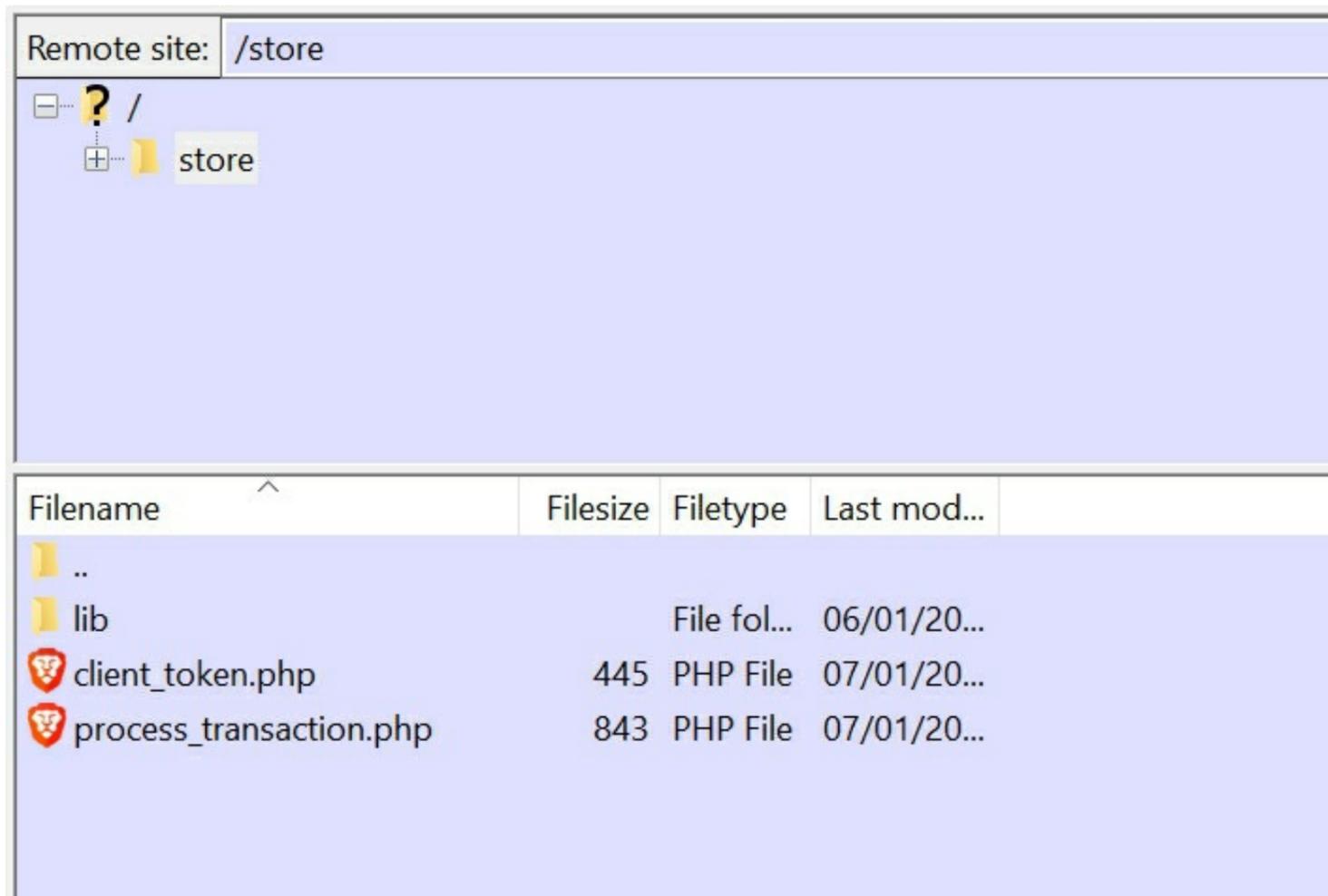
object associated with the customer’s payment method, the device data that was collected for fraud detection purposes, and the ISO code for the user’s selected currency. The **process\_transaction.php** file retrieves these bits of information using POST-based HTTP commands and uses the data to prepare a Transaction response object.

Each transaction currency that your store supports is associated with a different Braintree merchant account, as discussed in the section ‘Configuring your Braintree account to accept payment in multiple currencies’. To determine the appropriate merchant account, the above code uses a PHP switch to match the ISO code associated with the user’s selected currency with the appropriate merchant account ID. A PHP switch expression is equivalent to a when block in Kotlin. You must define a case in the switch for each currency that your store supports, as illustrated for the USD and EUR currencies in the above code. Simply add the ISO code for the currency, then set the merchantAccount variable to equal the merchant account ID you created for that currency in your Braintree account. The switch also contains a default block, which defines the merchant account that will be used if none of the previous cases applies. In this instance, the default block should use the default merchant account ID for your Braintree account.

The PHP code proceeds to create a Transaction response object that Braintree will use to charge the amount specified in the amount variable to the payment method supplied in the PaymentMethodNonce object. Braintree will also analyse the supplied device data for potential indications of credit or debit card fraud. If the transaction appears to be legitimate, then the payment is processed immediately because the submitForSettlement option is set to true. The above code details all the necessary steps that are required to process the payment; however, you can also supply additional information such as the customer’s billing address and personal information if you wish. For a full list of details you can supply when creating the Transaction response object then refer to Braintree’s official documentation: <https://developers.braintreepayments.com/reference/request/transaction/sale/php>

The outcome of the transaction is stored in a variable called result. If the transaction was successful, then the PHP file will emit a variable called outcome which is set to “SUCCESSFUL”. Otherwise, the emitted outcome variable will contain the text “UNSUCCESSFUL”. The Android app can then use this text response to determine the appropriate course of action. The if expression that emits the outcome variable is also where you will perform any post-transaction processing that is required for your business such as sending order confirmation emails or logging the order in a database.

Upload the completed **process\_transaction.php** file to the **store** directory in your website using your FTP client such as FileZilla.



Let’s now return to Android Studio and define a method called postNonceToServer that will communicate with the **process\_transaction.php** file and send the required information (order amount, payment method nonce and device data). Add the following code below the collectDeviceData method in the MainActivity class:

```
private fun postNonceToServer(nonce: String?) {
```

```

if (nonce == null) {
    Toast.makeText(this, getString(R.string.payment_error), Toast.LENGTH_LONG).show()
    return
}

collectDeviceData()

val client = AsyncHttpClient()
val params = RequestParams().apply {
    put("amount", sharedPreferences.getString("orderTotal", null) ?: return)
    put("currency_iso_code", storeViewModel.currency.value?.code ?: defCurrency)
    put("payment_method_nonce", nonce)
    put("client_device_data", deviceData)
}

saveOrderTotal(null)

// TODO: Replace YOUR-DOMAIN.com with your website domain
client.post("https://YOUR-DOMAIN.com/store/process_transaction.php", params,
    object : TextHttpResponseHandler() {
        override fun onSuccess(statusCode: Int, headers: Array<out Header>?, outcome: String?) {
            if (outcome == "SUCCESSFUL") {
                Toast.makeText(this@MainActivity, resources.getString(R.string.payment_successful),
                    Toast.LENGTH_LONG).show()
                clearCart()
            } else Toast.makeText(this@MainActivity, resources.getString(R.string.payment_error),
                Toast.LENGTH_LONG).show()
        }

        override fun onFailure(statusCode: Int, headers: Array<out Header>?, outcome: String?, throwable:
            Throwable?) { }
    }
)
}

```

The `postNonceToServer` method features an argument that accepts a payment method nonce string. If the nonce string is null, then it is likely there was an error authenticating the customer's payment method. In which case, a toast notification will inform the user that there was an error and no further processing will occur. Meanwhile, if a nonce is supplied, the method uses the `AsyncHttpClient` class to communicate with your website via an HTTP request. To facilitate the HTTP request, an instance of the `RequestParams` class is prepared using supplementary data that should be included with the request. In this instance, we supply the order amount, the ISO code associated with the customer's chosen currency, the payment method nonce and fraud prevention device data. The **process\_transaction.php** file can retrieve the data from the HTTP request payload based on each item's key. For example, the payment method nonce is stored under the key `payment_method_nonce` and can be retrieved using the following PHP code:

```
$_POST["payment_method_nonce"];
```

Once all of the request parameters have been defined, the `saveOrderTotal` method is run with a null value passed as the total parameter. The `saveOrderTotal` method will clear the total order price from the shared preferences file because the order is complete. Next, the `AsyncHttpClient` object submits the request to the **process\_transaction.php** web page. Remember to modify the `YOUR-DOMAIN.com` part of the get request URL to reflect the address of your web domain. The `TextHttpResponseHandler` object's `onSuccess` callback method is used to process the response from the **process\_transaction.php** web page. Remember, the **process\_transaction.php** file will echo a value of "SUCCESSFUL" if the payment was processed successfully, and a value of "UNSUCCESSFUL" if the transaction could not be completed. If the transaction was completed successfully, then a toast notification will inform the user their order is complete and a method called `clearCart` will empty the shopping basket. Meanwhile, if the transaction was unsuccessful, then a toast notification will inform the user that there was an error processing their payment.

## Clearing the shopping basket following successful payment

Once the user has completed their order, their shopping basket can be emptied. To handle this, add the following

code to the MainActivity class below the postNonceToServer method:

```
private fun clearCart() {  
    val products = storeViewModel.products.value?.listOf()  
    for (p in products) p.inCart = false  
    storeViewModel.products.value = products  
    storeViewModel.orderTotal.value = 0.00  
    storeViewModel.clearCart.value = true  
}
```

The clearCart method defined above retrieves the list of Product objects from the StoreViewModel view model and sets the inCart field of each object to false. It then sends the updated list of Product objects back to the view model, sets the value of the orderTotal variable to 0.00, and sets a view model variable called clearCart to true. The ProductsFragment class will monitor the clearCart variable. Whenever the clearCart variable is set to true, the ProductsFragment class will reset the “Add to basket” buttons for every product that had previously been in the shopping basket.

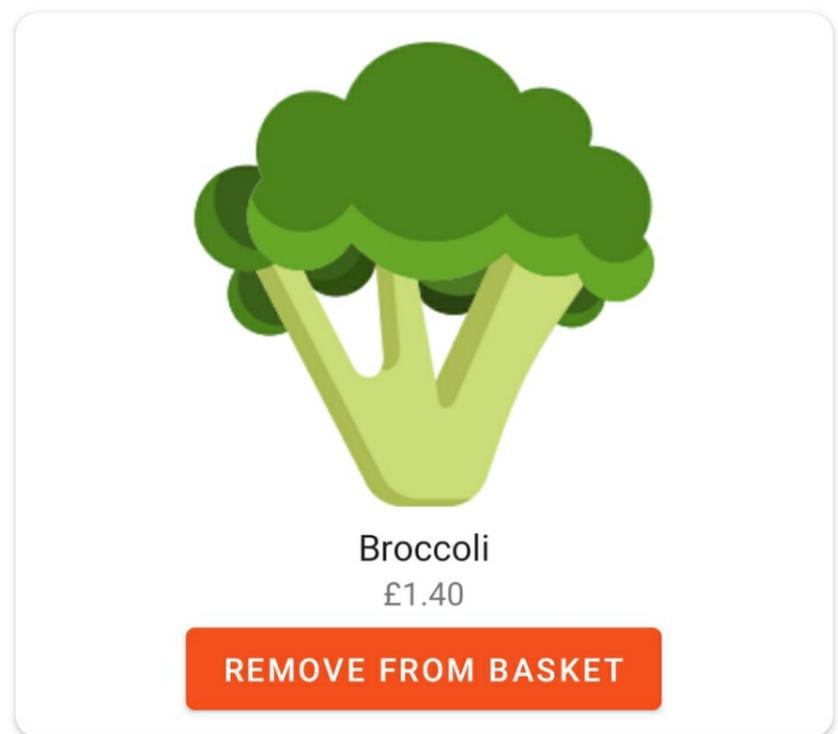
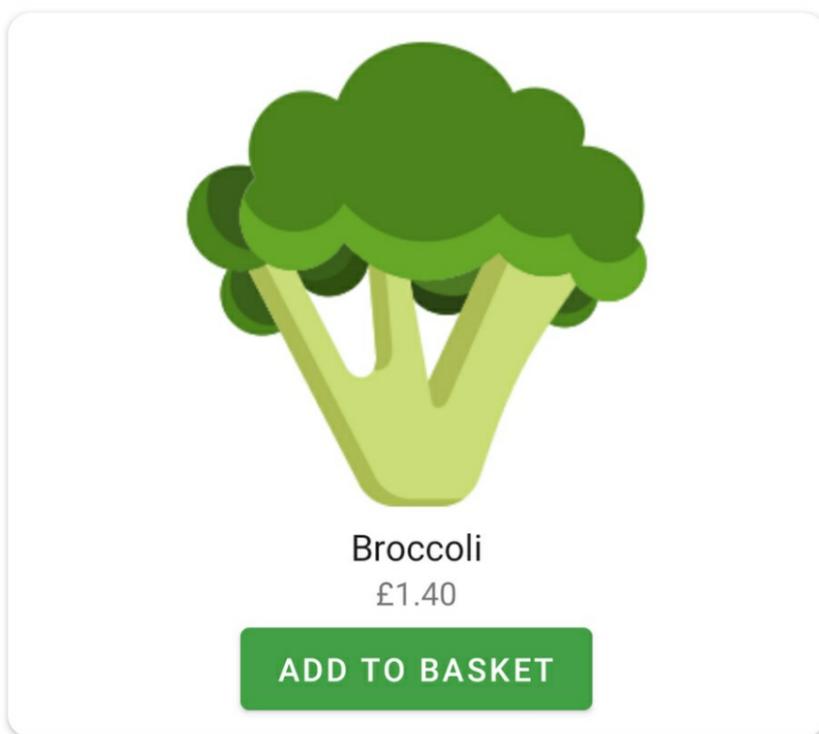
To enable this functionality, open the StoreViewModel class and add the following variable to the top of the class:

```
var clearCart = MutableLiveData(false)
```

Next, open the **ProductsFragment.kt** file (**Project > app > java > name of the project > ui > products**) and add the following code to the onCreateView method:

```
storeViewModel.clearCart.observe(viewLifecycleOwner, { clearCart ->  
    clearCart?.let {  
        if (clearCart) {  
            val products = productsAdapter.products  
            for ((i, p) in products.withIndex()) {  
                if (p.inCart) {  
                    p.inCart = false  
                    productsAdapter.products[i] = p  
                    productsAdapter.notifyItemChanged(i)  
                }  
            }  
            storeViewModel.clearCart.value = false  
        }  
    }  
})
```

The above code registers an observer on the StoreViewModel class’s clearCart variable. Whenever the value of the clearCart variable is set to true, the observer will retrieve the list of Product objects from ProductsAdapter class and iterate through each item. If the inCart value for a product is true, then the above code will revert it to false, replace the Product object in the adapter and call the adapter’s notifyItemChanged method to refresh the product’s position in the RecyclerView. This will change the addToBasket button from red with the text “Remove from basket”, to green with the text “Add to basket” (see the onBindViewHolder method in the ProductsAdapter class for the code that makes this possible). Once the list of products has been processed, the StoreViewModel class’s clearCart variable is set back to false.



## The BottomNavigationView widget

The app is almost finished; the user can add products to their shopping basket and pay for their order via PayPal. The last thing we need to do is configure the BottomNavigationView widget. The BottomNavigationView was created automatically by Android Studio as part of the Bottom Navigation Activity project template. It will allow the user to navigate between the key top-level destinations in the app, which in this case is the products fragment and checkout fragment.

The different destinations are defined in a navigation graph resource file called **mobile\_navigation.xml**, which can be found by navigating through **Project > app > res > navigation**. Open the file in Code view and edit the file so it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/mobile_navigation"
    app:startDestination="@+id/navigation_products">

    <fragment
        android:id="@+id/navigation_products"
        android:name="com.example.store.ui.products.ProductsFragment"
        android:label="@string/title_products"
        tools:layout="@layout/fragment_products" />

    <fragment
        android:id="@+id/navigation_checkout"
        android:name="com.example.store.ui.checkout.CheckoutFragment"
        android:label="@string/title_checkout"
        tools:layout="@layout/fragment_checkout" />

</navigation>
```

The above code defines separate navigation destinations for the products and checkout fragments. For each destination, the ID attribute identifies the destination in the navigation graph. Additional attributes include a name attribute, which defines the destination's location in the app, a label attribute which defines the text that is displayed in the app toolbar when that destination is active, and a layout attribute which defines the layout file that is associated with the destination.

The navigation destinations that are displayed in the BottomNavigationView widget are defined in a menu resource file that should automatically have been generated by Android Studio. To locate the file, navigate through **Project > app > res > menu** and open the file called **bottom\_nav\_menu.xml**. Switch the file to Code view and edit the file so it contains the following two menu items:

```
<item
```

```

android:id="@+id/navigation_products"
android:icon="@drawable/ic_store"
android:title="@string/title_products" />

```

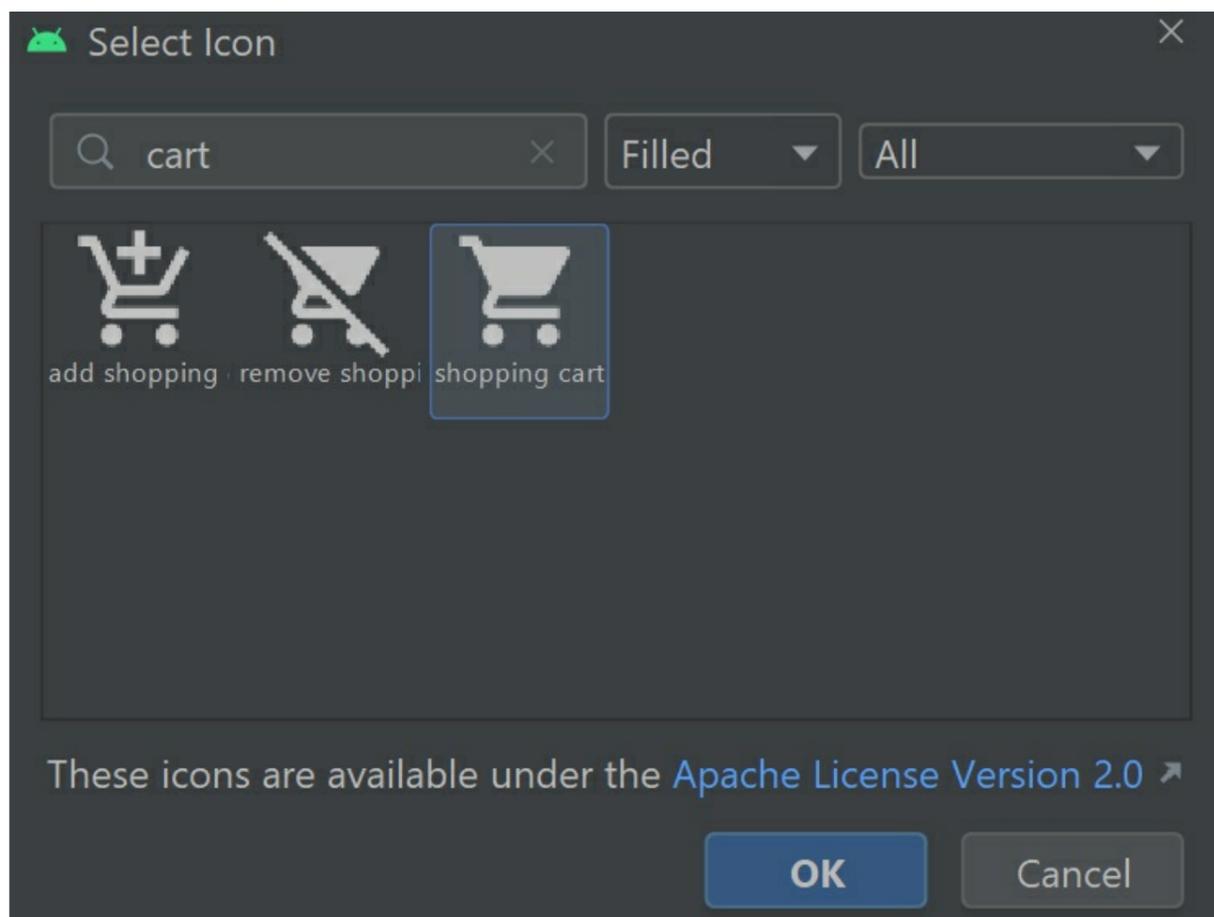
```

<item
  android:id="@+id/navigation_checkout"
  android:icon="@drawable/ic_cart"
  android:title="@string/title_checkout" />

```

The menu items define different options that will appear in the BottomNavigationView widget. The ID of each menu item should match the ID of the corresponding navigation destination in the **mobile\_navigation.xml** navigation graph. Each menu item also contains an icon and a title, which will be displayed in the BottomNavigationView.

The icons referenced for the navigation\_products and navigation\_checkout menu items have not yet been created. To address this, right-click the **drawable** directory (**Project > app > res**) then select **New > Vector Asset**. For the clip art, search for and select the icon called 'shopping cart' then press OK.



Set the name of the vector asset to 'ic\_cart' then press Next and Finish to save the icon. Repeat the above steps store drawable, except this time select the icon called 'store' and name the drawable 'ic\_store'.

The BottomNavigationView widget can be found in the **activity\_main.xml** layout (**Project > app > res**). The activity\_main layout is the main layout of the activity and will load when the app is launched. It contains the BottomNavigationView widget and a fragment, which will display the content of the user's current destination in the app. It is now convention to use a FragmentContainerView widget rather than a regular fragment to display content, so replace the fragment with the following code:

```

<androidx.fragment.app.FragmentContainerView
  android:id="@+id/nav_host_fragment"
  android:name="androidx.navigation.fragment.NavHostFragment"
  android:layout_width="match_parent"
  android:layout_height="0dp"
  app:defaultNavHost="true"
  app:navGraph="@navigation/mobile_navigation"
  app:layout_constraintTop_toTopOf="parent"
  app:layout_constraintBottom_toTopOf="@id/nav_view" />

```

The above code defines a FragmentContainerView widget that will source its content from the **mobile\_navigation.xml** navigation graph. The height of the FragmentContainerView is set to 0dp, which means its height will occupy the maximum available space according to its constraints. In this case, the FragmentContainerView is constrained to the top of the parent layout and the BottomNavigationView at the bottom

of the layout. These constraints mean the `FragmentManagerView` will occupy the maximum available space once it has left enough room for the `BottomNavigationView`.

Note the root `ConstraintLayout` in the `activity_main.xml` layout may contain a `paddingTop` attribute `android:paddingTop="?attr/actionBarSize"` designed to leave space for an action bar. This attribute will not be necessary for this app and can be deleted.

To make the `BottomNavigationView` widget operational, return to the `MainActivity.kt` file and replace the `NavController` and `AppBarConfiguration` variables in the `onCreate` method with the following code:

```
val navHostFragment = supportFragmentManager.findFragmentById(R.id.nav_host_fragment) as NavHostFragment
val NavController = navHostFragmentNavController
val appBarConfiguration = AppBarConfiguration(setOf(R.id.navigation_products, R.id.navigation_checkout))
```

The `onCreate` method now initialises a `NavHostFragment` object, which will provide access to the `FragmentManagerView` widget from the `activity_main.xml` layout. The `FragmentManagerView` will allow the user to navigate between the destinations defined in the `mobile_navigation.xml` navigation graph. We also defined a variable called `AppBarConfiguration`, which details the app's top-level destinations. A top-level destination is the origin of a navigation pathway. In the `Stope` app, the `products` fragment and the `checkout` fragment are top-level destinations.

## Going live

In this final section, we will discuss the changes required to switch from sandbox to production mode and begin processing genuine payments. Naturally, you should only follow these steps once you have confirmed your store works correctly in sandbox mode. First, you will require a production Braintree account, which you can apply for by completing this online form: <https://signups.braintreepayments.com/>. It is important to remember that your sandbox and production Braintree accounts are not linked. For this reason, you will have to replace all of the information in your app and web server PHP files that is specific to the sandbox account. A full list of items you need to change is included below:

- **Tokenization key**

Replace the value of the `TOKENIZATION_KEY` variable in the `MainActivity` class's companion object with the tokenization key for your production Braintree account.

- **Merchant ID, public key and private key**

In both the `client_token.php` and `process_transaction.php` files you need to update the `Gateway` object with your production Braintree account details and switch the environment to 'production':

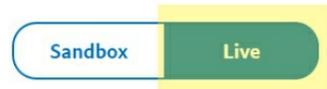
```
$gateway = new Gateway([
    'environment' => 'production',
    'merchantId' => 'PRODUCTION-MERCHANT-ID',
    'publicKey' => 'PRODUCTION-PUBLIC-KEY',
    'privateKey' => 'PRODUCTION-PRIVATE-KEY'
]);
```

- **Merchant account IDs**

In the `Business` section of your production Braintree account, you will need to create separate merchant accounts for each currency your store will support. For simplicity, it will be best to use the same merchant account IDs for each currency as in your sandbox Braintree account. Otherwise, you will need to update the `process_transaction.php` file with the new production merchant account IDs.

To process PayPal payments, you will require a live PayPal app for your store. To create a live app, log in to your PayPal developer account (<https://developer.paypal.com/developer/applications>) and navigate to the `My Apps & Credentials` page. Ensure you are viewing the page in `Live` mode then click `Create App`.

# My Apps & Credentials



## REST API apps

Create an app to receive REST API credentials for testing and live transactions.

App Name	Type	Actions
Create your first app to view it here.		



**Note:** Features available for live transactions are listed in your [account eligibility](#).

Give your app a name (e.g. store) then click Create App. Once the live app has been created, note down the client ID and secret and keep these details safe. Next, you need to link the live PayPal app with your production Braintree account. To do this, navigate to the Processing section of your production Braintree account and ensure the PayPal item in the Payment Methods section is switched on. You can then fill in the details of your live PayPal app and business PayPal account.

Once all the above steps have been implemented, your mobile store should be ready to begin processing genuine transactions. It might be worthwhile testing your store with a couple of low value orders to confirm all the processes work correctly following the transition from sandbox to production mode.

## Summary

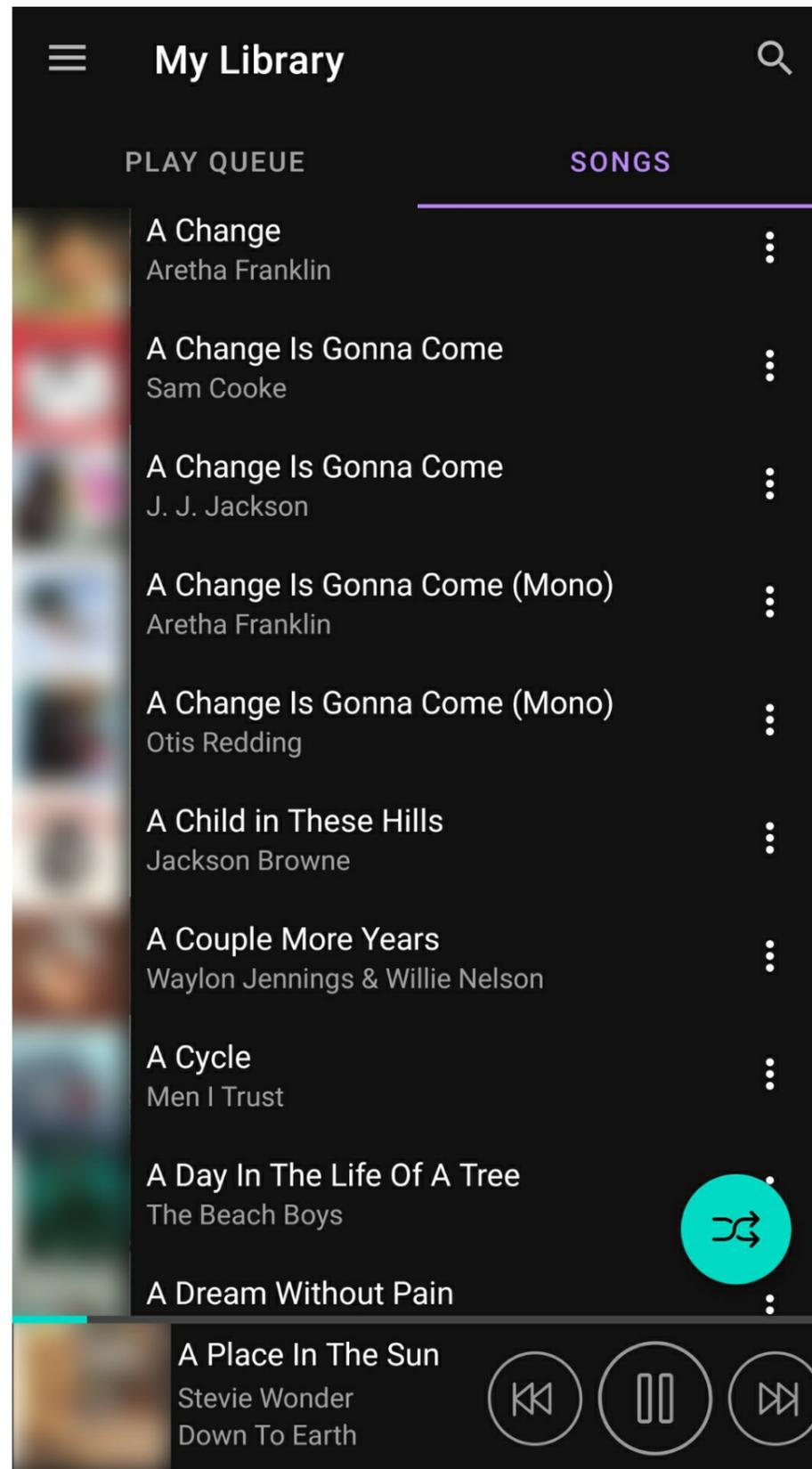
Congratulations on completing the Store app! In creating this app, you have covered the following skills and topics:

- Create an application using the Bottom Navigation Activity project template.
- Display products that the user can add to, or remove from, their shopping basket.
- Incorporate a currency exchange API that allows the user to shop in multiple different currencies.
- Integrate the Braintree Payments API with your project.
- Utilise Braintree's fraud prevention tools to protect your store against fraudulent transactions.
- Process payments via PayPal.
- Customise the PayPal payment flow.
- Use your website to generate Braintree client tokens and authenticate your app.
- Use the AsyncHttpClient class to manage HTTP requests and send data to web pages.

# How to create a Music application

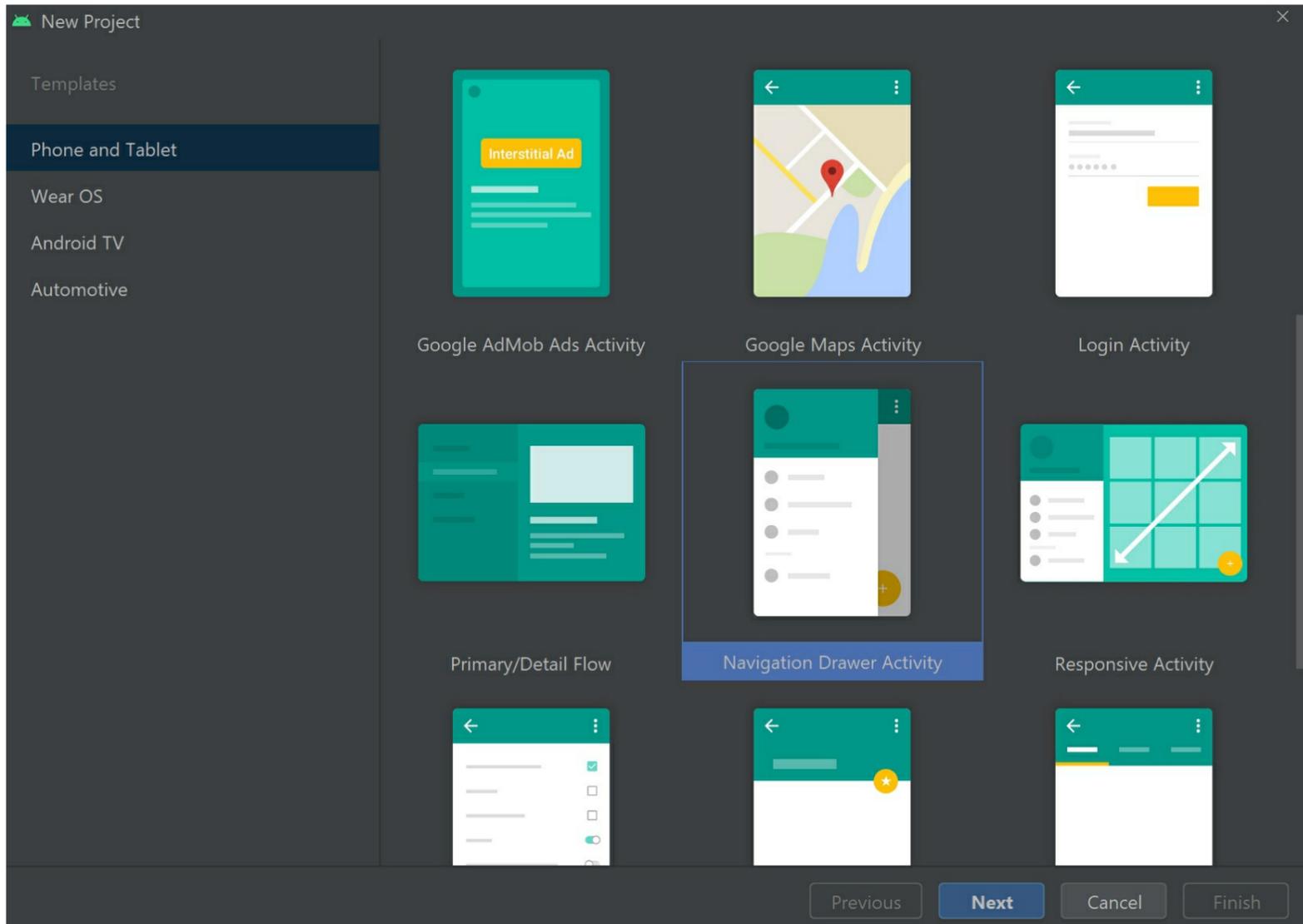
## Introduction

For this project, we will create a music app that allows the user to play songs on their device, build play queues and manage their music library. In creating this application, you will learn how to incorporate a Room SQLite database to organise the music library, use a media browser service to coordinate playback, create notifications and more. The techniques you will learn in this section were used to make an app called Supernova, which is available on the Google Play app store if you want to see this project in action: <https://play.google.com/store/apps/details?id=com.codersguidebook.supernova>.

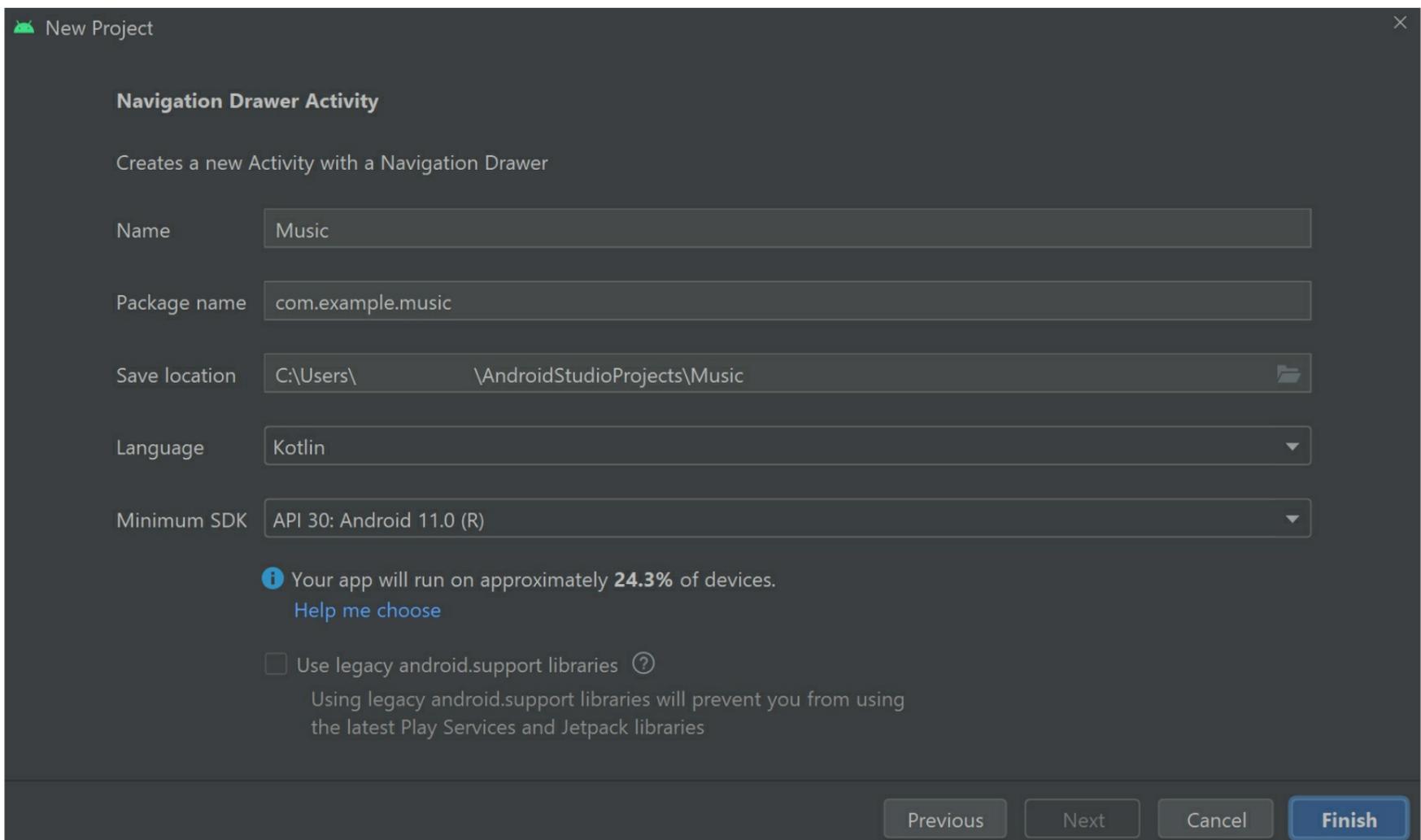


## Getting started

To begin, open Android Studio and create a new project. Select Navigation Drawer Activity as the project template. The Navigation Drawer Activity template provides your app with an expandable navigation panel, which will slide out from the left-hand side and allow the user to navigate to the different destinations in the app.



In the Create New Project window, set the name of the project to Music, choose the Kotlin language and select API level 30.



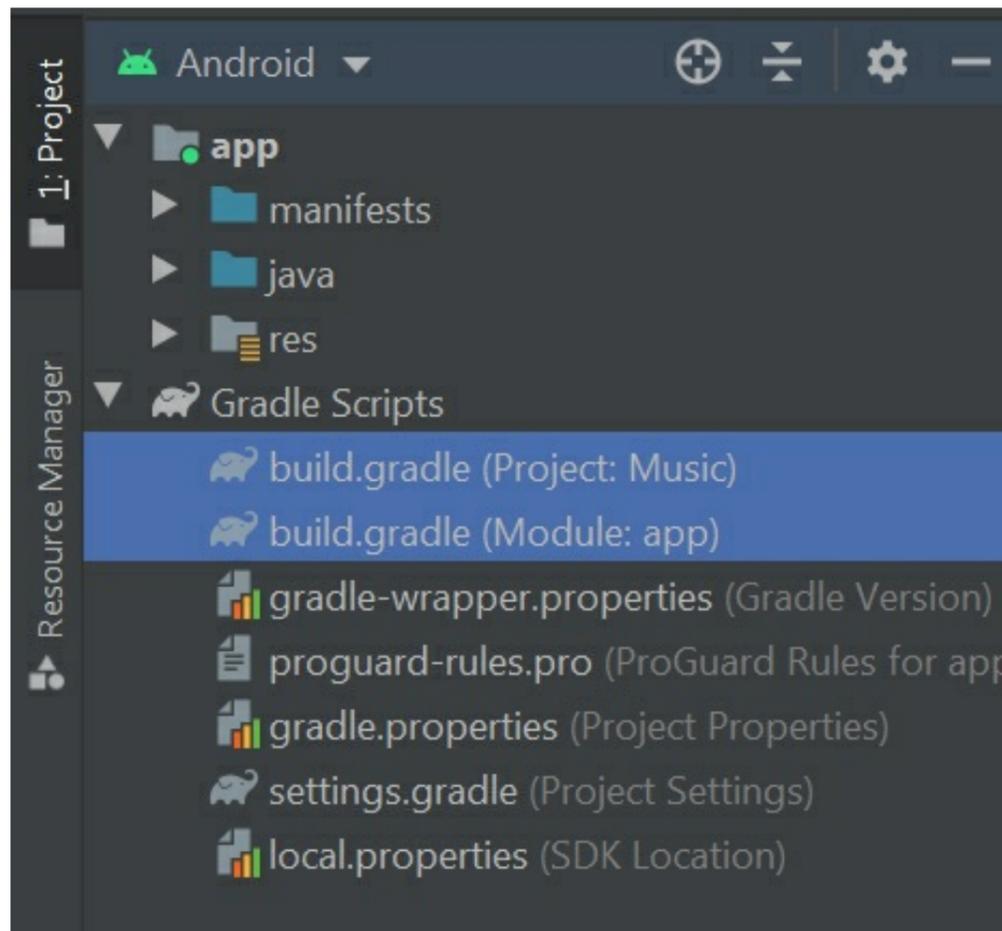
It is recommended you enable Auto Imports to direct Android Studio to add any necessary import statements to your Kotlin files as you code. These import statements are essential for incorporating the external classes and tools required for the app to run. To enable Auto Imports, open Android Studio's Settings window by clicking **File > Settings**. In the Settings window, navigate through **Editor > General > Auto Import** then select 'Add unambiguous imports on the fly' and 'Optimise imports on the fly' for both Java and Kotlin then press

Apply and OK.

Android Studio should now add most of the necessary import statements to your Kotlin class files automatically. Sometimes there are multiple classes with the same name and the Auto Import feature will not work. In these instances, the requisite import statement(s) will be specified explicitly in the example code. You can also refer to the finished project code which accompanies this book to find the complete files including all import statements.

## Configuring the Gradle scripts

For the Music app to perform all the operations we want it to, we must manually import several external packages using a toolkit called Gradle. To do this, navigate through **Project > Gradle Scripts** and open the Project and Module level **build.gradle** files:



In the Project-level **build.gradle** file, add the following classpath to the dependencies element:

```
classpath "androidx.navigation:navigation-safe-args-gradle-plugin:2.4.1"
```

The above classpath is required to use a feature called safe args, which is a method for transferring data between destinations in the app.

Next, switch to the Module-level **build.gradle** file and add the following lines to the plugins element at the top of the file:

```
id 'kotlin-kapt'  
id 'kotlin-parcelize'  
id 'androidx.navigation:safeargs.kotlin'
```

Finally, refer to the dependencies element and add the following code to the list of implementation statements:

```
def roomVersion = '2.4.2'  
  
implementation "androidx.media:media:1.5.0"  
implementation 'androidx.preference:preference-ktx:1.2.0'  
implementation 'androidx.recyclerview:recyclerview:1.3.0-alpha01'  
implementation 'androidx.cardview:cardview:1.0.0'  
implementation 'androidx.viewpager2:viewpager2:1.0.0'  
  
implementation "androidx.room:room-runtime:$roomVersion"  
annotationProcessor "androidx.room:room-compiler:$roomVersion"  
implementation "androidx.room:room-ktx:$roomVersion"  
kapt "androidx.room:room-compiler:$roomVersion"
```

```
implementation "androidx.lifecycle:lifecycle-extensions:2.2.0"
implementation "androidx.lifecycle:lifecycle-common-java8:2.4.1"
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.1"
```

```
implementation 'com.google.code.gson:gson:2.8.8'
```

```
implementation 'com.github.bumptech.glide:glide:4.11.0'
```

```
// FastScroll enables fast scrolling in recycler views https://github.com/timusus/RecyclerView-FastScroll
implementation 'com.simplicityapps:recyclerview-fastscroll:2.0.1'
```

The above implementation statements enable your app access to several features including a database management system called Room; a JSON string processor called GSON, which will help prepare complex objects for storage; and a package called RecyclerView-FastScroll (see: <https://github.com/timusus/RecyclerView-FastScroll>), which will allow the user to rapidly scroll through RecyclerView widgets which hold lots of data (e.g. the list of songs in the user's music library).

We're now finished with the Gradle Scripts files. Don't forget to re-sync your project when prompted!

Gradle files have changed since last project sync. A project sync may be necessary for the IDE ... [Sync Now](#)

## Configuring the Manifest file

We'll now turn our attention to the application's manifest file, which contains an overview of the app's activities, as well as the user permissions the app requires to run. Open the **AndroidManifest.xml** file by navigating through **Project > app > manifests** then add the following line of code above the application element:

```
<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

The above uses-permission elements signal to the device (and the Google Play store) that this app will require permission from the user to launch a foreground service (necessary to play music and show notifications) and access files on the user's device.

Next, locate the activity element and add the following attribute to the opening activity tag:

```
android:windowSoftInputMode="adjustPan"
```

The above code sets the windowSoftInputMode attribute for the MainActivity activity to adjustPan. Setting the soft input mode to adjustPan instructs the app not to reorganise its content when the keyboard is visible. Instead, the keyboard will slide up above the content if necessary. This is useful in several situations, such as when the user is searching for songs. Without the windowSoftInputMode attribute, the keyboard would push the playback controls up the screen and obscure the search results.

Moving on, add the following code below the activity element:

```
<service
  android:name=".MediaPlayerService"
  android:exported="false">
  <intent-filter>
    <action android:name="android.media.browse.MediaBrowserService" />
    <action android:name="android.intent.action.MEDIA_BUTTON" />
    <action android:name="android.media.AUDIO_BECOMING_NOISY" />
  </intent-filter>
</service>
```

The above code lays the groundwork for a service that will manage audio playback. The service will handle several types of external stimuli, including media buttons (such as those found on Bluetooth earphones) and audio becoming noisy (e.g. when earphones are disconnected during playback). Don't worry if **MediaPlayerService** is highlighted in red. This warning will disappear once we create the service.

## Defining the string resources used in the app

Like the other projects covered in this book, the Music app will store all the strings of text used throughout the application in a resource file. To define the string resources, navigate through **Project > app > res** and open the file

called **strings.xml**. Once the file opens in the editor, modify its contents so it reads as follows:

```
<resources>
  <!-- accessibility strings -->
  <string name="close_currently_playing">Close the currently playing view</string>
  <string name="handle_view_desc">Icon that allows you to reorder songs in the current playback list.</string>
  <string name="options_menu">Options menu</string>
  <string name="permission_required">Storage permission is required to run this application</string>
  <string name="play_or_pause_current_track">Play or pause current track</string>
  <string name="repeat_current_playlist">Repeat current playlist</string>
  <string name="search_results">Search results</string>
  <string name="settings">Settings</string>
  <string name="set_album_artwork">Set album artwork</string>
  <string name="shuffle_play_queue">Shuffle the current play queue</string>
  <string name="shuffle_tracks">Play a shuffled version of the music library</string>

  <!-- application strings -->
  <string name="app_name">Music</string>

  <!-- controls strings -->
  <string name="play_pause">Play or pause</string>
  <string name="play_prev">Play previous</string>
  <string name="skip_ahead">Skip ahead a track</string>
  <string name="skip_back">Skip back a track</string>

  <!-- library strings -->
  <string name="album_artwork">Album artwork</string>
  <string name="artist">Artist</string>
  <string name="check_fields_not_empty">Check none of the fields are empty</string>
  <string name="details_saved">Details saved</string>
  <string name="disc">Disc</string>
  <string name="error">An error has occurred. If the song file has been moved or deleted please wait for the library
to refresh.</string>
  <string name="library">My Library</string>
  <string name="no_results">No results found</string>
  <string name="songs">Songs</string>
  <string name="track">Track</string>
  <string name="year">Year</string>

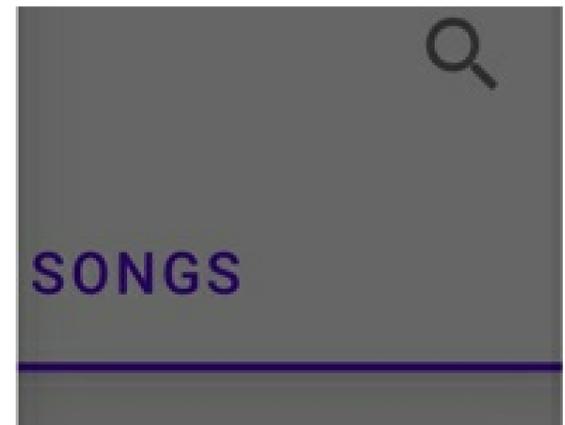
  <!-- menu strings -->
  <string name="added_to_queue">%1$s has been added to the play queue</string>
  <string name="cancel">Cancel</string>
  <string name="done">Done</string>
  <string name="edit_metadata">Edit music info</string>
  <string name="ok">OK</string>
  <string name="play_next">Play next</string>
  <string name="play_queue">Play queue</string>
  <string name="remove_from_queue">Remove from play queue</string>
  <string name="save">Save</string>
  <string name="search">Search</string>
  <string name="search_hint">Search music</string>
  <string name="title">Title</string>
  <string name="transition_image">transition_image</string>
  <string name="transition_title">transition_title</string>
  <string name="transition_subtitle">transition_subtitle</string>
  <string name="transition_subtitle2">transition_subtitle2</string>
  <string name="transition_back">transition_back</string>
  <string name="transition_play">transition_play</string>
  <string name="transition_forward">transition_forward</string>
</resources>
```

The above code separates the strings into categories using comments so you can get an idea of how they will be used in the app. For example, the accessibility strings will provide content descriptions for images to help people

with access needs. The only other noteworthy string is the `app_name` string in the application strings category. This string defines the name of the app, as specified in the **AndroidManifest.xml** file.

## Customising the application's themes

Similar to the strings resource file, the project will also contain a resource file detailing all the custom colours that are used throughout the app. Android Studio should already have generated a colours resource file called **colors.xml**, which you can locate by navigating through **Project** > **app** > **res** > **values**. We will define several custom colours that will be used to tint the icons that appear in the navigation drawer.



To define the custom colours, add the following items to the **colors.xml** file:

```
<color name="nav_queue">#E27D60</color>
<color name="nav_songs">#41B3A3</color>
```

We'll now turn our attention to the theme resource **themes.xml** files, which control the appearance of the app and its components. By default, Android Studio generates two **themes.xml** files: a base theme and a night theme. To locate the **themes.xml** files, navigate through **Project** > **app** > **res** > **values** > **themes**. There will be two theme files, one with the word 'night' in brackets after the filename and one without. The file without the term 'night' is the base theme file and the one that we will edit. All style guidelines defined in the base theme resource file will also apply to the night theme unless the night theme resource file contains guidelines that specify otherwise. In the base theme **themes.xml** file, find the style element that begins `<style name="Theme.Music"` and add the following items inside the element:

```
<item name="tabStyle">@style/Widget.Custom.TabLayout</item>
<item name="android:textViewStyle">@style/Widget.Custom.TextView</item>
```

The above code overrides the guidelines for the `TabLayout` and `TextView` widgets with custom style elements. The first custom guidelines will style the `TabLayout` widget that will allow the user to swipe between the play queue and songs tabs.

PLAY QUEUE

SONGS

To define the custom guidelines, add the following element to the base **themes.xml** file:

```
<style name="Widget.Custom.TabLayout" parent="Widget.MaterialComponents.TabLayout">
  <item name="tabBackground">@android:color/transparent</item>
  <item name="android:elevation">14dp</item>
  <item name="tabMode">fixed</item>
  <item name="tabMaxWidth">0dp</item>
  <item name="tabGravity">fill</item>
  <item name="tabPaddingEnd">0dp</item>
  <item name="tabPaddingStart">0dp</item>
</style>
```

The above style element is called `Widget.Custom.TabLayout`. The style element uses its parent attribute to inherit all the style instructions issued by Material Design for `TabLayout` widgets but overwrites specific attributes as required. For example, the style element contains items that add a 14dp elevation to the widget to create a shadow effect. Also, the `tabMode` property is set to `fixed` and the `tabGravity` property is set to `fill`, which together ensure the tab layout stretches across the full width of the device window and the individual tabs are equally spaced apart from one another.

Next, to define the custom TextView widget guidelines, add the following style element to the base **themes.xml** file:

```
<style name="Widget.Custom.TextView" parent="Widget.MaterialComponents.TextView">
  <item name="android:ellipsize">none</item>
  <item name="android:requiresFadingEdge">horizontal</item>
</style>
```

Similar to the TabLayout style element, the above code imports the default TextView style instructions from Material Design and overrides attributes as required. In this instance, the ellipsize property is set to none, which ensures text that stretches beyond the container of the TextView is not truncated with an ellipsis. Instead, a horizontal fading edge will appear to provide a faded effect for Text which is too large.

Come Get To This (Live At The London Pa  
Marvin Gaye

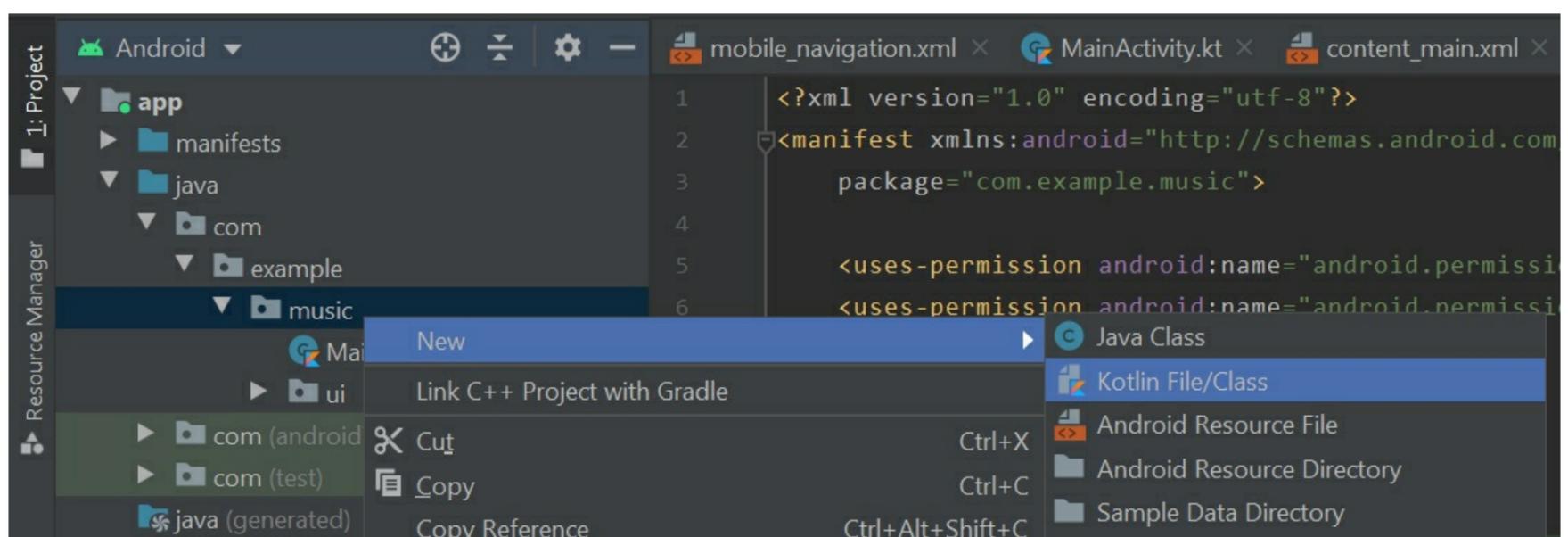
Finally, add the following element to the base **themes.xml** file to define style guidelines that we will apply to certain buttons such as the playback controls:

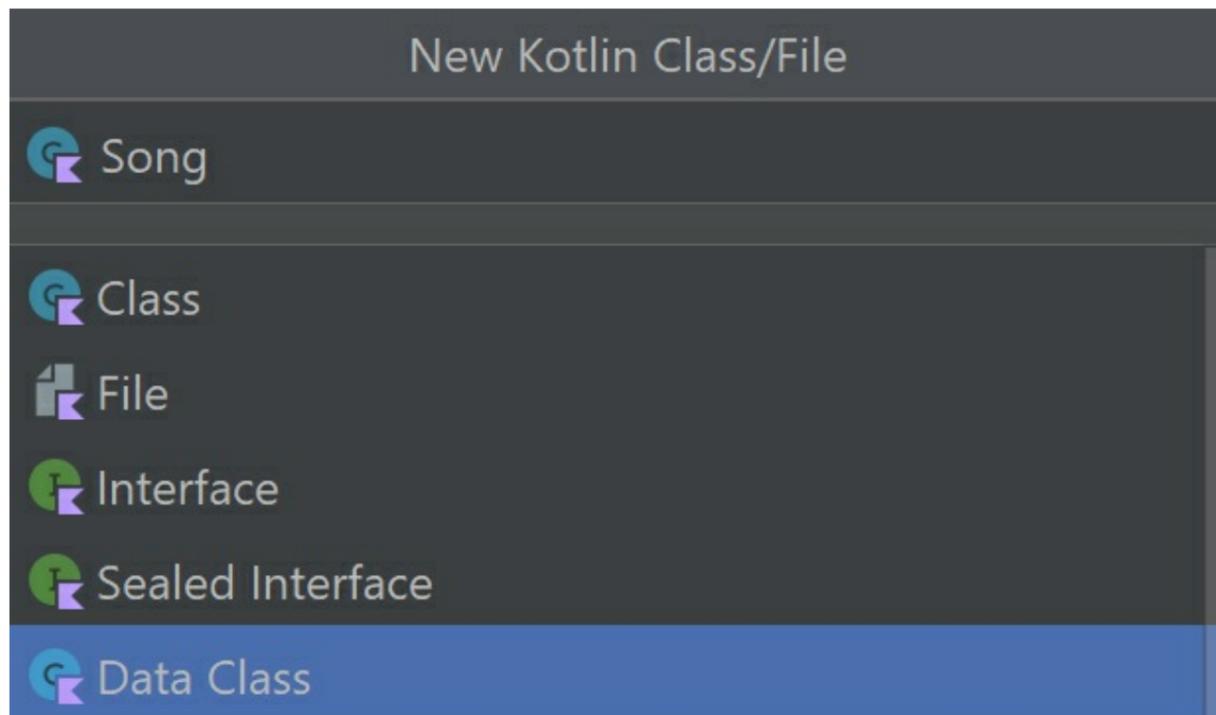
```
<style name="Widget.Custom.Button" parent="Widget.MaterialComponents.Button">
  <item name="android:textColor">?android:attr/textColorPrimary</item>
  <item name="android:tint">@color/material_on_surface_emphasis_medium</item>
  <item name="android:background">?attr/selectableItemBackgroundBorderless</item>
  <item name="android:scaleType">centerCrop</item>
  <item name="android:stateListAnimator">@null</item>
  <item name="android:padding">0dp</item>
</style>
```

The style element defined above changes the button's tint to a medium emphasis colour that contrasts with the surface. The material\_on\_surface\_emphasis\_medium color attribute uses the onSurface colour for the active theme, and so will automatically adapt to the base and night themes. Also, the background attribute is set to selectableItemBackgroundBorderless, which means the ripple effect that appears when the button is pressed will be circular and stretch beyond the normal rectangular/square border of the button.

## Creating the Song data class

To build the user's music library, the app will store information about each audio file in a dedicated data class called Song. A new instance of the Song class will be created for each audio file, and all the Song objects will be stored in an internal database. To create the Song data class, navigate through **Project > app > java** and right-click the folder with the name of the project. Next, select **New > Kotlin File/Class**, name the class Song and select Data Class from the list of options.





A file called **Song.kt** should open in the editor. Edit the class's code so it reads as follows:

```
import kotlinx.parcelize.Parcelize

@Parcelize
@Entity(tableName = "music_table")
data class Song(
    @PrimaryKey val songID: Long,
    @ColumnInfo(name = "song_track") var track: Int,
    @ColumnInfo(name = "song_title") var title: String,
    @ColumnInfo(name = "song_artist") var artist: String,
    @ColumnInfo(name = "song_album") var album: String,
    @ColumnInfo(name = "song_album_id") val albumID: String,
    @ColumnInfo(name = "song_uri") val uri: String,
    @ColumnInfo(name = "song_year") var year: String
) : Parcelable
```

The Song data class features several annotations (preceded by the @ symbol) because instances of the data class will be mapped to a database table using Room. Each annotation defines a characteristic of the data. The first annotation, while not strictly necessary for the Room database, is @Parcelize. The Parcelize annotation signals that objects of the class are Parcelable. Parcelable objects can be more easily sent between different areas of the app (e.g. activities, fragments and services).

Next, the Song data class is labelled with the @Entity annotation and assigned a table name. An entity is essentially a map for a table in the Room database. In this instance, we are defining a table called 'music\_table' and using the different properties of the Song data class as the columns. Columns will be created for the song's id, track number, title, artist, album name, album ID, URI (path to the file location) and year.

The songID property of the Song data class is labelled with the @PrimaryKey annotation. The primary key distinguishes different entries in the database table, and each entry must have a unique primary key value. For our purposes, it makes sense to set the song ID as the primary key because each audio file will have a unique ID in the device's MediaStore.

The final annotation used is @ColumnInfo. The ColumnInfo annotation defines information about the column in the database table, such as the column name, data type and default value. For example, in the above code, the ColumnInfo annotation specifies that the track parameter will be stored in a column called song\_track.

## Configuring the Room SQLite database

We'll now create a class that will configure the Room database. Right-click the folder that contains the MainActivity class (**Project > app > java > name of the project**) and select **New > Kotlin File/Class**. Name the file MusicDatabase and select Class from the list of options. Once the **MusicDatabase.kt** file opens in the editor, modify its code so it reads as follows:

```
@Database(entities = [Song::class], version = 1, exportSchema = false)
abstract class MusicDatabase : RoomDatabase() {
```

```

abstract fun musicDao(): MusicDao

companion object {
    @Volatile
    private var database: MusicDatabase? = null

    fun getDatabase(context: Context): MusicDatabase {
        database ?: kotlin.run {
            database = Room.databaseBuilder(context, MusicDatabase::class.java, "music_database")
                .build()
        }
    }

    return database!!
}
}
}

```

The above code states that the database will contain one entity, as mapped by the Song class. Interactions with the Song entity will be handled by a data access object called MusicDAO that we will create later. Inside the companion object, there is a method called getDatabase, which builds the Room database and names it music\_database.

### How to prepopulate a Room database entity with data

It is possible to automatically add entries to a Room database entity the moment it is created. To demonstrate how this works, we will configure the Song entity to always contain an entry for an imaginary song called “Guitar solo”. The first step is to modify the getDatabase method so it reads as follows:

```

fun getDatabase(
    context: Context,
    scope: CoroutineScope
): MusicDatabase {
    database ?: kotlin.run {
        database = Room.databaseBuilder(context, MusicDatabase::class.java, "music_database")
            .addCallback(MusicDatabaseCallback(scope))
            .build()
    }
}

return database!!
}

```

In the above code, an argument called scope is added to the getDatabase method. The scope variable will contain a CorourtineScope instance that will allocate database interactions to an alternative worker thread behind the scenes. In this app, the getDatabase method will be called by a view model that we will create. View models have a native coroutine scope called viewModelScope, so to run the getDatabase method with the new set of parameters, you could write the following:

```
MusicDatabase.getDatabase(application, viewModelScope)
```

Next, a callback called MusicDatabaseCallback is added to the Room database builder. The MusicDatabaseCallback class will detect when the Room database is up and running and insert the data we need.

```

private class MusicDatabaseCallback(private val scope: CoroutineScope) :
    RoomDatabase.Callback() {

    override fun onCreate(db: SupportSQLiteDatabase) {
        super.onCreate(db)
        database?.let { database ->
            scope.launch {

```

```

// Prepopulate the Song entity with a readymade Song object
database.musicDao().insert(Song(1,1001, "Guitar solo", "Guitarist", "Greatest Hits",
"22", "content://010101", "2021"))
}
}
}
}
}

```

Note you may need to add the following import statement to the top of the file:

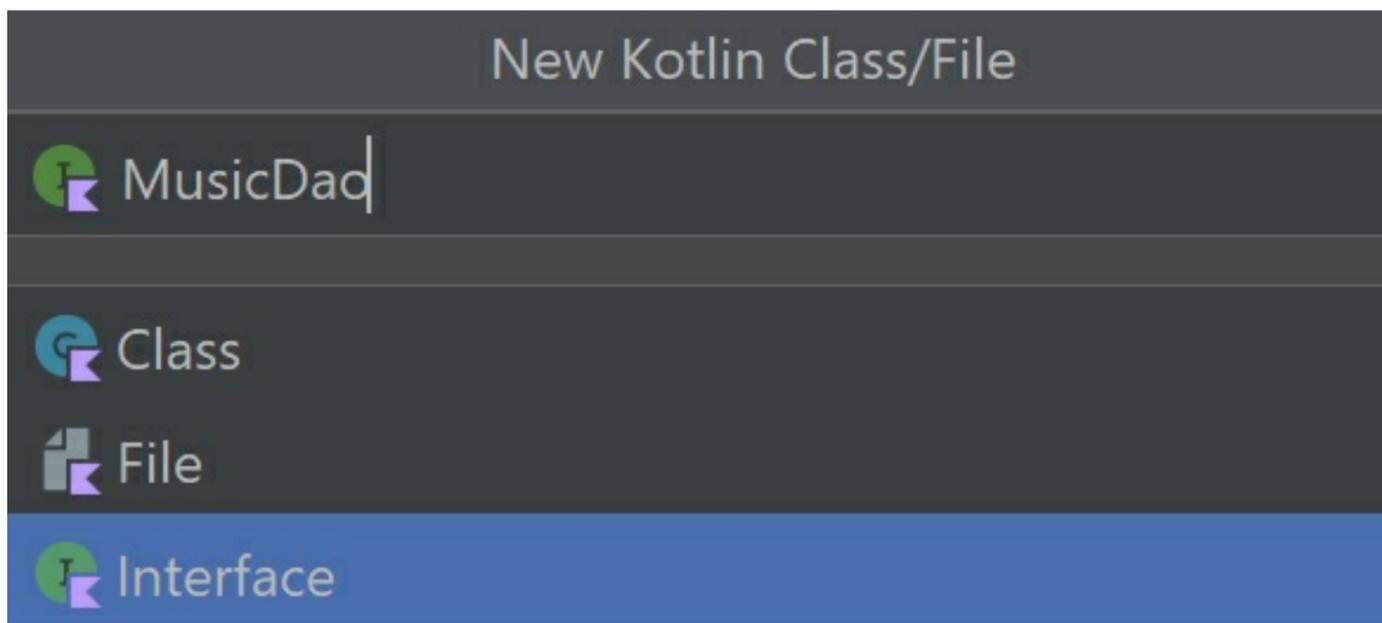
```
import kotlinx.coroutines.launch
```

The MusicDatabaseCallback class contains instructions that will be carried out at specific stages of the database lifecycle. For example, in the above code, the onCreate method is used to define what happens when all of the database tables have been created. Other database states you can override include: onOpen, which occurs when the database is opened; and onDestructiveMigration, which defines what happens if you attempt to update a Room entity (e.g. with columns added or removed) but Room is unable to successfully migrate data from the previous database configuration to the new database configuration. For more information about the database state callback methods see the official Android documentation <https://developer.android.com/reference/androidx/room/RoomDatabase.Callback>.

In this instance, we only wish to override the onCreate database state and use the opportunity to pre-populate the Song entity with a song called “Guitar solo”. This means when the database is created it will already have a readymade Song object. The completed Song object is inserted into the Room database using a MusicDAO method called insert. We’ll cover how to interact with the database in the next section.

## The music data access object (DAO)

Once the database has been created, you can interact with its entities and insert, delete, update and retrieve data. All database interactions are handled by a data access object (DAO). To make a DAO, right-click the folder that contains the MainActivity class (**Project** > **app** > **java** > **name of the project**) and select **New** > **Kotlin File/Class**. Select Interface from the list of options and name the interface MusicDao.



Once the **MusicDao.kt** file opens in the editor, edit its code so it reads as follows:

```

@Dao
interface MusicDao {

    @Query("SELECT * from music_table ORDER BY song_title ASC")
    fun getAlphabetizedSongs(): LiveData<List<Song>>

    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insert(song: Song)

    @Query("SELECT * from music_table WHERE song_album_id LIKE :albumID LIMIT 1")
    suspend fun doesAlbumIDExist(albumID: String): List<Song>
}

```

```

@Query("SELECT * FROM music_table WHERE song_title LIKE :search OR song_artist LIKE :search OR
song_album LIKE :search")
suspend fun findBySearchSongs(search: String): List<Song>

@update(onConflict = OnConflictStrategy.REPLACE)
suspend fun updateMusicInfo(song: Song)

@Delete
suspend fun delete(song: Song)
}

```

The above code details every query we will look to run on the music table. The role of each query will be discussed in greater depth when we use them in the code. For now, we'll discuss Room queries more generally so you can write your own if you wish!

The first thing to note is that Room DAOs offer several readymade "convenience" methods which allow you to easily perform routine tasks. The first method we will discuss is called Insert, which adds a new entry/row to the database table. For example, the following method will insert a Song object into the music table:

```

@Insert(onConflict = OnConflictStrategy.IGNORE)
suspend fun insert(song: Song)

```

The above code references an OnConflictStrategy. An OnConflictStrategy defines how Room will handle a request where two entries have the same primary key value (remember each entry must have a unique primary key). In this instance, the OnConflictStrategy is IGNORE. This tells Room not to insert Song objects into the database unless they have a unique primary key. Alternative OnConflictStrategy approaches include REPLACE, which replaces the previous entry for that primary key with the new information, and ABORT. The ABORT strategy is similar to IGNORE, in that conflicting entries will not be processed by Room; however, the ABORT strategy also reverses any other changes which were processed as part of the query. The ABORT strategy is only really applicable to complex queries.

Other convenience methods include:

**Delete:** which will remove a Song object from the database table:

```

@Delete
suspend fun delete(song: Song)

```

**Update:** will replace an entry in the database. The outgoing and incoming object must share the same primary key value; however, all other details can be different:

```

@update(onConflict = OnConflictStrategy.REPLACE)
suspend fun update(song: Song)

```

Room databases also allow you to select and retrieve entries that meet certain conditions. For example, imagine you wanted to retrieve all the Song objects which contain a reference to "Oasis". In the music DAO we have a function that does just that called findBySearchSongs:

```

@Query("SELECT * FROM music_table WHERE song_title LIKE :search OR song_artist LIKE :search OR
song_album LIKE :search")
suspend fun findBySearchSongs(search: String): List<Song>

```

The above query uses a SELECT command to retrieve all entries that have either a title, artist or album name containing the text specified in the "search" query variable. Results are reported as a live data list of Song objects, which means the query can issue a new set of results if the underlying data changes.

SELECT queries can be refined further. For instance, in the above example, we organise the list of songs in alphabetical order by adding ORDER BY song\_title ASC to the end of the query. Also, you can restrict the size of the list to 10 songs by adding LIMIT 10 to the end of the query:

```

@Query("SELECT * FROM music_table WHERE song_title LIKE :search OR song_artist LIKE :search OR
song_album LIKE :search ORDER BY song_title ASC LIMIT 10")
suspend fun findBySearchSongs(search: String): List<Song>

```

There are many different routes you can go down when writing Room queries. If you would like to learn more then you may find the official SQLite documentation useful: <https://www.sqlite.org/lang.html>

## The database repository

In this section, we'll turn our attention to the database repository. The repository will process Room DAO queries while keeping the underlying work of the database separate from the rest of the app. To set up the repository, create a new Kotlin class in the same folder as MainActivity, name it MusicRepository and insert the following code:

```
class MusicRepository(private val musicDao: MusicDao) {  
  
    val allSongs: LiveData<List<Song>> = musicDao.getAlphabetizedSongs()  
  
    suspend fun insertSong(song: Song) {  
        musicDao.insert(song)  
    }  
  
    suspend fun deleteSong(song: Song) {  
        musicDao.delete(song)  
    }  
  
    suspend fun updateMusicInfo(song: Song){  
        musicDao.updateMusicInfo(song)  
    }  
}
```

The MusicRepository class contains a variable called allSongs that will store a list of every song in the user's music library. The allSongs variable stores the list of songs in LiveData format, which means the contents of the variable can be observed from elsewhere in the app. All observers will be notified whenever the underlying data changes, such as when Song objects are added or removed from the list.

Next, several methods are defined. Each method features the suspend precursor, which means the method can be paused and resumed. Methods that feature the suspend precursor must be launched from a coroutine (a mechanism for handling tasks behind the scenes). The methods declared above perform several routine operations including inserting Song objects into the database, deleting entries and updating entries.

## The music and playback view models

The database is almost fully operational. All that remains is to create a view model, which will manage data and help the app coordinate tasks. The view model will make the songs in the user's music library available to different areas of the app and handle requests to insert, delete, and update database entries.

Create a new Kotlin class in the same folder as MainActivity, name it MusicViewModel and construct the class using the following code:

```
import androidx.lifecycle.ViewModelScope  
import kotlinx.coroutines.launch  
  
// we use AndroidViewModel so we can request application context  
class MusicViewModel(application: Application) : AndroidViewModel(application) {  
  
    private val repository: MusicRepository  
    val allSongs: LiveData<List<Song>>  
  
    init {  
        val musicDao = MusicDatabase.getDatabase(application).musicDao()  
        repository = MusicRepository(musicDao)  
        allSongs = repository.allSongs  
    }  
  
    fun deleteSong(song: Song) = viewModelScope.launch(Dispatchers.IO) {  
        repository.deleteSong(song)  
    }  
  
    fun insertSong(song: Song) = viewModelScope.launch(Dispatchers.IO) {  
        repository.insertSong(song)  
    }  
  
    fun updateMusicInfo(song: Song) = viewModelScope.launch(Dispatchers.IO) {  
        repository.updateMusicInfo(song)  
    }  
}
```

In the above code, the init block connects to the music repository and stores the contents of the repository's allSongs variable in a view model variable of the same name. Like the repository's allSongs variable, the view model's allSongs variable will store the list of songs in LiveData format. This means the contents of the list will automatically update whenever the underlying data in the repository (and hence the database) changes. The remainder of the view model comprises several methods for inserting, deleting and updating entries in the Song database entity via the repository. The view model's methods will be accessible to all areas of the app, so any activities or fragments looking to interact with the database can do so via the view model.

You may notice that each view model method uses a viewModelScope to launch the request to the repository. The viewModelScope allows methods to run behind the scenes as a coroutine, using different worker threads to complete the tasks in a resource-efficient manner. If you did not use coroutines, then the app would complete the task on the main thread, which may cause the app to freeze until the work is complete. Coroutines launched using the viewModelScope are automatically cancelled when the view model is destroyed. For this reason, if the view model is closed while an operation is ongoing then it will need to be restarted when the view model is next open.

In addition, there is a further view model we need to create called PlaybackViewModel. The PlaybackViewModel view model will store information relating to playback such as the play queue, playback progress and duration of the currently playing song. Create a new Kotlin class in the same folder as MainActivity, name it PlaybackViewModel and add the following code:

```
class PlaybackViewModel : ViewModel() {  
    var currentPlayQueue = MutableLiveData<List<Pair<Int, Song>>>()  
    var currentPlaybackDuration = MutableLiveData<Int>()  
    var currentPlaybackPosition = MutableLiveData<Int>()  
    var currentlyPlayingQueueID = MutableLiveData<Int>()  
    var currentlyPlayingSong = MutableLiveData<Song?>()  
    var isPlaying = MutableLiveData<Boolean>()  
}
```

The PlaybackViewModel view model will share information relating to the current playback state across the app. All variables in the PlaybackViewModel class store data in MutableLiveData format, which like LiveData, means the variables can be observed by other areas of the app. The difference between MutableLiveData and LiveData is that MutableLiveData can be modified.

You may notice the currentPlayQueue variable holds a list of Pair objects. The Pair data type is used to store data as value-value pairs. In the above code, we specify the first value in the Pair will be an integer and the second value will be a Song object. The integer will equal the index of the Song object when it was originally added to the play queue. In this way, we can sort the integer values and restore the play queue to the original order, even if the list has been shuffled or items have been added and removed.

## The app toolbar menu

To help the user interact with the app, we will create a toolbar menu that allows the user to search their music library and perform other actions depending on their current destination in the app. The code for the menu items will be stored in a file called **main.xml**, which should have been automatically generated by Android Studio. Locate and open the file by navigating through **Project > app > res > menu**, then replace the contents of the menu element with the following items:

```
<item android:id="@+id/search"  
    android:title="@string/search_hint"  
    android:icon="@drawable/ic_search"  
    app:actionViewClass="android.widget.SearchView"  
    app:showAsAction="ifRoom" />
```

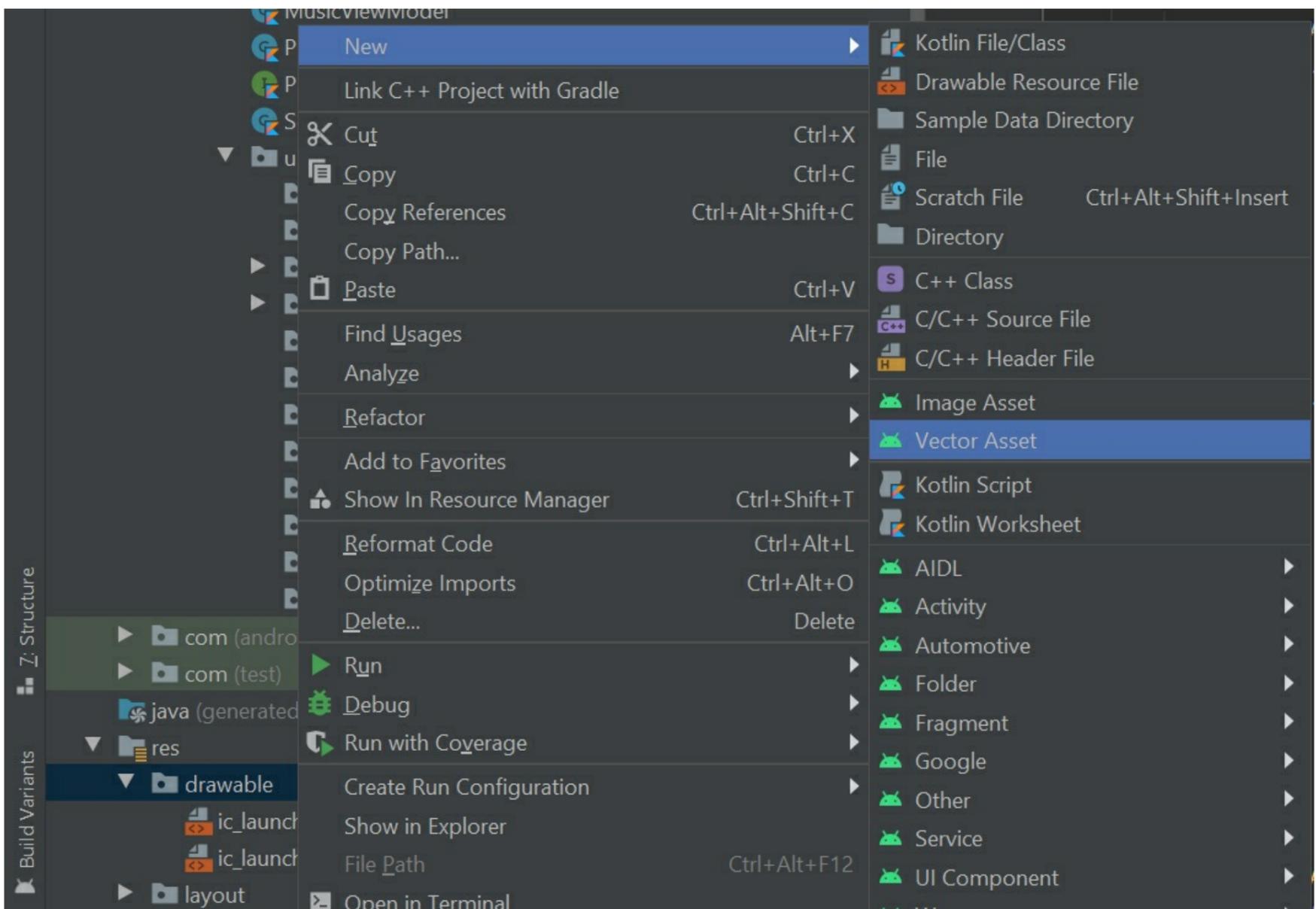
```
<item android:id="@+id/save"  
    android:title="@string/save"  
    android:visible="false"  
    app:showAsAction="ifRoom" />
```

```
<item android:id="@+id/done"  
    android:title="@string/done"  
    android:visible="false"  
    app:showAsAction="ifRoom" />
```

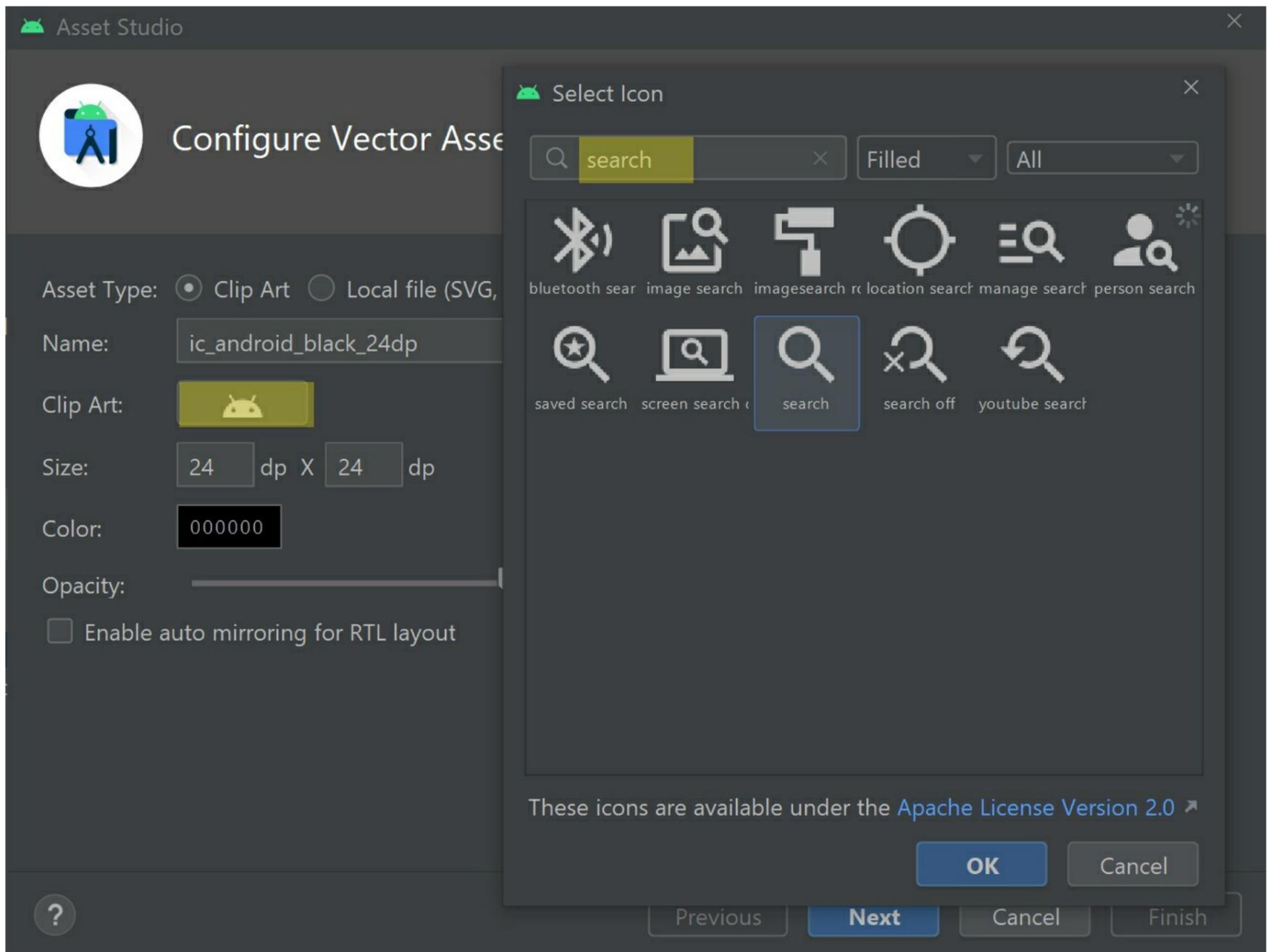
The above code defines menu items for a search icon, a save button and a done button. You may notice that the save and done menu items have a visible attribute set to false. The reason for this is that those items will be hidden by default, and will only be used by certain fragments. The search icon item is not hidden because the search functionality will be available in most areas of the app. Each menu item contains a showAsAction attribute that determines whether Android should try and insert the item into the app bar itself (ifRoom) or the overflow menu (never). When you copy the above code into your project, the reference to the search icon `@drawable/ic_search` may be highlighted in red. This is because the drawable resource file (which contains the icon image) does not exist yet. We'll address that in the next section.

## Defining the drawable resources used in the project

The images and icons that are used in the app must be defined as drawable resources. For example, you could create a drawable resource for a search icon that the user can click to find music in their library. To create a new drawable resource, navigate through **Project** > **app** > **res** then right-click the **drawable** directory and select **New** > **Vector Asset**.



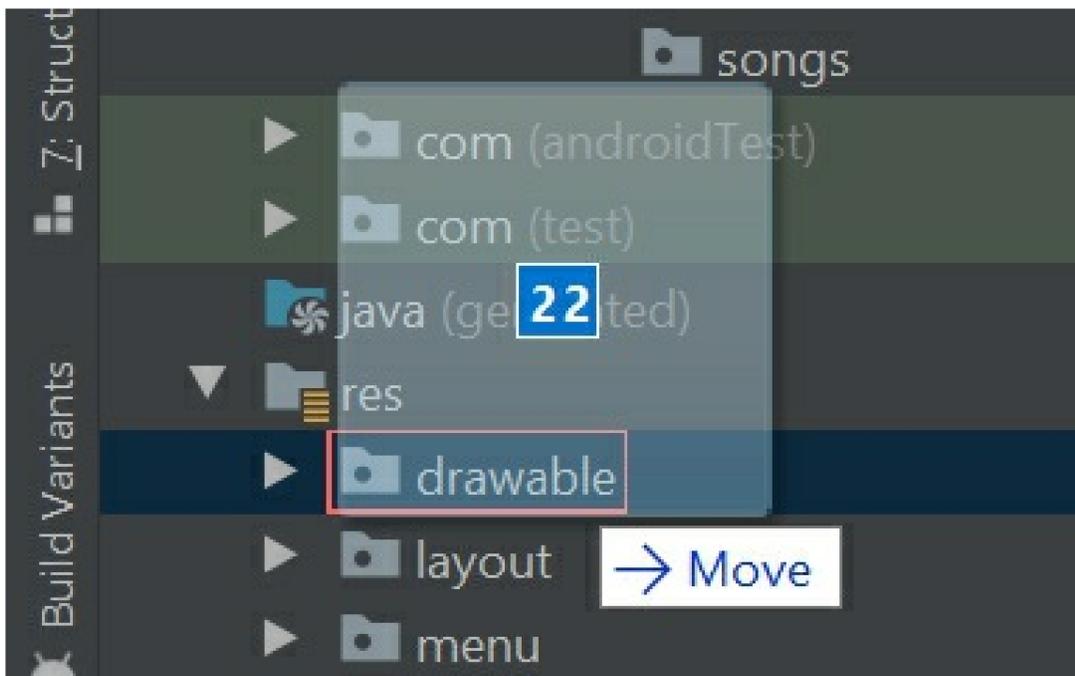
In the Configure Vector Asset window, click the clip art image then locate the search icon and click OK.



Set the name of the asset to `ic_search` then click Next and Finish to save the icon as a drawable resource. In this project, we will use over 20 different drawable resources. It would be quite tedious to create these resources manually so instead I would invite you to copy the source files from the example code into the **drawable** directory of your project. To locate the drawable files in the example code, navigate through **app > src > main > res > drawable** and copy the files highlighted below. You could also copy the unhighlighted files over; some are default drawable files created by Android Studio and others are custom files we will discuss later.

<input type="checkbox"/> Name	Date modified	Type	Size
<input checked="" type="checkbox"/>  ic_add	07/07/2020 20:16	XML Document	
<input checked="" type="checkbox"/>  ic_arrow_forward	24/07/2020 13:15	XML Document	
<input checked="" type="checkbox"/>  ic_back	03/07/2020 19:01	XML Document	
<input checked="" type="checkbox"/>  ic_down	15/08/2020 13:40	XML Document	
<input checked="" type="checkbox"/>  ic_drag_handle	19/06/2020 00:43	XML Document	
<input checked="" type="checkbox"/>  ic_edit	24/06/2020 21:12	XML Document	
 ic_launcher_background	08/02/2022 10:53	XML Document	
 ic_menu_camera	08/02/2022 10:53	XML Document	
 ic_menu_gallery	08/02/2022 10:53	XML Document	
 ic_menu_slideshow	08/02/2022 10:53	XML Document	
<input checked="" type="checkbox"/>  ic_more	03/07/2020 16:19	XML Document	
<input checked="" type="checkbox"/>  ic_nav_playing	09/06/2020 23:19	XML Document	
<input checked="" type="checkbox"/>  ic_nav_songs	09/06/2020 23:37	XML Document	
<input checked="" type="checkbox"/>  ic_next	03/07/2020 16:07	XML Document	
<input checked="" type="checkbox"/>  ic_pause	03/07/2020 19:03	XML Document	
<input checked="" type="checkbox"/>  ic_photo	09/08/2020 16:05	XML Document	
<input checked="" type="checkbox"/>  ic_play	03/07/2020 19:03	XML Document	
<input checked="" type="checkbox"/>  ic_queue	07/07/2020 14:48	XML Document	
<input checked="" type="checkbox"/>  ic_refresh	15/07/2020 00:04	XML Document	
<input checked="" type="checkbox"/>  ic_repeat	27/06/2020 12:33	XML Document	
<input checked="" type="checkbox"/>  ic_repeat_one	07/07/2020 12:27	XML Document	
 ic_search	09/02/2022 09:56	XML Document	
<input checked="" type="checkbox"/>  ic_settings	15/08/2020 13:41	XML Document	
<input checked="" type="checkbox"/>  ic_shuffle	17/07/2020 20:17	XML Document	
<input checked="" type="checkbox"/>  pause	23/07/2020 01:01	XML Document	
<input checked="" type="checkbox"/>  play	23/07/2020 01:01	XML Document	
 side_nav_bar	08/02/2022 10:53	XML Document	

To copy files into your Android Studio project, simply drag and drop them into the **drawable** directory as shown below:



Most of the icons used in this project are open source Material Design icons (see <https://material.io/resources/icons>), except for the following files:

- **ic\_play.xml**
- **ic\_pause.xml**
- **ic\_next.xml**
- **Ic\_back.xml**

These icons were made by Elias Bikbulatov from [www.flaticon.com](http://www.flaticon.com) and are free for commercial use with attribution.

## The media browser service - part 1

The Music app will use a service to manage audio playback. The service will handle audio files, display notifications, respond to Bluetooth devices and more. To set up the service, create a new Kotlin class in the same folder as MainActivity (**Project** > **app** > **java** > **name of the project**) and name it **MediaPlayerService**. Once the **MediaPlayerService.kt** file opens in the editor, add the following code to define the **MediaPlayerService** class and its variables:

```
import android.media.AudioManager.*
import android.os.Handler

class MediaPlayerService : MediaBrowserServiceCompat() {

    private val channelId = "music"
    private var currentlyPlayingSong: Song? = null
    private val handler = Handler(Looper.getMainLooper())
    private val logTag = "AudioPlayer"
    private var mediaPlayer: MediaPlayer? = null
    private lateinit var audioFocusRequest: AudioFocusRequest
    private lateinit var mMediaSessionCompat: MediaSessionCompat

    private val mMediaSessionCallback: MediaSessionCompat.Callback = object : MediaSessionCompat.Callback() {
        // TODO: Playback actions will be defined here
    }

    private val mNoisyReceiver = object : BroadcastReceiver() {
        override fun onReceive(context: Context, intent: Intent) {
            if (mMediaPlayer != null && mMediaPlayer!!.isPlaying) mMediaSessionCallback.onPause()
        }
    }

    override fun onCreate() {
        super.onCreate()

        mMediaSessionCompat = MediaSessionCompat(baseContext, logTag).apply {
            setCallback(mMediaSessionCallback)
        }
    }
}
```

```

        setSessionToken(sessionToken)
    }
    val filter = IntentFilter(ACTION_AUDIO_BECOMING_NOISY)
    registerReceiver(mNoisyReceiver, filter)
}

override fun onGetRoot(clientPackageName: String, clientId: Int, rootHints: Bundle?): BrowserRoot? {
    return if (TextUtils.equals(clientPackageName, packageName)) {
        BrowserRoot(getString(R.string.app_name), null)
    } else null
}

override fun onLoadChildren(parentId: String, result: Result<List<MediaBrowserCompat.MediaItem>>) {
    result.sendResult(null)
}
}

```

In the above code, the `MediaPlayerService` class extends the `MediaBrowserServiceCompat` class, which provides the `MediaPlayerService` class with all the data required to behave as a media browser service and coordinate media playback. When the service starts, the `onCreate` stage of the service lifecycle will run. In the above code, we override the `onCreate` method and initialise a variable called `mMediaSessionCompat`. The `mMediaSessionCompat` variable will contain an instance of the `MediaSessionCompat` class, which we configure to handle playback-related processes. First, the `MediaSessionCompat` is linked with a callback variable called `mMediaSessionCallback`, which we will later configure to support playback functions such as play, pause, skip forward etc. Next, we assign the `MediaSessionCompat` instance a session token, which identifies the media session and allows us to coordinate session processes such as notifications.

At the end of the `onCreate` method, a broadcast receiver variable called `mNoisyReceiver` is registered. This broadcast receiver will detect when audio becomes “noisy”, which occurs when earphones are unplugged and playback would otherwise continue to play through the device speakers. In the broadcast receiver’s `onReceive` callback method, we respond to earphones becoming unplugged by pausing playback.

After the `onCreate` method, we implement two methods called `onGetRoot` and `onLoadChildren`. Both methods are mandatory when extending the `MediaBrowserServiceCompat`, however, we will not use them in the Music app. Typically, the `onGetRoot` method would return information about the service client, while the `onLoadChildren` method returns data about a media item’s child elements.

Moving on, let’s define the playback actions that will be coordinated by the `MediaSessionCompat` callback. Locate the `mMediaSessionCallback` variable and replace the `TODO` comment with the following code:

```

override fun onMediaButtonEvent(mediaButtonEvent: Intent?): Boolean {
    val ke: KeyEvent? = mediaButtonEvent?.getParcelableExtra(Intent.EXTRA_KEY_EVENT)
    if (ke != null && mMediaPlayer != null) {
        when (ke.keyCode) {
            KeyEvent.KEYCODE_MEDIA_PLAY_PAUSE -> {
                if (mMediaPlayer!!.isPlaying) onPause()
                else onPlay()
            }
            KeyEvent.KEYCODE_MEDIA_PLAY -> onPlay()
            KeyEvent.KEYCODE_MEDIA_PAUSE -> onPause()
            KeyEvent.KEYCODE_MEDIA_SKIP_BACKWARD -> onSkipToPrevious()
            KeyEvent.KEYCODE_MEDIA_SKIP_FORWARD -> onSkipToNext()
        }
    }
    return super.onMediaButtonEvent(mediaButtonEvent)
}

```

The `onMediaButtonEvent` callback method defines how the service should respond to buttons on external hardware such as Bluetooth earphones. Each type of button and action is linked with a `keyCode` and the above method links these `keyCodes` with the appropriate response. For example, the `KEYCODE_MEDIA_PLAY` `keyCode` initiates playback via the `onPlay` method (which we’ll define shortly). You can find a full list of media `keyCodes` in the official Android documentation:

[https://developer.android.com/reference/android/view/KeyEvent.html#KEYCODE\\_MEDIA\\_AUDIO\\_TRACK](https://developer.android.com/reference/android/view/KeyEvent.html#KEYCODE_MEDIA_AUDIO_TRACK)

To define the callback method that prepares audio files for playback, add the following code below the `onMediaButtonEvent` method:

```
override fun onPrepareFromUri(uri: Uri?, extras: Bundle?) {
    super.onPrepareFromUri(uri, extras)
    val bundle = extras!!.getString("song")
    val type = object : TokenType<Song>() {}.type
    currentlyPlayingSong = Gson().fromJson(bundle, type)
    setCurrentMetadata()

    if (mMediaPlayer != null) mMediaPlayer!!.release()
    try {
        mMediaPlayer = MediaPlayer().apply {
            setAudioAttributes(
                AudioAttributes.Builder()
                    .setUsage(AudioAttributes.USAGE_MEDIA)
                    .setContentType(AudioAttributes.CONTENT_TYPE_MUSIC)
                    .build()
            )
            setDataSource(application, Uri.parse(currentlyPlayingSong!!.uri))
            setOnErrorListener(this@MediaPlaybackService)
            prepare()
            // Refresh the notification so user can see song has changed
            showNotification(false)
        }
    } catch (e: IOException) {
        onError(mMediaPlayer, MEDIA_ERROR_UNKNOWN, MEDIA_ERROR_IO)
    } catch (e: IllegalStateException) {
        onError(mMediaPlayer, MEDIA_ERROR_UNKNOWN, MEDIA_ERROR_IO)
    } catch (e: IllegalArgumentException) {
        onError(mMediaPlayer, MEDIA_ERROR_UNKNOWN, MEDIA_ERROR_MALFORMED)
    }
}
```

Note you should ensure the following import statement is included at the top of the file:

```
import android.media.MediaPlayer.*
```

The `onPrepareFromUri` method will be called whenever a new song should be loaded into the service. The method requires two arguments to be supplied: a URI that's associated with the filepath of the audio file, and a `Bundle` containing extra information about the song. In this app, we will package the `Song` object associated with the audio file into the `Bundle`. Unfortunately, we cannot add the `Song` object directly to the `Bundle` because the `Song` data class is not a primitive data type like a `String` or an `Integer`. For this reason, before being packaged into the bundle, the `Song` object will be converted to a JSON string. This means the `onPrepareFromUri` method must restore the JSON string to a `Song` object using GSON. GSON is a package that converts JSON strings into other forms of data, including custom classes like the `Song` data class.

Next, the song URI is loaded into an instance of the `MediaPlayer` class using the `setDataSource` method. If the `MediaPlayer` instance has already been initialised then the `release` method will remove any previously loaded URIs. Finally, we state that the `MediaPlaybackService` class will handle any playback errors and ready the media player for playback using the `prepare` method. Also, a method that we will define later called `showNotification` is run. The `showNotification` method will display a notification containing the details of the currently playing song.

Much of the code in the `onPrepareFromUri` method is wrapped in a `try` block to catch several errors that may occur when the `MediaPlayer` instance is being prepared. For example, the media player may be in an illegal state or the source file may no longer be accessible. If a problem occurs, then a method called `error` will run to reset the media player and notify the user. When designing `try/catch` blocks yourself, you can find a list of potential exceptions (errors) that the methods you're using may throw in the Android documentation. For example, a list of exceptions that the `setDataSource` method can throw is listed here:

[https://developer.android.com/reference/android/media/MediaPlayer#setDataSource\(android.content.Context,%20ar](https://developer.android.com/reference/android/media/MediaPlayer#setDataSource(android.content.Context,%20ar)

The `MediaPlaybackService` class must be capable of handling media player errors. To facilitate this, edit the `MediaPlaybackService` class declaration so it extends the `MediaPlayer` class's `OnErrorListener` interface, as shown below:

```
class MediaPlayerService : MediaBrowserServiceCompat(), OnErrorListener {
```

To extend the OnErrorListener interface successfully, we must implement a method called onError. To achieve this, add the following code below the onCreate method:

```
override fun onError(mp: MediaPlayer?, what: Int, extra: Int): Boolean {  
    mMediaPlayer?.reset()  
    mMediaPlayer = null  
    setMediaPlayerState(PlaybackStateCompat.STATE_STOPPED, 0L, 0f, null)  
    currentlyPlayingSong = null  
    stopForeground(true)  
    Toast.makeText(application, getString(R.string.error), Toast.LENGTH_LONG).show()  
    return true  
}
```

The onError method requires three arguments to be supplied: the MediaPlayer instance associated with the error (or null if no MediaPlayer instance involved); the type of media error that has occurred, which can be either MEDIA\_ERROR\_UNKNOWN for unspecified errors or MEDIA\_ERROR\_SERVER\_DIED for server errors that require the media player to be recreated; and an extra error-specific code. From referring to the documentation for the OnErrorListener interface (<https://developer.android.com/reference/android/media/MediaPlayer.OnErrorListener>), we can see the error-specific code can be one of several values including MEDIA\_ERROR\_IO, MEDIA\_ERROR\_MALFORMED, MEDIA\_ERROR\_UNSUPPORTED and MEDIA\_ERROR\_TIMED\_OUT. You could use the error code to determine the cause of the error and respond accordingly.

In this app, we will respond to all media player errors the same way. The onError method defined above resets the MediaPlayer instance, removes the currently playing song and sets the playback state to STATE\_STOPPED. The onError method also removes the media browser service from the foreground state, which closes the media player notification and signals to the device that the service can be closed if processing power needs to be reallocated elsewhere. Finally, the onError method displays a toast notification advising the user an error has occurred. Most often, media player errors will occur if the user attempts to play an audio file that has been deleted or moved. The app automatically scans for deleted songs but this process takes a little bit of time. Even if an error occurs, the user can continue to attempt to play other songs.

The playback state of the service should be broadcast to the other areas of the app so the user interface can be updated and other application processes can respond accordingly. Changes in playback state will be broadcast by a method called setMediaPlayerState, which you can define by adding the following code below the onError method:

```
private fun setMediaPlayerState(state: Int, position: Long, playbackSpeed: Float, bundle: Bundle?) {  
    val playbackState = PlaybackStateCompat.Builder()  
        .setState(state, position, playbackSpeed)  
        .setExtras(bundle)  
        .build()  
    mMediaSessionCompat.setPlaybackState(playbackState)  
}
```

The setMediaPlayerState method sets the playback state to the desired value and also attaches any extra data such as the current playback position, playback speed and extras bundle. Later in the project, we will configure the MainActivity class to respond to changes in the playback state and execute any necessary actions such as preparing a song for playback and updating components of the user interface (e.g. changing the progress of the seekbar, alternating the play/pause button etc.).

Let's now return to the mMediaSessionCallback variable and continue to define the media player callback actions. To handle requests to initiate or resume playback, add the following code below the onPrepareFromUri method:

```
override fun onPlay() {  
    super.onPlay()  
    if (currentlyPlayingSong != null) {  
        val audioManager = applicationContext.getSystemService(Context.AUDIO_SERVICE) as AudioManager  
        // Request audio focus for playback  
        audioFocusRequest = AudioFocusRequest.Builder(AUDIOFOCUS_GAIN).run {  
            setAudioAttributes(AudioAttributes.Builder().run {  
                setOnAudioFocusChangeListener(afChangeListener)  
            })  
        }  
    }  
}
```



current playback position and duration of the currently playing song are packaged into a bundle and delivered to any components which are monitoring the service via a media playback state update. The onPlay method also attaches an onComplete listener to the media player, which will run whenever a song finishes. In which case, the playback state is set to STATE\_SKIPPING\_TO\_NEXT, which the MainActivity class will interpret as a signal to load the next song in the play queue (more on this later).

Moving on, add the following code below the onPlay method to handle requests to pause audio playback:

```
override fun onPause() {
    super.onPause()
    if (mMediaPlayer != null && mMediaPlayer!!.isPlaying) {
        mMediaPlayer!!.pause()
        val playbackPosition = mMediaPlayer!!.currentPosition.toLong()
        val playbackDuration = mMediaPlayer!!.duration
        val bundle = Bundle()
        bundle.putInt("duration", playbackDuration)
        setMediaPlaybackState(PlaybackStateCompat.STATE_PAUSED, playbackPosition, 0f, bundle)
        showNotification(false)
    }
}
```

The above code pauses the media player and packages the current playback position in a bundle. The bundle is communicated to areas of the app that are monitoring the service via a playback state update. Also, the showNotification method is run to refresh the notification and replace the pause button with a play button, so the user can resume playback from the notification bar if they wish.

Next, add the following code below the onPause method to respond to requests to skip to the next and previous tracks in the play queue, respectively:

```
override fun onSkipToNext() {
    super.onSkipToNext()

    setMediaPlaybackState(PlaybackStateCompat.STATE_SKIPPING_TO_NEXT, 0, 0f, null)
}
```

```
override fun onSkipToPrevious() {
    super.onSkipToPrevious()

    // currentPosition returns the playback position in milliseconds
    // if the media player is more than 5 seconds into song then restart the song, otherwise, skip back to the previous song
    if (mMediaPlayer != null && mMediaPlayer!!.currentPosition > 5000) onSeekTo(0)
    else setMediaPlaybackState(PlaybackStateCompat.STATE_SKIPPING_TO_PREVIOUS, 0, 0f, null)
}
```

The onSkipToNext and onSkipToPrevious methods are similar but perform opposing actions. Both methods update the playback state, which instructs the MainActivity class to load the previous or next song in the play queue, as appropriate. However, the onSkipToPrevious method contains additional code to check whether the media player has progressed further than five seconds into the current song. If more than five seconds has elapsed, then the onSkipToPrevious method will restart the current song rather than skip back to the previous track.

Moving on, to handle requests to stop playback, add the following code below the onSkipToPrevious method:

```
override fun onStop() {
    super.onStop()
    if (mMediaPlayer != null) {
        mMediaPlayer!!.stop()
        mMediaPlayer!!.release()
        mMediaPlayer = null
        currentlyPlayingSong = null
        stopForeground(true)
    }
    try {
        val audioManager = getSystemService(Context.AUDIO_SERVICE) as AudioManager
        audioManager.abandonAudioFocusRequest(audioFocusRequest)
    }
}
```

```

    } catch (ignore: UninitializedPropertyAccessException){ }
}
setMediaPlaybackState(PlaybackStateCompat.STATE_STOPPED, 0L, 0f, null)
stopSelf()
}

```

The onStop method defined above will run when the play queue has ended or the app is closed and the service must shut down. It stops and resets the media player, removes the media player service and notification from the foreground, abandons the media focus, and sets the playback state to STATE\_STOPPED. The code that abandons the audio focus is wrapped in a try/catch block because there is a chance the service could be opened and closed before the audioFocusRequest variable is initialised. Finally, the above code closes the service using the stopSelf method.

The final callback method we will define is onSeekTo, which will handle requests to seek a specific playback position. To define the onSeekTo method, add the following code below the onStop method:

```

override fun onSeekTo(pos: Long) {
    super.onSeekTo(pos)
    mMediaPlayer?.let {
        if (it.isPlaying) {
            it.pause()
            it.seekTo(pos.toInt())
            it.start()
            val playbackPosition = it.currentPosition.toLong()
            setMediaPlaybackState(PlaybackStateCompat.STATE_PLAYING, playbackPosition, 1f, null)
        } else {
            it.seekTo(pos.toInt())
            val playbackPosition = it.currentPosition.toLong()
            setMediaPlaybackState(PlaybackStateCompat.STATE_PAUSED, playbackPosition, 0f, null)
        }
    }
}
}

```

The onSeekTo method will set the playback position of the media player to the value specified in the method's pos argument. The code which performs this action is wrapped in a let block that is attached to the mMediaPlayer variable, which means the code will only be executed if the MediaPlayer instance is not null. It is important to wrap the code in a let block because attempts to interact with a null MediaPlayer instance could cause an exception. If the song is currently playing then it must be temporarily paused while the playback position is changed because the audio may become distorted. The new playback position is conveyed to other areas of the app by updating the playback state.

## The media browser service - part 2

In this section, we will continue to work on the media browser service and enable further functionality. For example, during playback, the service should regularly communicate the playback progress to the MainActivity class. The MainActivity class can use the playback position data to update a ProgressBar widget that will be included in the playback controls. To monitor the playback position, we will create a Runnable object that checks the playback progress at regular intervals. The Runnable object will be stored in a variable called playbackPositionChecker, which you can define by adding the following code to the list of variables at the top of the MediaPlayerService class:

```

private var playbackPositionChecker = object : Runnable {
    override fun run() {
        try {
            if (mMediaPlayer != null && mMediaPlayer!!.isPlaying) {
                val playbackPosition = mMediaPlayer!!.currentPosition.toLong()
                setMediaPlaybackState(PlaybackStateCompat.STATE_PLAYING, playbackPosition, 1f, null)
            }
        } finally {
            handler.postDelayed(this, 1000L)
        }
    }
}
}

```

```
}
```

The Runnable object defined in the playbackPositionChecker variable contains a callback method called run that will repeat at regular intervals. The run method retrieves the playback position of the currently playing song from the media player and communicates the playback position to other areas of the app by updating the playback state. An if expression is used to ensure the playback state is only updated when playback is in progress because it is not necessary to send updates if playback is paused or stopped. Once the playback state has been updated, the Handler class's postDelayed method runs the Runnable task again after a 1000 ms delay. Altogether, the above code dispatches progress updates once per second during playback.

To initiate the Runnable task when the service is launched, add the following code to the bottom of the onCreate method:

```
playbackPositionChecker.run()
```

Moving on, we will now define a method called setCurrentMetadata that will process the metadata associated with the currently playing song. To define the setCurrentMetadata method, add the following code below the setMediaPlaybackState method:

```
private fun setCurrentMetadata() {  
    val metadata = MediaMetadataCompat.Builder().apply {  
        putString(  
            MediaMetadataCompat.METADATA_KEY_TITLE,  
            currentlyPlayingSong?.title  
        )  
        putString(  
            MediaMetadataCompat.METADATA_KEY_ARTIST,  
            currentlyPlayingSong?.artist  
        )  
        putString(  
            MediaMetadataCompat.METADATA_KEY_ALBUM,  
            currentlyPlayingSong?.album  
        )  
        putBitmap(  
            MediaMetadataCompat.METADATA_KEY_ALBUM_ART,  
            getArtwork(currentlyPlayingSong?.albumID) ?: BitmapFactory.decodeResource(application.resources,  
R.drawable.ic_launcher_foreground)  
        )  
    }.build()  
    mMediaSessionCompat.setMetadata(metadata)  
}
```

The above code defines the metadata for the currently playing song, which includes information on the song's title, artist, album and artwork. These details are all based on the Song object that was received by the onPrepareFromUri method. The album artwork will be located by a method called getArtwork using the song's album ID. If the artwork cannot be found, then the app launcher icon (the drawable resource called ic\_launcher\_foreground) will be used instead. That is why the Elvis operator ?: is used when assigning the value of the bitmap; if the left-hand side value of the Elvis operator is null (i.e. the getArtwork method returns a null value because it cannot find the artwork) then the value on the right-hand side of the Elvis operator will be used instead.

The getArtwork method can be defined by adding the following code below the setCurrentMetadata method:

```
private fun getArtwork(albumArtwork: String?) : Bitmap? {  
    try {  
        return BitmapFactory.Options().run {  
            inJustDecodeBounds = true  
            val cw = ContextWrapper(applicationContext)  
            val directory = cw.getDir("albumArt", Context.MODE_PRIVATE)  
            val f = File(directory, "$albumArtwork.jpg")  
            BitmapFactory.decodeStream(FileInputStream(f))  
  
            inSampleSize = calculateInSampleSize(this)  
            inJustDecodeBounds = false  
        }  
    }  
}
```

```

        BitmapFactory.decodeStream(FileInputStream(f))
    }
} catch (ignore: FileNotFoundException) { }
return null
}

```

The `getArtwork` method searches the `albumArt` directory in the app's internal files for an image associated with the album ID of the currently playing song. If a file is not found, then a `FileNotFoundException` exception will occur and the method will return a value of `null`. Meanwhile, if an image is found then a method called `calculateInSampleSize` will assess whether the image needs to be compressed. To define the `calculateInSampleSize` method, add the following code below the `getArtwork` method:

```

private fun calculateInSampleSize(options: BitmapFactory.Options): Int {
    val reqWidth = 100
    val reqHeight = 100
    val (height: Int, width: Int) = options.run { outHeight to outWidth }
    var inSampleSize = 1

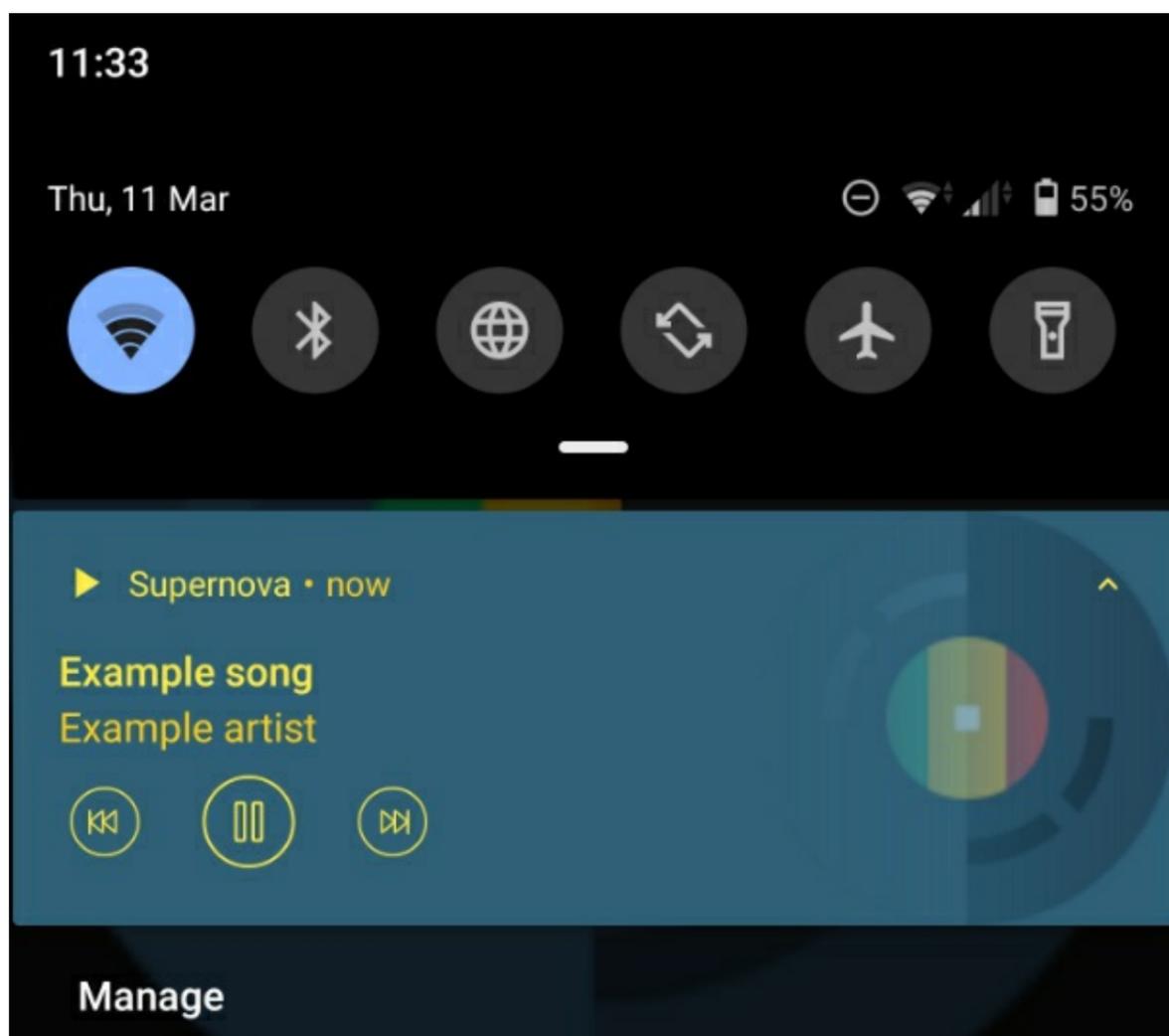
    if (height > reqHeight || width > reqWidth) {
        val halfHeight: Int = height / 2
        val halfWidth: Int = width / 2

        // Calculate the largest inSampleSize value that is a power of 2 and keeps both
        // height and width larger than the requested height and width.
        while (halfHeight / inSampleSize >= reqHeight && halfWidth / inSampleSize >= reqWidth) {
            inSampleSize *= 2
        }
    }

    return inSampleSize
}

```

The `calculateInSampleSize` method assesses whether the size of the image is larger than 100 pixels \* 100 pixels. If the image is larger, then the image size specifications are halved, which compresses the image and helps preserve the device's memory. The compressed image will be attached to the currently playing song's metadata and used in the notification that is displayed to the user. The media player notification will contain the details of the currently playing song (as specified in the metadata) and some action buttons to allow the user to pause, play and skip the current song.



To handle the notification, add the following code below the `calculateInSampleSize` method:

```
private fun showNotification(isPlaying: Boolean) {
    val playPauseIntent = if (isPlaying) Intent(applicationContext,
MediaPlayerService::class.java).setAction("ACTION_PAUSE")
    else Intent(applicationContext, MediaPlayerService::class.java).setAction("ACTION_PLAY")
    val nextIntent = Intent(applicationContext, MediaPlayerService::class.java).setAction("ACTION_NEXT")
    val prevIntent = Intent(applicationContext,
MediaPlayerService::class.java).setAction("ACTION_PREVIOUS")

    val intent = packageManager
        .getLaunchIntentForPackage(packageName)
        ?.setPackage(null)
        ?.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED)
    val activityIntent = PendingIntent.getActivity(applicationContext, 0, intent, PendingIntent.FLAG_IMMUTABLE)

    val builder = NotificationCompat.Builder(applicationContext, channelId).apply {
        // Get the session's metadata
        val controller = mMediaSessionCompat.controller
        val mediaMetadata = controller.metadata

        // previous button
        addAction(
            NotificationCompat.Action(
                R.drawable.ic_back,
                getString(R.string.play_prev),
                PendingIntent.getService(applicationContext, 0, prevIntent, PendingIntent.FLAG_IMMUTABLE)
            )
        )

        // play/pause button
        val playOrPause = if (isPlaying) R.drawable.ic_pause
        else R.drawable.ic_play

        addAction(
            NotificationCompat.Action(
                playOrPause,
                getString(R.string.play_pause),
                PendingIntent.getService(applicationContext, 0, playPauseIntent,
PendingIntent.FLAG_IMMUTABLE)
            )
        )

        // next button
        addAction(
            NotificationCompat.Action(
                R.drawable.ic_next,
                getString(R.string.play_next),
                PendingIntent.getService(applicationContext, 0, nextIntent, PendingIntent.FLAG_IMMUTABLE)
            )
        )

        setStyle(androidx.media.app.NotificationCompat.MediaStyle()
            .setShowActionsInCompactView(0, 1, 2)
            .setMediaSession(mMediaSessionCompat.sessionToken)
        )

        val smallIcon = if (isPlaying) R.drawable.play
        else R.drawable.pause
        setSmallIcon(smallIcon)

        setContentIntent(activityIntent)

        // Add the metadata for the currently playing track
        setTitle(mediaMetadata.getString(MediaMetadataCompat.METADATA_KEY_TITLE))
    }
}
```

```

setContentText(mediaMetadata.getString(MediaMetadataCompat.METADATA_KEY_ARTIST))
setLargeIcon(mediaMetadata.getBitmap(MediaMetadataCompat.METADATA_KEY_ALBUM_ART))

// Make the transport controls visible on the lockscreen
setVisibility(NotificationCompat.VISIBILITY_PUBLIC)
priority = NotificationCompat.PRIORITY_DEFAULT
}
// Display the notification and place the service in the foreground
startForeground(1, builder.build())
}

```

Note you may need to add the following import statement to the top of the file:

```
import androidx.core.app.NotificationCompat
```

The notification will feature playback control buttons that the user can interact with. If the user clicks a button, then an intent will be sent from the notification to the media browser service. For example, if the user presses the pause button then an intent will be dispatched with an action of “ACTION\_PAUSE”, which will direct the service to pause playback. Likewise, the skip forward, skip back and play buttons are associated with intent actions that the service can use to determine the appropriate response.

In addition to the buttons, the notification itself has an intent assigned to it, as defined in the `activityIntent` variable. When clicked, the notification should return the user to the application. To achieve this, the `getLaunchIntentForPackage` method finds the launcher activity for the application package. If you refer to the **AndroidManifest.xml** file (**Project > app > manifests**), you will see the `MainActivity` activity is assigned the `LAUNCHER` category. The intent is also assigned the flags `FLAG_ACTIVITY_NEW_TASK` and `FLAG_ACTIVITY_RESET_TASK_IF_NEEDED`, which means the intent will bring the `MainActivity` activity to the foreground above any other apps that may be running. In other words, if another application is in the foreground and the user clicks the notification, the Music app will become the foreground application again.

The notification is assembled using a `NotificationCompat` class builder. The builder will direct the notification to launch using the “music” channel ID. The channel ID helps the application monitor the notifications it is responsible for. The notification builder proceeds to define the skip back, play/pause and skip forward buttons and set the style of the notification to a `MediaStyle` object. The `MediaStyle` object is a readymade style package that automatically defines the notification layout and adjusts the notification colour scheme to contrast the album artwork bitmap that is loaded into the notification using the `setLargeIcon` method.

In addition to defining a large icon, which will occupy the main body of the notification, you can also specify a small icon. The small icon will appear in the notification bar and top left corner of the notification itself. For this app, the small icon will be either a play symbol or a pause symbol based on whether playback is in progress. The name and artist of the currently playing song will be loaded into the notification using the `setContentTitle` and `setContentText` commands, respectively. Also, the notification visibility is set to public to ensure the notification is always visible in the notification bar, even when the screen is locked. Finally, once all of the notification details have been set, the notification is launched using the `startForeground` method.

As discussed, the notification contains playback control buttons that fire an intent when clicked. The service can detect incoming intents via its `onStartCommand` method, so add the following code below the `onError` method to define how the service should handle the different intent actions:

```

override fun onStartCommand(intent: Intent, flags: Int, startId: Int): Int {
    intent.action?.let {
        when (it) {
            "ACTION_PLAY" -> mMediaSessionCallback.onPlay()
            "ACTION_PAUSE" -> mMediaSessionCallback.onPause()
            "ACTION_NEXT" -> mMediaSessionCallback.onSkipToNext()
            "ACTION_PREVIOUS" -> mMediaSessionCallback.onSkipToPrevious()
        }
    }
    return super.onStartCommand(intent, flags, startId)
}

```

The above code uses a `when` block to respond to the various intent actions. For example, if the notification’s play button is pressed then the action will be `ACTION_PLAY`. The `when` block will respond to this intent by running the service’s `onPlay` method to commence playback.

The last thing we need to do is define what happens when the service is destroyed. To do this, add the following code below the onStartCommand method:

```
override fun onDestroy() {
    super.onDestroy()
    handler.removeCallbacks(playbackPositionChecker)
    unregisterReceiver(mNoisyReceiver)
    mMediaSessionCallback.onStop()
    mMediaSessionCompat.release()
    NotificationManagerCompat.from(this).cancel(1)
}
```

The onDestroy method refers to a stage of the service lifecycle that occurs when the service is shut down. In this instance, we direct the onDestroy method to cancel any pending Runnable tasks that have been assigned to the Handler instance. Also, we close down the noisy broadcast receiver, media session and notification manager so they do not continue to consume computational resources.

## Interacting with the media browser service

The majority of the application's interactions with the media browser service will occur via the MainActivity class. For example, the MainActivity class will change the currently playing song, update the user interface based on the playback state and respond to user interactions with the playback controls. To begin, open the **MainActivity.kt** file (**Project > app > java > name of your project**) and add the following variables under the binding variable at the top of the class:

```
private val channelId = "music"
private var completeLibrary = listOf<Song>()
private var currentlyPlayingQueueID = 0
private var pbState = STATE_STOPPED
private val playbackViewModel: PlaybackViewModel by viewModels()
var playQueue = mutableListOf<Pair<Int, Song>>()
private lateinit var mediaBrowser: MediaBrowserCompat
private lateinit var sharedPreferences: SharedPreferences
```

Note you may need to add the following import statements to the top of the file:

```
import android.support.v4.media.session.PlaybackStateCompat.*
import androidx.activity.viewModels
```

The above variables provide the MainActivity class access to several features including the notification channel ID, which should match the channel ID you defined in MediaPlayerService class; the complete list of Song objects that comprise the user's music library; the queue ID of the currently playing song; the current playback state of the media browser service; the PlaybackViewModel view model that will store information relating to the playback of media including the play queue, current playback position, duration of the currently playing song etc.; the current play queue; an instance of the MediaBrowserCompat class that can interact with the media browser service; and an instance of the SharedPreferences class that can manage the user's preferences. Several of the variables use the lateinit modifier, which means they have not been initialised but should be treated as non-null.

When the MainActivity activity is launched, we should initialise the mediaBrowser and sharedPreferences variables. To do this, add the following code to the bottom of the onCreate method:

```
sharedPreferences = PreferenceManager.getDefaultSharedPreferences(this)
mediaBrowser = MediaBrowserCompat(
    this,
    ComponentName(this, MediaPlayerService::class.java),
    connectionCallbacks,
    intent.extras
)
mediaBrowser.connect()
```

Note you may need to add the following import statement to the top of the file:

```
import androidx.preference.PreferenceManager
```

The above code initialises the application's shared preferences manager and builds an instance of the

MediaBrowserCompat class. The MediaBrowserCompat instance will be associated with the media browser service we created earlier and a Callback object variable called connectionCallbacks. The Callback object will respond to the different media browser service connection states (connected, disconnected etc.). To define the Callback object, add the following code to the list of variables at the top of the class:

```
private val connectionCallbacks = object : MediaBrowserCompat.ConnectionCallback() {
    override fun onConnected() {
        super.onConnected()

        mediaBrowser.sessionToken.also { token ->

            val mediaControllerCompat = MediaControllerCompat(this@MainActivity, token)
            mediaControllerCompat.registerCallback(controllerCallback)
            MediaControllerCompat.setMediaController(this@MainActivity, mediaControllerCompat)
        }
    }
}

MediaControllerCompat.getMediaController(this@MainActivity)
    .registerCallback(controllerCallback)
}
```

The MediaBrowserCompat.ConnectionCallback object defined above contains a method called onConnected that will run when a connection with the media browser service is established. The onConnected method retrieves the media browser service's session token and uses it to register a MediaControllerCompat.Callback object. The Callback object will monitor the media session and respond to media button actions (play, pause, skip forward etc.) and changes in playback state and media metadata. To define the MediaControllerCompat.Callback object, add the following code below the connectionCallbacks variable:

```
private var controllerCallback = object : MediaControllerCompat.Callback() {
    override fun onPlaybackStateChanged(state: PlaybackStateCompat?) {
        super.onPlaybackStateChanged(state)
        val mediaController = MediaControllerCompat.getMediaController(this@MainActivity)
        if (state == null) return
        when (state.state) {
            STATE_PLAYING -> {
                pbState = state.state
                val playbackPosition = state.position.toInt()
                if (state.extras != null) {
                    val playbackDuration = state.extras!!.getInt("duration")
                    playbackViewModel.currentPlaybackDuration.value = playbackDuration
                }
                playbackViewModel.currentPlaybackPosition.value = playbackPosition
                playbackViewModel.isPlaying.value = true
            }
            STATE_PAUSED -> {
                pbState = state.state
                val playbackPosition = state.position.toInt()
                if (state.extras != null) {
                    val playbackDuration = state.extras!!.getInt("duration")
                    playbackViewModel.currentPlaybackDuration.value = playbackDuration
                }
                playbackViewModel.currentPlaybackPosition.value = playbackPosition
                playbackViewModel.isPlaying.value = false
            }
            STATE_STOPPED -> {
                pbState = state.state
                playbackViewModel.isPlaying.value = false
                playbackViewModel.currentPlayQueue.value = mutableListOf()
                playbackViewModel.currentlyPlayingQueueID.value = 0
                playbackViewModel.currentlyPlayingSong.value = null
                playbackViewModel.currentPlaybackDuration.value = 0
                playbackViewModel.currentPlaybackPosition.value = 0
            }
        }
    }
}
```

```

STATE_SKIPPING_TO_NEXT -> {
    val repeatSetting = sharedPreferences.getInt("repeat", REPEAT_MODE_NONE)
    when {
        repeatSetting == REPEAT_MODE_ONE -> {}
        playQueue.isNotEmpty() && playQueue[playQueue.size - 1].first != currentlyPlayingQueueID -> {
            val index = playQueue.indexOfFirst {
                it.first == currentlyPlayingQueueID
            }
            currentlyPlayingQueueID = playQueue[index + 1].first
        }
        // We have reached the end of the queue. check whether we should start over from the beginning
        repeatSetting == REPEAT_MODE_ALL -> currentlyPlayingQueueID = playQueue[0].first
        else -> {
            mediaController.transportControls.stop()
            return
        }
    }
}

```

```

lifecycleScope.launch {
    updateCurrentlyPlaying()
    if (pbState == STATE_PLAYING) play()
}
}

```

```

STATE_SKIPPING_TO_PREVIOUS -> {
    if (playQueue.isNotEmpty() && currentlyPlayingQueueID != playQueue[0].first) {
        val index = playQueue.indexOfFirst {
            it.first == currentlyPlayingQueueID
        }
        currentlyPlayingQueueID = playQueue[index - 1].first
        lifecycleScope.launch {
            updateCurrentlyPlaying()
            if (pbState == STATE_PLAYING) play()
        }
    }
}
else -> return
}
}
}

```

Note you may need to add the following import statements to the top of the file:

```

import androidx.lifecycle.LifecycleScope
import kotlinx.coroutines.*

```

The `MediaControllerCompat.Callback` object will monitor the playback state (delivered in a `PlaybackStateCompat` object) of the media browser service. Each time the playback state changes, the `when` block will determine the appropriate response:

- `STATE_PLAYING` or `STATE_PAUSED` - the current playback position and song duration are extracted from the `PlaybackStateCompat` object and uploaded to the `PlaybackViewModel` view model. Also, the view model's `isPlaying` variable will be set to true or false based on whether playback is in progress.
- `STATE_STOPPED` - the play queue, currently loaded song and playback position details are all cleared from the `MainActivity` and `PlaybackViewModel` classes because playback has ended.
- `STATE_SKIPPING_TO_NEXT` - this playback state will occur whenever the user clicks the skip forward button or the currently playing song finishes. In which case, the code retrieves the user's repeat mode preference via the `SharedPreferences` class. If the repeat mode setting is set to `REPEAT_MODE_ONE`, then the media browser service should continue repeating the current song, so no further adjustments are required. For the other repeat mode settings, the code checks whether there are further songs in the play queue. If there are more songs to play, then the `currentlyPlayingQueueID` is updated to reflect the ID of the next item in the play queue. If we have reached the end of the play queue, then the code checks whether the repeat mode is set to `REPEAT_MODE_ALL`. If so, then the `currentlyPlayingQueueID` is set to the ID

of the first song in the play queue. Otherwise, all songs have been played and the user has not instructed the app to repeat the play queue. In which case, the media browser service's `onStop` method is called to end playback and set the playback state to `STATE_STOPPED`.

- `STATE_SKIPPING_TO_PREVIOUS` - this playback state will run when the user clicks the skip backward button on the playback controls. In which case, the code loads the previous song in the play queue or reloads the current song if there are no prior songs.

Once the `STATE_SKIPPING_TO_NEXT` and `STATE_SKIPPING_TO_PREVIOUS` blocks have determined the next song to play, a method called `updateCurrentlyPlaying` (which we'll define later) will load the song into the media browser service. If playback was in progress when the playback state was changed, then a method called `play` will initiate playback of the new song. Otherwise, the new song will load but not play until directed to do so by the user. These actions are managed by a `LifecycleScope` coroutine scope, which means they will run behind the scenes and not block other app processes. Actions launched using `LifecycleScope` are cancelled when the lifecycle owner (the `MainActivity` activity in this case) is destroyed.

Moving on, you may recall the media browser service will display a notification containing the details of the currently playing song. This notification will be broadcast through a notification channel with a channel ID of "music". The `MainActivity` activity needs to initialise the notification channel before it can be used by the media browser service. To handle this, add the following code below the `onSupportNavigateUp` method in the `MainActivity` class:

```
private fun createChannel() {
    val channel = NotificationChannel(
        channelId, "Notifications",
        NotificationManager.IMPORTANCE_DEFAULT
    ).apply {
        description = "All app notifications"
        setSound(null, null)
        setShowBadge(false)
    }
    val notificationManager = getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
    notificationManager.createNotificationChannel(channel)
}
```

The `createChannel` method creates a channel through which notifications can be broadcast. It also defines the characteristics of the notification. For example, the `setSound` method is run using `null` values for the sound and `audioAttributes` arguments, which prevents a sound from playing when the notification appears. Otherwise, there would be an alert noise each time the notification is changed. To create the notification channel when the app is launched, add the following line of code to the bottom of the `onCreate` method:

```
createChannel()
```

Finally, we need to disconnect the media browser service when the app is closed. For this purpose, we will instruct the `MainActivity` activity to close the service when it enters the `onDestroy` stage of the activity lifecycle. To arrange this, add the following code below the `onSupportNavigateUp` method:

```
override fun onDestroy() {
    super.onDestroy()

    MediaControllerCompat.getMediaController(this)?.apply {
        transportControls.stop()
        unregisterCallback(controllerCallback)
    }
    mediaBrowser.disconnect()
}
```

The `MainActivity` class is now equipped to connect and disconnect from the media browser service, respond to changes in the playback state, and create a notification channel for playback notifications to be broadcast.

## Initiating playback

In this section, we'll configure the `MainActivity` class to support media playback and manage the play queue. The first method we'll discuss is called `playNewSongs` and will prepare new play queues. To define the `playNewSongs` method, add the following code below the `createChannel` method:

```

fun playNewSongs(playlist: List<Song>, startSong: Int?, shuffle: Boolean) = lifecycleScope.launch(
    Dispatchers.Main) {
    if (playlist.isNotEmpty()) {
        playQueue = mutableListOf()
        for ((i, s) in playlist.withIndex()) {
            val queueItem = Pair(i, s)
            playQueue.add(queueItem)
        }
        if (shuffle) playQueue.shuffle()

        currentlyPlayingQueueID = if (shuffle) playQueue[0].first
        else startSong ?: 0

        sharedPreferences.edit()
            .putBoolean("shuffle", shuffle)
            .apply()

        updateCurrentlyPlaying()
        play()
    }
}

```

The `playNewSongs` method features arguments called `playlist`, which contains the list of `Song` objects that will form the play queue; `startSong`, which contains the queue item index playback should begin or null if playback should start from the beginning of the play queue; and `shuffle`, which contains a boolean (true/false value) indicating whether the play queue should be shuffled. The `playNewSongs` method iterates through the `Song` objects used in the `playlist` argument and constructs a play queue. The play queue comprises `Pair` objects consisting of the original index of the queue item (starting at 0 and increasing incrementally) and the corresponding `Song` object. If the user has indicated that the play queue should be shuffled, then the list of `Pair` objects is shuffled and the user's shuffle preference is saved using the `SharedPreferences` class. Finally, methods called `updateCurrentlyPlaying` and `play` load the first song in the play queue to the media browser service and initiate playback, respectively.

To define the `updateCurrentlyPlaying` method, add the following code below the `playNewSongs` method:

```

private suspend fun updateCurrentlyPlaying() {
    val index = playQueue.indexOfFirst {
        it.first == currentlyPlayingQueueID
    }

    val currentQueueItem = if (playQueue.isNotEmpty() && index != -1) playQueue[index]
    else return

    withContext(Dispatchers.IO) {
        playbackViewModel.currentlyPlayingSong.postValue(currentQueueItem.second)
        playbackViewModel.currentPlayQueue.postValue(playQueue)
        playbackViewModel.currentlyPlayingQueueID.postValue(currentlyPlayingQueueID)
        val bundle = Bundle()
        // convert the Song object for each song to JSON and store it in a bundle
        val gPretty = GsonBuilder().setPrettyPrinting().create().toJson(currentQueueItem.second)
        bundle.putString("song", gPretty)
        mediaController.transportControls.prepareFromUri(Uri.parse(currentQueueItem.second.uri), bundle)
    }
}

```

The `updateCurrentlyPlaying` method retrieves the `Pair` object that is associated with the queue index stored in the `currentlyPlayingQueueID` variable. The `Song` object associated with the queue item is then uploaded to the `PlaybackViewModel` view model along with the play queue and the index of the currently playing queue item. Finally, the currently playing `Song` object is converted to a JSON string using `GSON`, packaged into a bundle and sent to the media browser service using the `MediaController` class's `prepareFromUri` method.

Moving on, to define the `play` method, add the following code below the `updateCurrentlyPlaying` method:

```

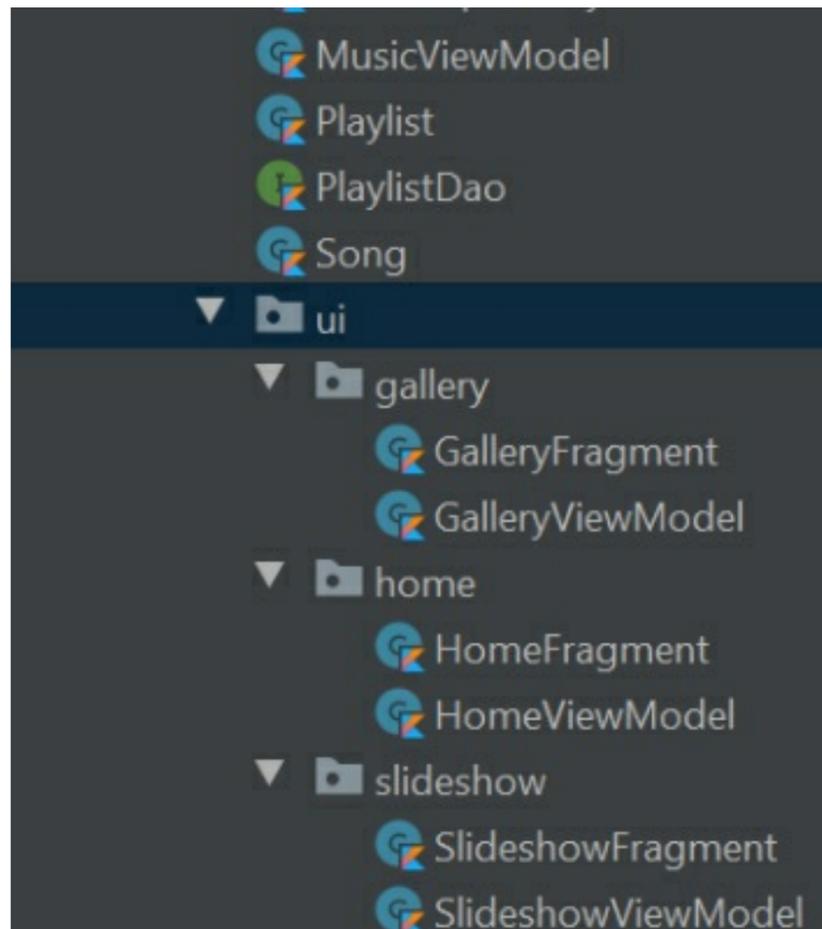
private fun play() = mediaController.transportControls.play()

```

The `updateCurrentlyPlaying` and `play` methods are defined separately because sometimes it is sufficient to prepare a song without initiating playback. For example, the user could be skipping through tracks while playback is paused. The `play` method is quite simple. It uses the `MediaController` class to run the media browser service's `onPlay` method and initiate playback of the song that was loaded into the media player by the `onPrepareFromUri` method.

## Preparing the fragment packages

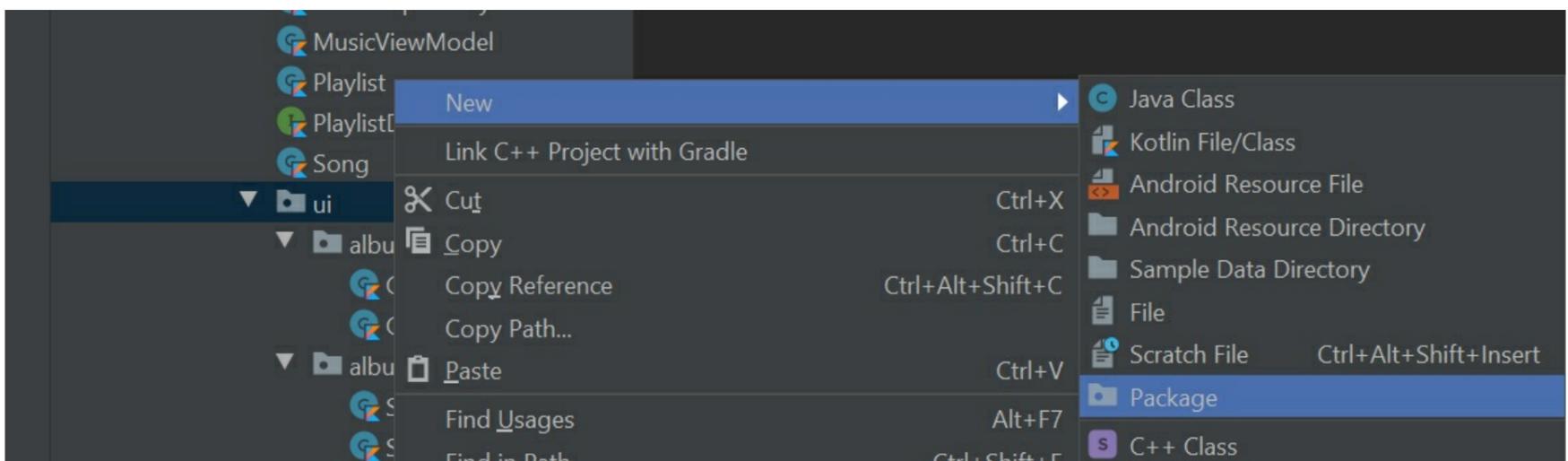
The Music app will contain multiple fragments. A fragment is a destination within an activity that has a distinct lifecycle and user interface but can communicate information to other fragments and the parent activity when needed. When the project was created using the Navigation Drawer Activity template, Android Studio will likely have generated readymade fragment packages called `Gallery`, `Home` and `Slideshow`, as shown below.



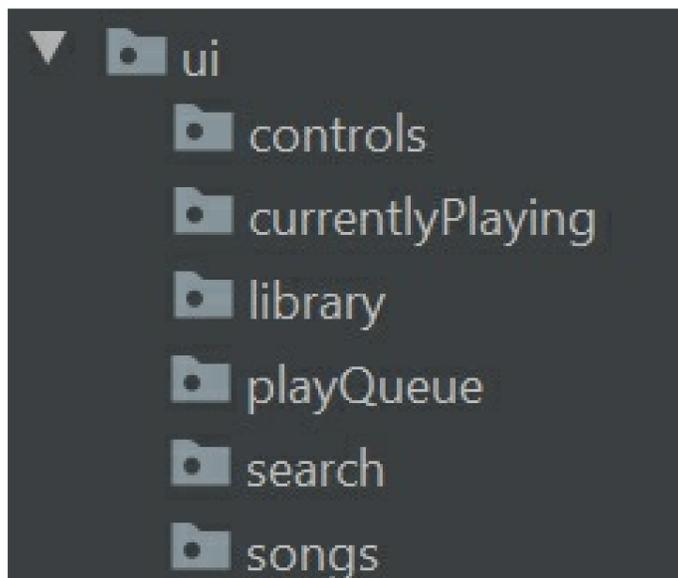
The music app will require six packages in total:

- `controls`
- `currentlyPlaying`
- `library`
- `playQueue`
- `search`
- `songs`

To prepare these packages, it is best to delete the `Gallery`, `Home` and `Slideshow` packages and create six new packages from scratch. To create a package, right-click the `ui` directory then select **New** > **Package** and enter a name from the above list.

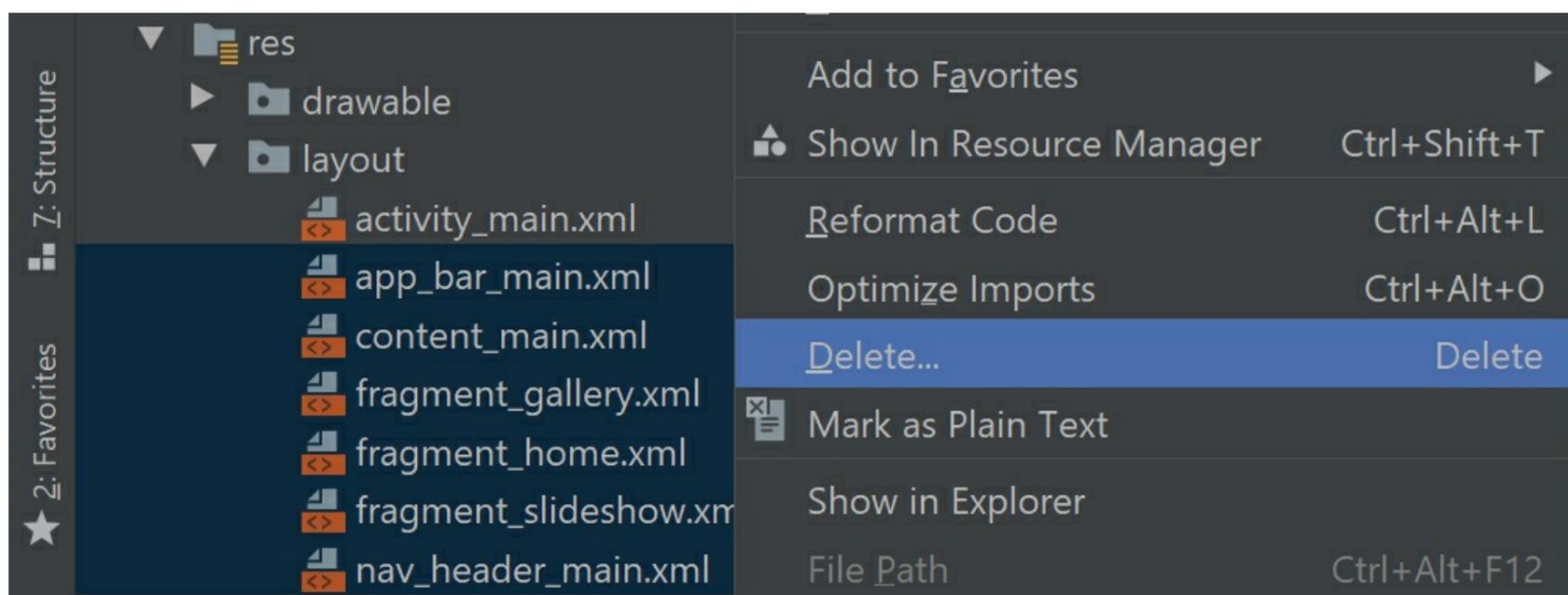


Once all required packages have been created the `ui` directory should look like this:



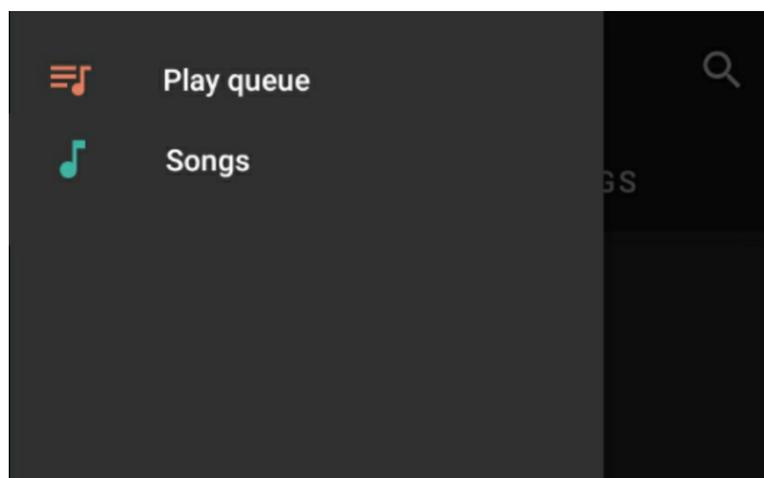
## Designing the activity\_main layout

User interfaces are defined using layout resource files. When the project was created using the Navigation Drawer Activity template, Android Studio will likely have generated several layout resource files, most of which we will not use. To delete the superfluous layout files, navigate through **Project > app > res > layout** and delete every file except **activity\_main.xml**.



Next, open the **activity\_main.xml** layout file in Code view. This is the layout that will load when the app is launched, as directed by the setContentView statement in the MainActivity class's onCreate method. The activity\_main layout will coordinate the fragments and other components such as the toolbar and audio playback controls.

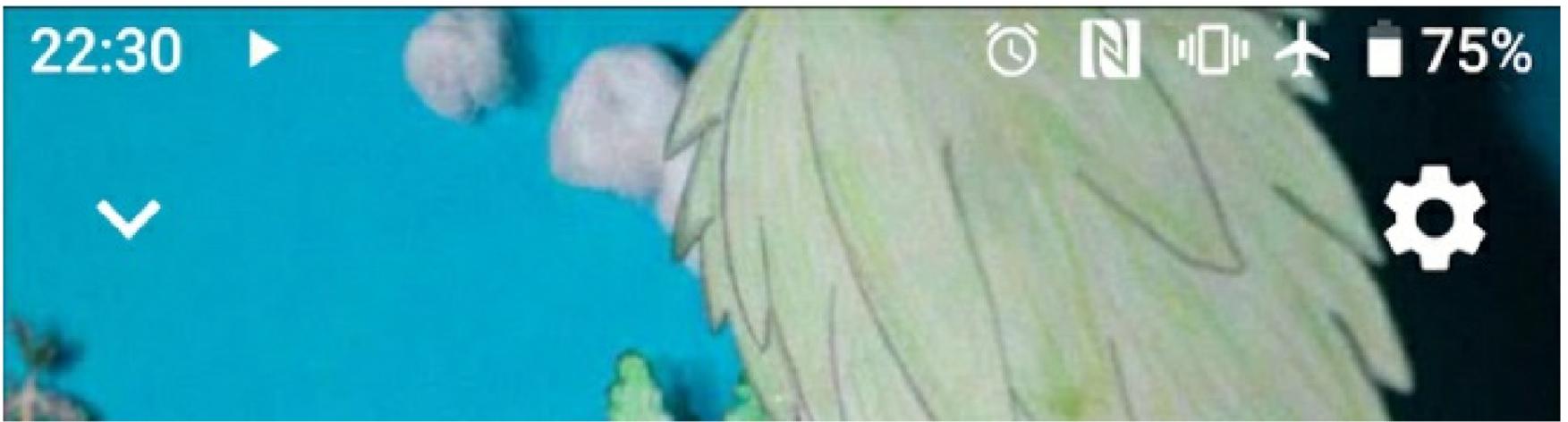
The root element of the activity\_main layout is a DrawerLayout widget. DrawerLayout widgets act as a top-level container and feature a side panel that can be pulled from the left or right side of the window. The direction that the panel can be pulled is determined using the DrawerLayout's openDrawer property (start = left; end = right). In the Music app, we will draw a side panel from the left-hand side and the panel will contain items for the navigational destinations in the app.



In the activity\_main layout, locate the DrawerLayout element and delete the following line of code:

```
android:fitsSystemWindows="true"
```

When set to true, the `fitsSystemWindows` command prevents the layout from stretching below the device's notification bar. This restriction will not be necessary for this app because when the user views the currently playing song, the artwork of that song should be visible under the notification bar as shown below:



Next, locate the include element and replace it with the following code:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <com.google.android.material.appbar.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="@android:color/transparent"
        android:fitsSystemWindows="true"
        app:elevation="0dp">

        <androidx.appcompat.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize" />
        </com.google.android.material.appbar.AppBarLayout>

        <androidx.fragment.app.FragmentContainerView
            android:id="@+id/nav_host_fragment"
            android:name="androidx.navigation.fragment.NavHostFragment"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_marginBottom="84dp"
            app:defaultNavHost="true"
            app:navGraph="@navigation/mobile_navigation"
            app:layout_behavior="@string/appbar_scrolling_view_behavior" />

        <androidx.fragment.app.FragmentContainerView
            android:id="@+id/nav_controls_fragment"
            android:name="androidx.navigation.fragment.NavHostFragment"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            app:navGraph="@navigation/controls_navigation" />
    </androidx.coordinatorlayout.widget.CoordinatorLayout>
```

In the above code, we define a `CoordinatorLayout` widget that will contain the app toolbar and two `FragmentContainerView` widgets. The `FragmentContainerViews` will house fragments that display the user's current destination in the app and the playback controls, respectively. The toolbar will contain menu items to facilitate user interactions with the app and navigation drawer. The `AppBarLayout` widget which contains the toolbar has an elevation of `0dp`, transparent background and the `fitsSystemWindows` attribute set to true. Altogether, these attributes ensure there is no shadow under the toolbar, the toolbar is the same colour as the underlying fragment and the toolbar will leave enough space for the notification bar at the top of the screen. The `FragmentContainerView` widgets both contain a reference to a `navGraph` (navigation graph), which defines the network of fragments that the `FragmentContainerView` can host. The `FragmentContainerView` widgets can use the navigation graphs to facilitate user navigation. We'll discuss the navigation graphs in more detail later.

The last change to make in the **activity\_main.xml** file is to locate the **NavigationView** widget and remove the **headerLayout** attribute because the navigation drawer will not use a header:

```
app:headerLayout="@layout/nav_header_main"
```

## The main navigation graph

In the previous section, we added a **FragmentManager** widget to the **activity\_main** layout with the ID **nav\_host\_fragment**. The **FragmentManager** references a navigation graph called **mobile\_navigation.xml**, which will define the network of the core destinations in the app. To define the graph, navigate through **Project > app > res > navigation**, open the file called **mobile\_navigation.xml** in Code view and edit the file so it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/mobile_navigation"
    app:startDestination="@+id/nav_library">

    <fragment
        android:id="@+id/nav_queue"
        android:name="com.example.music.ui.playQueue.PlayQueueFragment"
        android:label="@string/play_queue"
        tools:layout="@layout/fragment_play_queue" />

    <fragment
        android:id="@+id/nav_library"
        android:name="com.example.music.ui.library.LibraryFragment"
        android:label="@string/library"
        tools:layout="@layout/fragment_library" >
        <argument
            android:name="position"
            android:defaultValue="1"
            app:argType="integer" />
    </fragment>

    <fragment
        android:id="@+id/nav_search"
        android:name="com.example.music.ui.search.SearchFragment"
        android:label="@string/search"
        tools:layout="@layout/fragment_search" />

    <fragment
        android:id="@+id/nav_songs"
        android:name="com.example.music.ui.songs.SongsFragment"
        android:label="@string/songs"
        tools:layout="@layout/fragment_songs" />

    <fragment
        android:id="@+id/nav_edit_song"
        android:name="com.example.music.ui.songs.EditSongFragment"
        android:label="@string/edit_metadata"
        tools:layout="@layout/fragment_edit_song">
        <argument
            android:name="song"
            android:defaultValue="@null"
            app:nullable="true"
            app:argType="com.example.music.Song" />
    </fragment>

    <action
        android:id="@+id/action_edit_song"
        app:destination="@id/nav_edit_song" />
```

```

<action
  android:id="@+id/action_library"
  app:destination="@id/nav_library" />
</navigation>

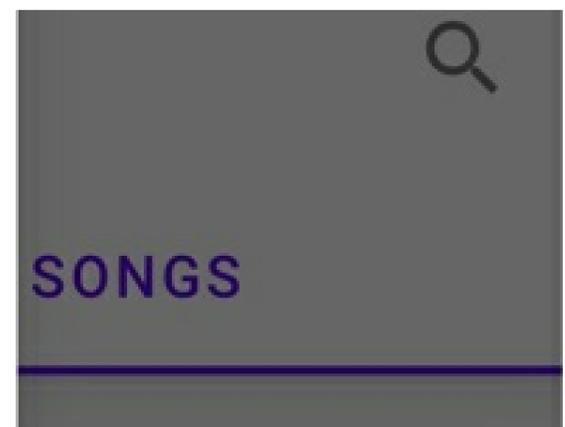
```

The root element of the above navigation graph contains an attribute called `startDestination`. The `startDestination` attribute defines the origin of the navigation network. In this instance, the `startDestination` is set to a fragment called `nav_library`, which will contain an overview of the user's music library and the tabs that allow the user to switch between the play queue and songs fragments. Next, the details of each fragment destination are defined. Each fragment is assigned an ID that will identify that destination in the navigation graph. The name of the fragment is its position within the project's directory structure. Note you may need to replace the "example.music" part of each fragment name to reflect the structure of your project. Each fragment also contains a label. The label will be displayed in the toolbar when the fragment is open. Finally, each fragment contains a layout attribute that defines the layout that will display the fragment's content.

You may notice some fragment entries also contain an argument. The argument describes a piece of data that is sent to the fragment when it is loaded. Argument data types can include strings, integers and even custom objects such as the `Song` data class. Arguments allow us to transmit information from one destination to another. For example, when the user navigates to the `EditSongFragment` it is necessary to send the `Song` object that the user wishes to modify the details of. At the end of the navigation graph, a couple of global actions are defined. Each action contains a destination attribute that specifies the fragment that the action will load. The actions are considered global because they are not tied to any particular fragment and can be launched from anywhere within the navigation network.

## The navigation drawer

To help the user navigate around the app, we will create a navigation drawer, which is a panel that slides out from the left-hand side of the screen.



The items in the navigation drawer are defined in a menu resource file called `activity_main_drawer.xml`. Locate the file by navigating through **Project** > **app** > **res** > **menu**, open it and replace the items inside the group element with the following code:

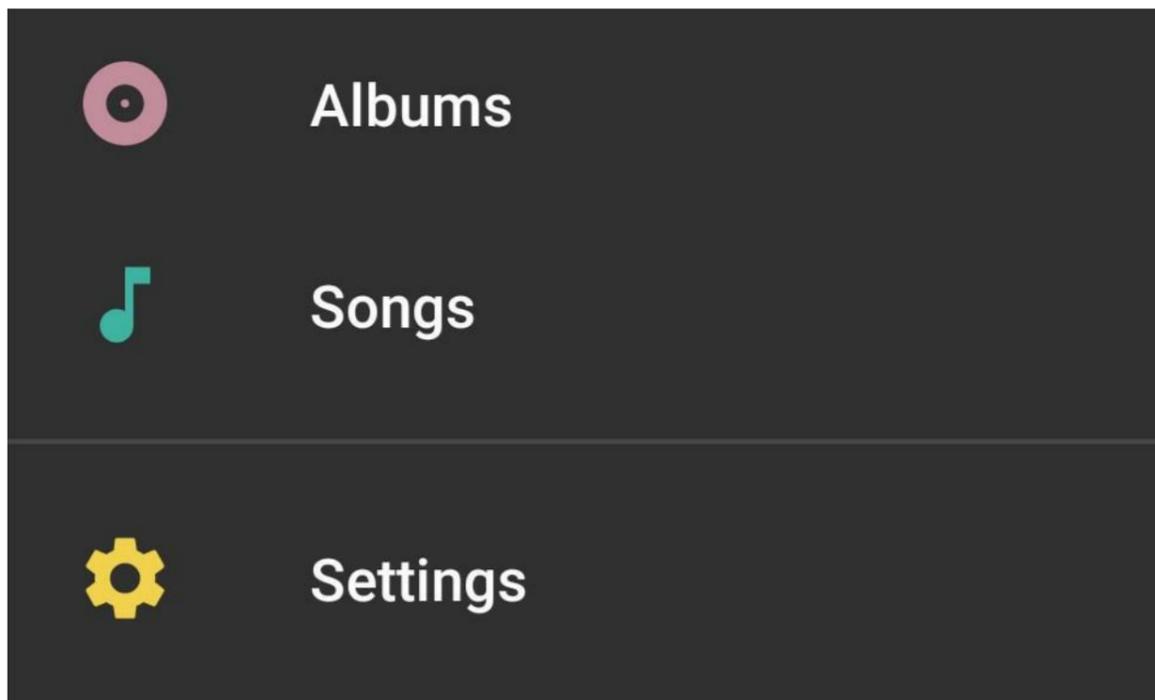
```

<item
  android:id="@+id/nav_queue"
  android:icon="@drawable/ic_nav_playing"
  android:title="@string/play_queue" />
<item
  android:id="@+id/nav_songs"
  android:icon="@drawable/ic_nav_songs"
  android:title="@string/songs" />

```

The above code defines two menu items that will appear in the navigation drawer. The menu items will allow the user to navigate to the play queue and songs fragments, although you could add further destinations if you wish (e.g. Albums, Artists, Playlists etc.). Each item features an icon and title, which will be displayed to the user.

Sometimes you may like to add a dividing line between navigation drawer items as shown below.



To achieve this, you must define two separate menu groups for items that should appear above and below the dividing line, respectively. Assign the first group an `orderInCategory` value of 1 and the second group an `orderInCategory` value of 2 e.g.

```
android:orderInCategory="1"
```

You should now see a dividing line between group 1 and group 2.

To make the navigation drawer operational, return to the `MainActivity` class and locate the part of the `onCreate` method that initialises the `AppBarConfiguration` variable:

```
AppBarConfiguration = AppBarConfiguration(setOf(  
    R.id.nav_home, R.id.nav_gallery, R.id.nav_slideshow), drawerLayout)
```

The fragments listed in the `AppBarConfiguration` instance will be treated as top-level destinations. Top-level destinations are considered the origins of navigational pathways. Also, top-level destinations influence the composition of the app toolbar. For example, when a top-level destination is open there will be a hamburger icon in the top-left corner of the toolbar. Meanwhile, regular destinations will have a back arrow instead. The hamburger icon will open the navigation drawer, while the back arrow will return the user to their previous destination.



To define the top-level destinations for this app, edit the `AppBarConfiguration` variable initialisation code so it reads as follows:

```
AppBarConfiguration = AppBarConfiguration(setOf(R.id.nav_queue, R.id.nav_library, R.id.nav_songs),  
drawerLayout)
```

Moving on, let's configure the `MainActivity` class to support the navigation drawer. First, edit `onCreate` method's `setSupportActionBar` call so it reads as follows:

```
setSupportActionBar(binding.toolbar)
```

Also, delete the following lines of code from the `onCreate` method because they do not apply to the Music app:

```
binding.appBarMain.fab.setOnClickListener { view ->  
    Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)  
        .setAction("Action", null).show()  
}
```

Next, locate the line of code that defines a variable called `navController` and replace it with the following:

```
val navHostFragment = supportFragmentManager.findFragmentById(R.id.nav_host_fragment) as NavHostFragment
val navController = navHostFragment.navController
```

The above code initialises a `NavHostFragment` object, which will provide access to the first `FragmentManager` widget from the `activity_main.xml` layout. The `NavController` object associated with the `FragmentManager` is then assigned to a variable called `navController`, so the `MainActivity` class can interact with the network defined in the `mobile_navigation.xml` navigation graph.

Next, add the following code to the bottom of the `onCreate` method:

```
val mOnNavigationItemSelectedListener = NavigationView.OnNavigationItemSelectedListener { item ->
    when (item.itemId) {
        R.id.nav_queue -> {
            val action = MobileNavigationDirections.actionLibrary(0)
            navController.navigate(action)
        }
        R.id.nav_songs -> {
            val action = MobileNavigationDirections.actionLibrary(1)
            navController.navigate(action)
        }
    }
    binding.drawerLayout.closeDrawer(GravityCompat.START)
    true
}
binding.navView.setNavigationItemSelectedListener(mOnNavigationItemSelectedListener)
```

The above code uses a `when` block to define the appropriate response when a given navigation drawer item is selected. In this instance, both the play queue and songs items trigger the `actionLibrary` navigation action from the `mobile_navigation.xml` navigation graph; however, each item passes a different value to the `nav_library` destination's position argument. This is because the library fragment is split into two sections and the value of the position argument determines whether the library fragment will load the play queue (0) or list of songs in the user's music library (1). The navigation drawer must manually be closed once an item has been selected, which is why the `closeDrawer` method is run after the `when` block.

Finally, add the following line of code to the bottom of the `onCreate` method

```
binding.navView.itemIconTintList = null
```

The above code sets the `NavigationView`'s `itemIconTintList` property to `null`, which prevents Android from overriding the colour of the navigation drawer's items. Otherwise, all the icons would appear grey (or a different colour depending on the theme).

## Setting up the Library fragment

The first fragment that loads when the app is launched will be called `LibraryFragment`. The `LibraryFragment` fragment will help coordinate the play queue and songs fragments and allow the user to swipe between them using tabs and the `ViewPager2` library. `ViewPager2` is a tool that allows you to display content pages that the user may swipe through, much like pages in a book.

First, let's create a layout resource file for the `LibraryFragment` fragment. Right-click the **layout** directory (**Project** > **app** > **res**) then select **New** > **Layout Resource File**. Name the file `fragment_library` and edit the layout's code so it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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" >

    <com.google.android.material.tabs.TabLayout
        android:id="@+id/tabLayout"
        android:layout_width="match_parent"
```

```
android:layout_height="wrap_content"  
app:layout_constraintTop_toTopOf="parent" />
```

```
<androidx.viewpager2.widget.ViewPager2  
    android:id="@+id/viewPager"  
    android:layout_width="match_parent"  
    android:layout_height="0dp"  
    app:layout_constraintTop_toBottomOf="@id/tabLayout"  
    app:layout_constraintBottom_toBottomOf="parent"/>  
</androidx.constraintlayout.widget.ConstraintLayout>
```

The `fragment_library` layout comprises a `TabLayout` widget, which will contain the names of the different sections (Play Queue and Songs), and a `ViewPager2` widget, which will display the content. The user can navigate between the different sections by pressing one of the `TabLayout` tabs or by swiping the `ViewPager2` widget.

Next, let's create the `LibraryFragment` class. Locate and right-click the **library** directory (**Project** > **app** > **java** > **name of the project** > **ui**) then select **New** > **Kotlin Class/File**. Name the file **LibraryFragment** and select **Class** from the list of options. Once the **LibraryFragment.kt** file opens in the editor, edit its code to the following:

```
import androidx.fragment.app.Fragment  
  
class LibraryFragment : Fragment() {  
    private var _binding: FragmentLibraryBinding? = null  
    private val binding get() = _binding!!  
    private var viewPagerPosition: Int? = null  
  
    override fun onCreateView(  
        inflater: LayoutInflater,  
        container: ViewGroup?,  
        savedInstanceState: Bundle?  
    ): View {  
        arguments?.let {  
            val safeArgs = LibraryFragmentArgs.fromBundle(it)  
            viewPagerPosition = safeArgs.position  
        }  
  
        _binding = FragmentLibraryBinding.inflate(inflater, container, false)  
  
        val viewPagerAdapter = ViewPagerAdapter(this)  
        binding.viewPager.adapter = viewPagerAdapter  
        binding.viewPager.currentItem = viewPagerPosition ?: 0  
  
        val navView: NavigationView = requireActivity().findViewById(R.id.nav_view)  
        val pageChangeCallback = object : ViewPager2.OnPageChangeCallback() {  
            override fun onPageSelected(position: Int) {  
                when (position) {  
                    0 -> navView.setCheckedItem(R.id.nav_queue)  
                    1 -> navView.setCheckedItem(R.id.nav_songs)  
                }  
            }  
        }  
        binding.viewPager.registerOnPageChangeCallback(pageChangeCallback)  
  
        val namesArray = arrayOf(getString(R.string.play_queue), getString(R.string.songs))  
        TabLayoutMediator(binding.tabLayout, binding.viewPager) { tab, position ->  
            tab.text = namesArray[position]  
        }.attach()  
        binding.tabLayout.tabGravity = TabLayout.GRAVITY_FILL  
  
        return binding.root  
    }  
  
    override fun onDestroyView() {  
        super.onDestroyView()  
        _binding = null  
    }  
}
```

```
}  
}
```

In the above code, the LibraryFragment class's onCreateView method retrieves the integer that was supplied for the destination's position argument, as defined in the **mobile\_navigation.xml** navigation graph. The integer will equal either 0 or 1, with 0 representing the play queue fragment and 1 representing the songs fragment. By default, the value will be 1, as specified in the argument section of the nav\_library item's entry in the mobile\_navigation navigation graph. Hence, the LibraryFragment will load the songs fragment unless advised otherwise.

The onCreate method proceeds to configure the ViewPager2 widget and set its current position to the value stored in the viewPagerPosition variable. Also, an OnPageChangeCallback object is assigned to the ViewPager2. The object contains a callback method called onPageSelected that monitors the active ViewPager2 page position and updates the active item in the navigation drawer accordingly. From the previous section, you may recall that the active navigation drawer item will have an overlay effect for emphasis, so it is important to ensure the active item is always correct. The remainder of the code initialises the TabLayout widget and populates two tabs called Play Queue and Songs. Also, the TabLayout widget's gravity property is set to GRAVITY\_FILL to ensure the TabLayout occupies the full width of the window.

To make the ViewPager2 and TabLayout widgets operational, we must create an adapter that will load the appropriate fragment when the user swipes through the ViewPager2 or clicks a tab in the TabLayout. Create a new Kotlin class in the **library** directory called ViewPagerAdapter and add the following code to the file:

```
import androidx.fragment.app.Fragment  
import com.example.music.ui.playQueue.PlayQueueFragment  
import com.example.music.ui.songs.SongsFragment  
  
class ViewPagerAdapter(fragment: Fragment) : FragmentStateAdapter(fragment) {  
  
    override fun getItemCount(): Int = 2  
  
    override fun createFragment(position: Int): Fragment {  
        return if (position == 0) PlayQueueFragment()  
        else SongsFragment()  
    }  
}
```

The above code is quite simple. First, the number of items loaded into the adapter (as defined in the getItemCount method) is set to 2 because there will only be two fragments available (PlayQueueFragment and SongsFragment). Next, the createFragment method instructs the view pager to return the PlayQueueFragment when the user is at the first position (index 0) and SongsFragment if the user is at any other position (i.e. index 1). As the user swipes through the view pager from start to finish, they will be taken from the PlayQueue fragment to the Songs fragment.

## Setting up the Songs fragment and layout

We'll now design the SongsFragment fragment, which will contain a list of every song in the user's music library. This fragment will require a layout. To create a new layout file, right-click the **layout** directory (**Project** > **app** > **res**) then select **New** > **Layout Resource File**. Name the file fragment\_songs then press OK. Once the **fragment\_songs.xml** layout opens in the editor, switch to Code view and modify the file so it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>  
<androidx.constraintlayout.widget.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" >  
  
    <com.simplecityapps.recyclerview_fastscroll.views.FastScrollRecyclerView  
        android:id="@+id/recyclerView"  
        android:layout_width="match_parent"  
        android:layout_height="match_parent"  
        app:fastScrollAutoHide="true"  
        app:fastScrollAutoHideDelay="1500"  
        app:fastScrollTrackColor="@android:color/transparent"  
        app:fastScrollThumbInactiveColor="@color/design_default_color_primary"  
        app:fastScrollEnableThumbInactiveColor="true"  
        app:fastScrollPopupBackgroundSize="62dp"
```

```

app:fastScrollPopupBgColor="@color/design_default_color_primary"
app:fastScrollPopupPosition="adjacent"
app:fastScrollPopupTextColor="@android:color/white"
app:fastScrollPopupTextSize="32dp"
app:fastScrollPopupTextVerticalAlignmentMode="font_metrics"
app:fastScrollThumbColor="@color/design_default_color_primary"
app:fastScrollThumbEnabled="true" />

```

```

<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/fab"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_margin="16dp"
    android:contentDescription="@string/shuffle_tracks"
    app:srcCompat="@drawable/ic_shuffle"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>

```

The entirety of the `fragment_songs` layout is occupied by a modified RecyclerView widget from timusus's Github (<https://github.com/timusus/RecyclerView-FastScroll>). The custom RecyclerView widget is fast scroll-ready. The fast scroll feature will allow the user to quickly browse the list of songs, which is important given how the user could potentially have thousands of songs in their music library. The fast scroll RecyclerView has a customisable thumb that shows the user's scroll position in the overall list of data.

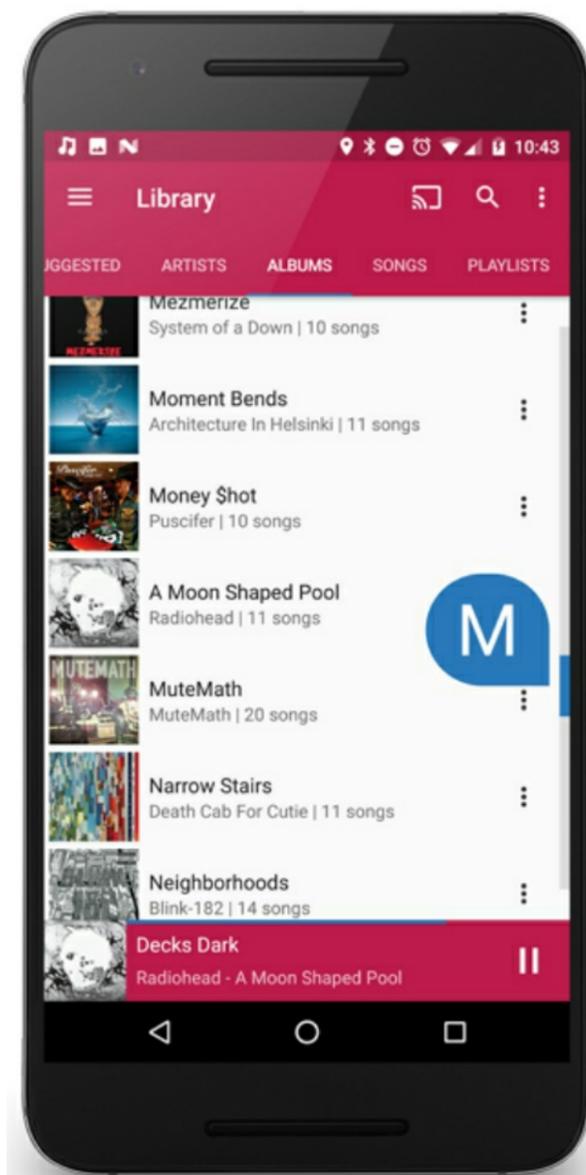


Image source: <https://github.com/timusus/RecyclerView-FastScroll>

Also, the `fragment_songs` layout contains a floating action button that is constrained to the bottom right corner of the layout. The button uses the `ic_shuffle` drawable resource as its icon image because the button will play all the songs in the user's music library in shuffle mode when clicked.

To make the Songs fragment operational, create a new Kotlin class by right-clicking the **songs** directory (**Project** > **app** > **java** > **name of the project** > **ui**) then selecting **New** > **Kotlin Class/File**. Name the file `SongsFragment` and select **Class** from the list of options. Once the `SongsFragment.kt` file opens in the editor, modify its code so it

reads as follows:

```
import androidx.fragment.app.Fragment

class SongsFragment : Fragment() {

    private var _binding: FragmentSongsBinding? = null
    private val binding get() = _binding!!
    private var completeLibrary = mutableListOf<Song>()
    private var isProcessing = false
    private lateinit var songsAdapter: SongsAdapter
    private lateinit var musicViewModel: MusicViewModel
    private lateinit var callingActivity: MainActivity

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        _binding = FragmentSongsBinding.inflate(inflater, container, false)
        callingActivity = activity as MainActivity
        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        binding.recyclerView.layoutManager = LinearLayoutManager(activity)
        binding.recyclerView.itemAnimator = DefaultItemAnimator()

        // TODO: Initialise the SongsAdapter class here

        musicViewModel = ViewModelProvider(this)[MusicViewModel::class.java]
        musicViewModel.allSongs.observe(viewLifecycleOwner, { songs ->
            songs?.let {
                if (it.isNotEmpty() || completeLibrary.isNotEmpty()) processSongs(it)
            }
        })

        // Shuffle the music library then play it
        binding.fab.setOnClickListener {
            callingActivity.playNewSongs(completeLibrary, 0, true)
        }

        binding.recyclerView.addOnScrollListener(object: RecyclerView.OnScrollListener() {
            override fun onScrolled(recyclerView: RecyclerView, dx: Int, dy: Int) {
                super.onScrolled(recyclerView, dx, dy)
                if (dy > 0 && binding.fab.visibility == View.VISIBLE) binding.fab.hide()
                else if (dy < 0 && binding.fab.visibility != View.VISIBLE) binding.fab.show()
            }
        })
    }
}
```

The core components of the SongsFragment fragment are initialised within the onCreateView and onViewCreated methods. First, a linear layout manager and default item animator are attached to the RecyclerView from the fragment\_songs layout. The contents of the RecyclerView will stack vertically one atop of another, and a standard animation will be used when items are inserted, removed or updated. Next, the onCreateView method registers an observer on MusicViewModel view model's allSongs variable, which contains a list of every Song object in the user's music library. Changes to the list of Song objects will be handled by a method called processSongs, which we'll define shortly. Also, the onCreateView method attaches an onClick listener to the floating action button from the fragment\_songs layout. If the user clicks the button, then the MainActivity class's playNewSongs method will play the user's entire music library on shuffle. Finally, an onScroll listener is attached to the RecyclerView. If the user scrolls down the RecyclerView, then the floating action button will be hidden. Likewise, if the user scrolls up the RecyclerView, then the floating action button will be revealed.

The last thing we'll do in this section is set the `_binding` variable back to null when the fragment is being destroyed. This helps prevent the fragment from accessing components of the layout in the event the layout has been closed but the fragment is not yet shut down. Add the following code below the `onViewCreated` method:

```
override fun onDestroyView() {  
    super.onDestroyView()  
    _binding = null  
}
```

## Displaying songs in the RecyclerView

In this section, we will design and implement an adapter that will load the list of Song objects that comprise the user's music library into the RecyclerView widget from the `fragment_songs.xml` layout. To facilitate this, we need to create a layout file that will hold the details of each song. Create a new layout resource file by right-clicking the **layout** directory (**Project** > **app** > **res**) and selecting **New** > **Layout Resource File**. Name the layout `song_preview` then press OK. Once the layout opens in the editor, switch to Code view and edit the file so it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>  
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="match_parent"  
    android:layout_height="60dp"  
    android:background="?attr/selectableItemBackground">
```

```
    <ImageView  
        android:id="@+id/artwork"  
        android:layout_width="60dp"  
        android:layout_height="match_parent"  
        android:layout_alignParentStart="true"  
        android:layout_centerVertical="true"  
        android:contentDescription="@string/album_artwork" />
```

```
    <LinearLayout  
        android:layout_width="0dp"  
        android:layout_height="wrap_content"  
        android:orientation="vertical"  
        android:layout_toEndOf="@id/artwork"  
        android:layout_toStartOf="@id/menu"  
        android:layout_marginHorizontal="8dp"  
        android:layout_centerVertical="true" >
```

```
        <TextView  
            android:id="@+id/title"  
            android:layout_width="match_parent"  
            android:layout_height="wrap_content"  
            android:singleLine="true"  
            android:textColor="?attr/colorOnSurface"  
            android:textSize="16sp" />
```

```
        <TextView  
            android:id="@+id/artist"  
            android:layout_width="match_parent"  
            android:layout_height="wrap_content"  
            android:singleLine="true"  
            android:textSize="14sp"  
            android:textColor="@color/material_on_surface_emphasis_medium" />
```

```
    </LinearLayout>
```

```
    <ImageButton  
        android:id="@+id/menu"  
        android:layout_width="26dp"  
        android:layout_height="26dp"  
        android:layout_marginEnd="12dp"  
        android:src="@drawable/ic_more"
```

```

    android:layout_alignParentEnd="true"
    android:layout_centerVertical="true"
    android:contentDescription="@string/options_menu"
    style="@style/Widget.AppCompat.ActionButton.Overflow" />
</RelativeLayout>

```

The root element of the `song_preview` layout is a `RelativeLayout` widget. The `RelativeLayout` contains a background attribute set to `selectableItemBackground`, which means a ripple overlay effect will occur whenever the layout is pressed. The ripple will help show the user which song they have selected. Inside the `RelativeLayout`, there is an `ImageView` widget that will display album artwork and two `TextView` widgets that will contain information about the song title and artist. There is also a menu icon `ImageButton` widget that the user can press to load an options menu for that item.

Moving on, we'll now create an adapter class that will load the library of `Song` objects into the `RecyclerView` and handle user interactions. Right-click the `songs` directory then select **New > Kotlin Class/File**. Name the file `SongsAdapter` and select `Class` from the list of options. Once the `SongsAdapter.kt` file opens in the editor, edit its code so it reads as follows:

```

class SongsAdapter(private val activity: MainActivity):
    RecyclerView.Adapter<SongsAdapter.SongsViewHolder>(), FastScrollRecyclerView.SectionedAdapter {
    var songs = mutableListOf<Song>()

```

```

    override fun getSectionName(position: Int): String {
        return songs[position].title[0].uppercaseChar().toString()
    }

```

```

    inner class SongsViewHolder(itemView: View) :
        RecyclerView.ViewHolder(itemView),
        View.OnClickListener {

```

```

        internal var mArtwork = itemView.findViewById<View>(R.id.artwork) as ImageView
        internal var mTitle = itemView.findViewById<View>(R.id.title) as TextView
        internal var mArtist = itemView.findViewById<View>(R.id.artist) as TextView
        internal var mMenu = itemView.findViewById<ImageButton>(R.id.menu)

```

```

        init {
            itemView.isClickable = true
            itemView.setOnClickListener(this)
            itemView.setOnLongClickListener {
                // TODO: Open options dialog
                return@setOnLongClickListener true
            }
        }

```

```

        override fun onClick(view: View) {
            activity.playNewSongs(songs, layoutPosition, false)
        }

```

```

        override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): SongsViewHolder {
            return SongsViewHolder(LayoutInflater.from(parent.context).inflate(R.layout.song_preview, parent, false))
        }

```

```

        override fun onBindViewHolder(holder: SongsViewHolder, position: Int) {
            val current = songs[position]

```

```

            activity.insertArtwork(current.albumID, holder.mArtwork)

```

```

            holder.mTitle.text = current.title
            holder.mArtist.text = current.artist
            holder.mMenu.setOnClickListener {
                // TODO: Open options dialog
            }
        }

```

```

        override fun getItemCount() = songs.size

```

The SongsAdapter class features a parameter called activity in its primary constructor. The activity parameter will hold a reference to the MainActivity class and allow the adapter to reference the class's data. In the body of the adapter, we define a variable called songs which will store the Song objects that comprise the user's music library. A method called getSectionName is also defined. The getSectionName method will return the first letter of the title of the song associated with the user's scroll position in the RecyclerView. This letter will be loaded into the scrollbar's thumb to help the user find their position as they are scrolling through their music library. For example, the image below shows the letter M appearing in the thumb.

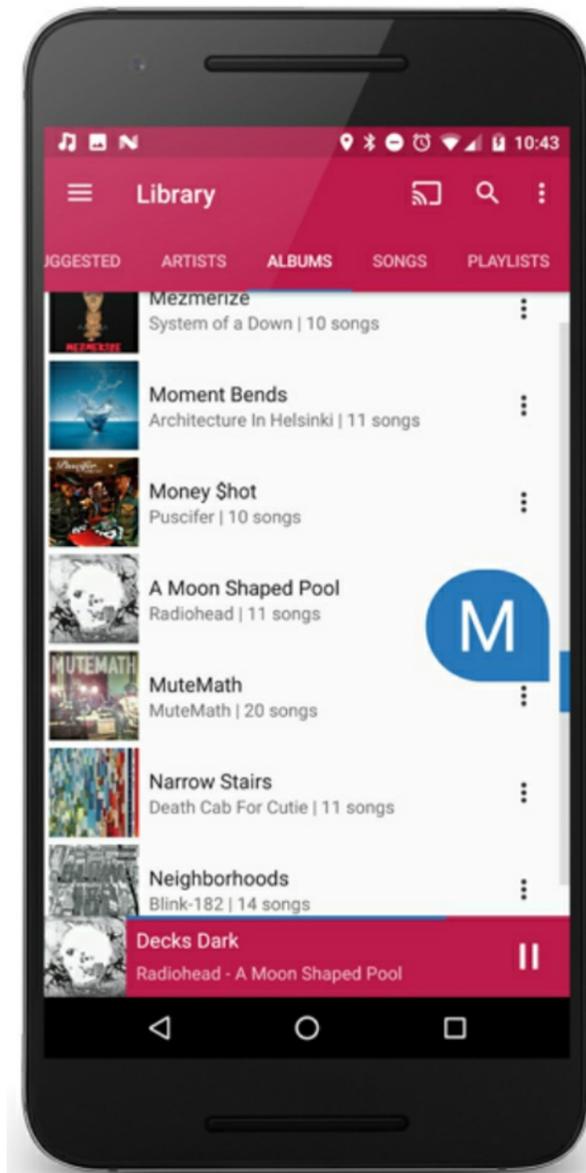


Image source: <https://github.com/timusus/RecyclerView-FastScroll>

Next, an inner class called SongsViewHolder is established. This inner class will initialise the components of the **song\_preview.xml** layout and define what actions will be performed if the user clicks or long clicks the layout. The adapter knows to use the **song\_preview.xml** layout because this is the layout that is inflated by the onCreateView method. If the user single clicks an item in the RecyclerView, then the list of Song objects loaded in the adapter and the index of the selected item are sent to the MainActivity class's playNewSongs method. The playNewSongs method will then begin to play the user's music library, starting with the user's selected song. Meanwhile, if the user long clicks an item (presses it for several seconds) then an options dialog will open. We'll discuss the options dialog in greater depth in the next section. In brief, it will allow the user to perform actions such as adding the song to the play queue.

The adapter also contains a method called onBindViewHolder, which populates the data at each position in the RecyclerView. In this case, the onBindViewHolder method retrieves the Song object from the songs variable associated with the current position in the RecyclerView. It then uses the information from the Song object to load the album artwork into the artwork ImageView widget and load the song's title and artist name into the two TextView widgets. The artwork image will be loaded by a method in the MainActivity class called insertArtwork. The last method in the adapter is called getItemCount and it will determine how many items are loaded into the RecyclerView. In this case, the number of items will equal the size of the user's music library.

Album artwork will be loaded into the artwork ImageView widget by a MainActivity method called insertArtwork. To define the insertArtwork method, open the **MainActivity.kt** file (**Project > app > java > name of the project**) and add the following code below the play method:

```
fun insertArtwork(albumID: String?, view: ImageView) {
```

```

var file: File? = null
if (albumID != null) {
    val cw = ContextWrapper(this)
    val directory = cw.getDir("albumArt", Context.MODE_PRIVATE)
    file = File(directory, "$albumID.jpg")
}
Glide.with(this)
    .load(file ?: R.drawable.ic_launcher_foreground)
    .transition(DrawableTransitionOptions.withCrossFade())
    .centerCrop()
    .signature(ObjectKey(file?.path + file?.lastModified()))
    .override(600, 600)
    .into(view)
}

```

When the user's music library is built, the artwork for each album will be stored as a JPEG image in an internal directory called albumArt. Each artwork file will be named according to the album's ID (e.g. 213.jpg), so the insertArtwork method begins by attempting to build a File object for the artwork image file. If the artwork for a given song is not available, then the file variable will remain null. Next, an image-rendering framework called Glide will insert the artwork into the ImageView widget that was supplied in the insertArtwork method's view argument. Glide will attempt to load the artwork image referenced in the File object; however, if the file variable is null then the ic\_launcher\_foreground drawable resource will be loaded instead.

Several further properties are applied to Glide to customise how images are handled. First, a crossfade transition is used to make images fade into the ImageView when loaded. Second, the centerCrop method is used to centrally position images within the ImageView. Next, a signature is generated for each image that is loaded. The signature will contain the file path of the image and the time the file was last modified. Using the signature will help Glide manage its cache, which contains the history of previously loaded images. If Glide is directed to reload an image, then it can retrieve the image from its cache, which requires significantly less working memory compared to loading the image file from scratch; however, if the image file has been updated then the cached version will not be suitable. This is where the signature comes in. The signature contains the time the image file was last modified and so if the underlying file has been changed then the signature will be different. A change in signature tells Glide that the image stored in its cache is outdated and must be reloaded. This feature will be especially useful if the user updates the artwork associated with an album.

## Handling changes in the user's music library

The user's music library will be regularly updated to ensure new audio files are added and deleted audio files are removed. Any changes to the music library will need to be reflected in the list of songs that are displayed in the songs fragment. To address this, open the **SongsFragment.kt** file (**Project > app > java > name of the project > ui > songs**) and add the following variables to the list of variables at the top of the class:

```

private var isProcessing = false
private lateinit var songsAdapter: SongsAdapter

```

The above code defines a variable called isProcessing that will prevent multiple requests to update the RecyclerView from being initiated simultaneously (especially important when the user's music library is being built for the first time). It also defines a variable called songsAdapter that will provide access to the SongsAdapter class. To initialise the songsAdapter variable and apply it to the RecyclerView, replace the TODO comment in the onCreateView method with the following code:

```

songsAdapter = SongsAdapter(callingActivity)
binding.recyclerView.adapter = songsAdapter
songsAdapter.stateRestorationPolicy = RecyclerView.Adapter.StateRestorationPolicy.PREVENT_WHEN_EMPTY

```

In addition to initialising the SongsAdapter class, the above code also applies a state restoration policy of PREVENT\_WHEN\_EMPTY to the RecyclerView adapter. This restoration policy directs the adapter to refrain from restoring the RecyclerView until its content has been reloaded into the adapter. This restoration policy will help preserve the user's scroll position when they return to the songs fragment from elsewhere in the app. Without the restoration policy, the adapter would attempt to restore the user's scroll position before the list of Song objects has been loaded into the RecyclerView. Invariably, the scroll position would be unavailable and the user would always return to the top of the RecyclerView, which could be frustrating.

Moving on, we'll now define a method called `processSongs` that will load `Song` objects into the `SongsAdapter` and handle changes in the user's music library. To define the `processSongs` method, add the following code below the `onViewCreated` method:

```
private fun processSongs(songList: List<Song>) {
    // Use the isProcessing boolean to prevent the processSongs method from running multiple times simultaneously
    // (e.g. when the library is being built for the first time)
    if (!isProcessing) {
        isProcessing = true
        completeLibrary = songList.sortedBy { song ->
            song.title.uppercase(Locale.ROOT)
        }.toMutableList()

        val songs = songsAdapter.songs
        songsAdapter.songs = completeLibrary
        when {
            songs.isEmpty() -> songsAdapter.notifyItemRangeInserted(0, completeLibrary.size)
            completeLibrary.size > songs.size -> {
                val difference = completeLibrary - songs.toSet()
                for (s in difference) {
                    val index = completeLibrary.indexOfFirst {
                        it.songID == s.songID
                    }
                    if (index != -1) songsAdapter.notifyItemInserted(index)
                }
            }
            completeLibrary.size < songs.size -> {
                val difference = songs - completeLibrary.toSet()
                for (s in difference) {
                    val index = songs.indexOfFirst {
                        it.songID == s.songID
                    }
                    if (index != -1) songsAdapter.notifyItemRemoved(index)
                }
            }
        }
        isProcessing = false
    }
}
```

The `processSongs` method will run whenever the observer that is registered to the `MusicViewModel` view model's `allSongs` variable is updated. When the music library is being built for the first time, the `allSongs` variable will be updated very frequently and the `processSongs` method could be run multiple times simultaneously. If one instance of the `processSongs` method is run before a previous instance has finished, then this can distort the data that is being supplied to the `SongsAdapter` class. To resolve this, the `processSongs` sets a variable called `isProcessing` to true when it begins running and reverts the variable to false when it finishes. The main body of the `processSongs` method can only run when the value of the `isProcessing` variable is false, which prevents multiple instances of the `processSongs` method from running simultaneously and attempting to update the `SongsAdapter` class at the same time.

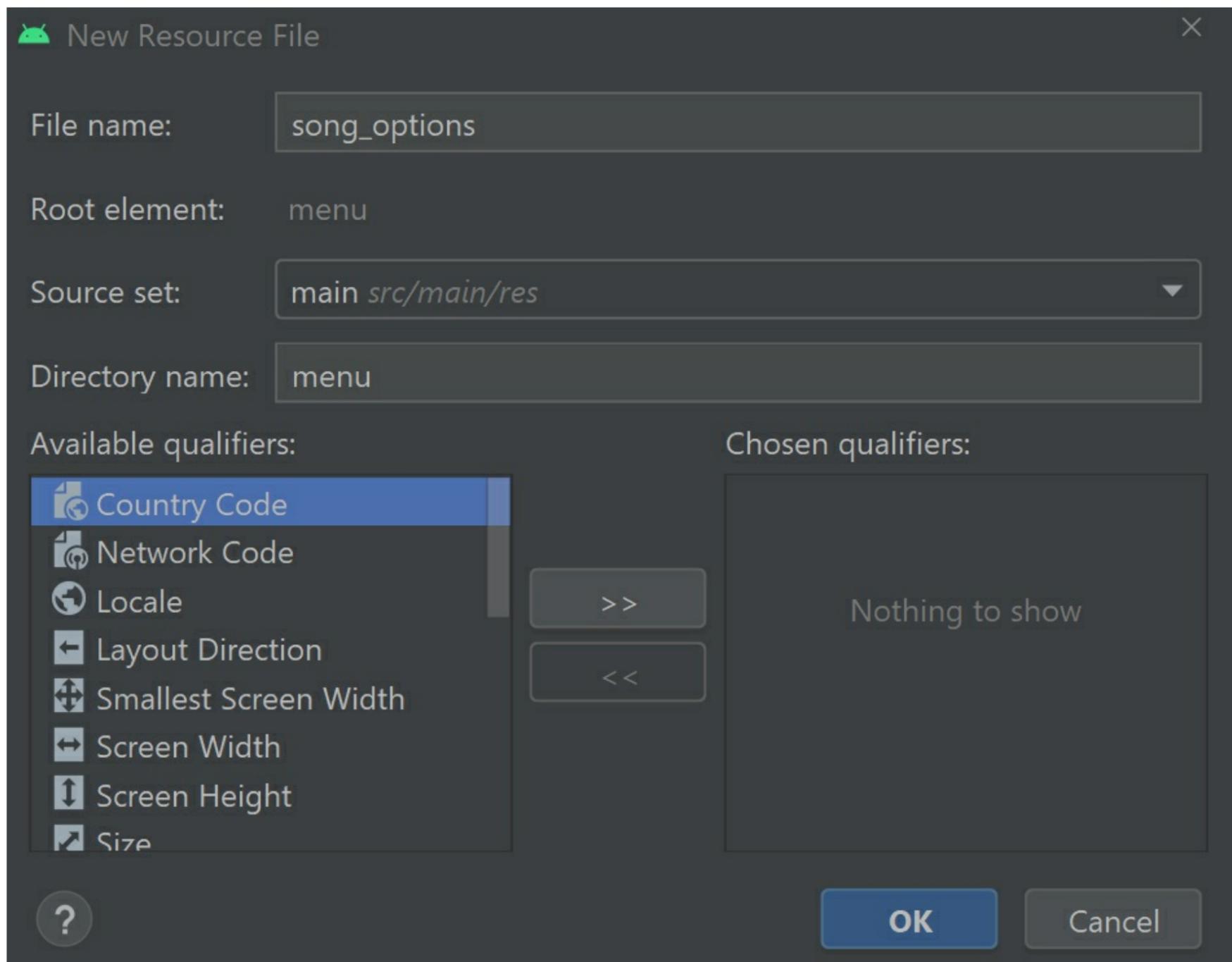
In the body of the `processSongs` method, the list of `Song` objects that comprise the user's music library is sorted alphabetically based on the first letter of the song's title. For sorting purposes, the first letter of the song title is capitalised because otherwise songs with lower case names will be sent to the end of the list. Next, the sorted list of `Song` objects is loaded into the `SongsAdapter` class's `songs` variable and a `when` block determines how best to update the `RecyclerView`. If the `SongsAdapter` class's `songs` variable had previously been empty, then it is likely the fragment has just been loaded and so the adapter's `notifyItemRangeInserted` method is used to load all the `Song` objects at once.

On the other hand, if the list of songs in the adapter was not empty, then this suggests the user's music library has already been loaded but one or more songs have been added or removed. To find out how the list of `Song` objects has changed, the `when` block checks whether the new list of songs is longer or shorter than the list of songs in the adapter. If the new list is longer, then the added songs are found by subtracting the contents of the old list from the

new list. Any remaining songs (i.e. the ones to be added) are stored in a variable called difference. The index of each new Song object is then run through the adapter's notifyItemInserted method to update the RecyclerView and display the new song. Vice versa, if one or more songs have been removed, then the index of any removed Song objects are run through the adapter's notifyItemRemoved method to update the RecyclerView.

## Handling user interactions with songs

If the user long clicks an item in the songs fragment's RecyclerView, a popup menu will invite the user to add the song to the play queue or edit its metadata. To define the contents of the popup menu, locate the **menu** directory (**Project > app > res**) then select **New > Menu Resource File**. Name the file `song_options` then click OK.



Open the **song\_options.xml** resource file in Code view and add the following two items inside the menu element:

```
<item android:id="@+id/play_next"
    android:title="@string/play_next" />
<item android:id="@+id/edit_metadata"
    android:title="@string/edit_metadata" />
```

Each menu item contains an ID, which we can use to refer to the menu item, and a title, which displays text to the user. In this case, the two menu items will invite the user to play the selected song next or edit its metadata, respectively. To make the popup menu operational, return to the MainActivity class and add the following method below the insertArtwork method:

```
fun showSongPopup(view: View, song: Song) {
    PopupMenu(this, view).apply {
        inflate(R.menu.song_options)
        setOnMenuItemClickListener {
            when (it.itemId) {
                R.id.play_next -> {
                    playNext(song)
                }
            }
        }
    }
}
```

```

        true
    }
    R.id.edit_metadata -> {
        val action = SongsFragmentDirections.actionEditSong(song)
        findNavController(R.id.nav_host_fragment).navigate(action)
        true
    }
    else -> super.onOptionsItemSelected(it)
}
}
show()
}
}
}

```

Note you may need to add the following import statements to the top of the file:

```

import android.widget.PopupMenu
import androidx.navigation.findNavController

```

The `showSongPopup` method receives two arguments: the layout View that the popup menu will appear over (i.e. the RecyclerView item that the user has long pressed) and the Song object associated with the selected RecyclerView item. The `showSongPopup` method then uses an instance of the `PopupMenu` class to inflate the `song_options.xml` menu resource file and applies an action to each menu item. If the `play_music` menu item is clicked, then the selected Song object will be added to the next position in the play queue by a MainActivity method called `playNext`. Meanwhile, the `edit_metadata` menu item will transport the user to a fragment called `EditSongFragment` via an action called `action_edit_song` that we defined in the `mobile_navigation` navigation graph. The user's selected Song object is packaged in the action so it can be retrieved by the `EditSongFragment` class.

The song options popup menu is now complete; however, there is some extra code we need to write for it to open when the user long presses an item in the SongsFragment RecyclerView. First, open the **SongsAdapter.kt** file (**Project > app > java > name of the project > ui > songs**) and replace the TODO comment in the `SongsViewHolder` inner class with the following code:

```

activity.showSongPopup(it, songs[layoutPosition])

```

Likewise, replace the TODO comment in the menu button's `onClick` listener found in the `onBindViewHolder` method with the following code:

```

activity.showSongPopup(it, current)

```

Now, if the user long presses an item in the RecyclerView or clicks the menu button for an item, the adapter will run the MainActivity class's `showSongPopup` method to load the options menu and invite the user to queue the song or edit its metadata.

## Setting up the EditSongInfo fragment and layout

In this section, we'll create the fragment that will enable the user to update the metadata of the songs in their music library. The edit song info fragment will require a layout. To create a new layout file, right-click the **layout** directory then select **New > Layout Resource File**. Name the file `fragment_edit_song` then press OK. Once the `fragment_edit_song.xml` layout opens in the editor, switch to Code view and modify the file so it reads as follows:

```

<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_height="match_parent"
    android:layout_width="match_parent"
    android:scrollbars="none">

    <RelativeLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:paddingBottom="12dp" >

        <ImageView
            android:id="@+id/editSongArtwork"
            android:layout_width="match_parent"

```

```
    android:layout_height="300dp"
    android:clickable="true"
    android:contentDescription="@string/set_album_artwork"
    android:layout_alignParentTop="true" />
```

```
<ImageView
    android:id="@+id/editSongArtworkIcon"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/ic_edit"
    android:clickable="true"
    android:layout_margin="12dp"
    android:contentDescription="@string/set_album_artwork"
    android:layout_alignBottom="@id/editSongArtwork"
    android:layout_alignEnd="@id/editSongArtwork" />
```

```
<TextView
    android:id="@+id/editSongInfo"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="8dp"
    android:text="@string/edit_metadata"
    android:textSize="14sp"
    android:textColor="@color/design_default_color_primary"
    android:layout_below="@id/editSongArtwork" />
```

```
<TextView
    android:id="@+id/editSongTitleHeading"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="8dp"
    android:text="@string/title"
    android:textSize="12sp"
    android:layout_below="@id/editSongInfo" />
```

```
<EditText
    android:id="@+id/editSongTitle"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginHorizontal="8dp"
    android:textSize="16sp"
    android:inputType="text"
    android:maxLength="100"
    android:hint="@string/title"
    android:importantForAutofill="no"
    android:layout_below="@id/editSongTitleHeading" />
```

```
<TextView
    android:id="@+id/editSongArtistHeading"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="8dp"
    android:text="@string/artist"
    android:textSize="12sp"
    android:layout_below="@id/editSongTitle" />
```

```
<EditText
    android:id="@+id/editSongArtist"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginHorizontal="8dp"
    android:textSize="16sp"
    android:inputType="text"
```

```
    android:maxLength="100"  
    android:hint="@string/artist"  
    android:importantForAutofill="no"  
    android:layout_below="@id/editSongArtistHeading" />
```

```
<TextView  
    android:id="@+id/editSongDiscHeading"  
    android:layout_width="60dp"  
    android:layout_height="wrap_content"  
    android:layout_margin="8dp"  
    android:text="@string/disc"  
    android:textSize="12sp"  
    android:layout_below="@id/editSongArtist" />
```

```
<EditText  
    android:id="@+id/editSongDisc"  
    android:layout_width="60dp"  
    android:layout_height="wrap_content"  
    android:layout_marginHorizontal="8dp"  
    android:textSize="16sp"  
    android:inputType="number"  
    android:maxLength="1"  
    android:hint="@string/disc"  
    android:importantForAutofill="no"  
    android:layout_below="@id/editSongDiscHeading" />
```

```
<TextView  
    android:id="@+id/editSongTrackHeading"  
    android:layout_width="60dp"  
    android:layout_height="wrap_content"  
    android:layout_marginHorizontal="16dp"  
    android:layout_marginVertical="8dp"  
    android:text="@string/track"  
    android:textSize="12sp"  
    android:layout_below="@id/editSongArtist"  
    android:layout_toEndOf="@id/editSongDiscHeading" />
```

```
<EditText  
    android:id="@+id/editSongTrack"  
    android:layout_width="60dp"  
    android:layout_height="wrap_content"  
    android:textSize="16sp"  
    android:inputType="number"  
    android:maxLength="3"  
    android:hint="@string/track"  
    android:importantForAutofill="no"  
    android:layout_below="@id/editSongTrackHeading"  
    android:layout_alignStart="@id/editSongTrackHeading" />
```

```
<TextView  
    android:id="@+id/editSongYearHeading"  
    android:layout_width="100dp"  
    android:layout_height="wrap_content"  
    android:layout_margin="8dp"  
    android:text="@string/year"  
    android:textSize="12sp"  
    android:layout_below="@id/editSongArtist"  
    android:layout_toEndOf="@id/editSongTrackHeading" />
```

```
<EditText  
    android:id="@+id/editSongYear"  
    android:layout_width="100dp"  
    android:layout_height="wrap_content"
```

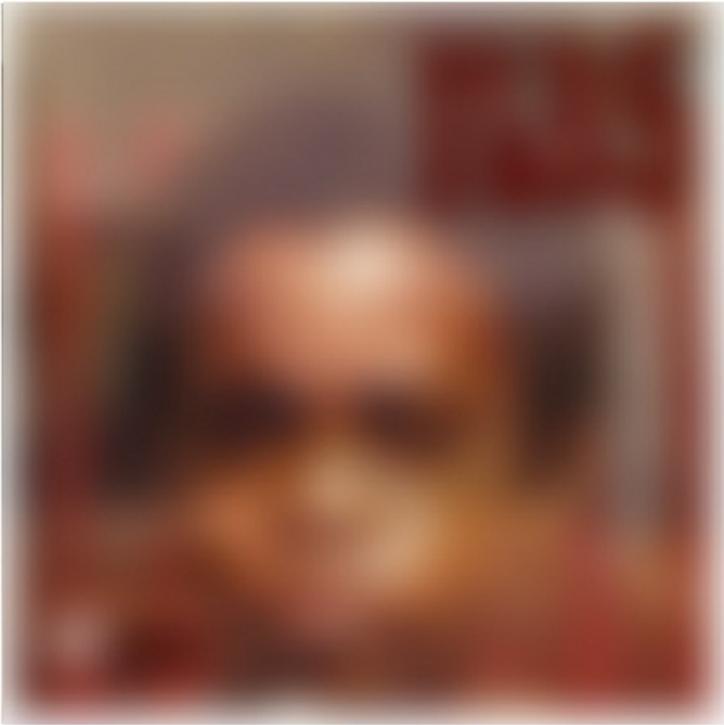
```

        android:textSize="16sp"
        android:inputType="number"
        android:maxLength="4"
        android:hint="@string/year"
        android:importantForAutofill="no"
        android:layout_below="@id/editSongYearHeading"
        android:layout_alignStart="@id/editSongYearHeading" />
</RelativeLayout>
</ScrollView>

```

The root element of the `fragment_edit_song` layout is a `ScrollView` widget that will allow the user to scroll down if the layout's contents are too large to fit in the window. The `ScrollView` widget has a `scrollbars` attribute set to `none`, which means while the user can still scroll the layout's content, no scrollbar will be visible down the side of the screen. Inside the `ScrollView` widget, there is a `RelativeLayout` widget that will organise the various `TextView`, `EditText` and `ImageView` widgets that will help the user edit the song's information. The `EditText` widgets will be populated with the song's metadata. Each `EditText` widget is accompanied by a `TextView` widget that labels what data is being displayed (e.g. title, artist, album etc.).

←
**Edit music info**
SAVE



**Edit music info**

Title

**One Love**

---

Artist

**Nas**

---

Disc	Track	Year
<b>1</b>	<b>7</b>	<b>1994</b>

*Album artwork blurred for copyright reasons.*

The user can edit the information in the `EditText` widgets if they wish. Some `EditText` widgets have extra attributes to restrict user input. For example, several `EditText` widgets feature a `maxLength` attribute of 100 to ensure the user can not enter values that are over 100 characters long, while others have an `inputType` of "number" which means users can only enter numeric values (e.g. for the song's disc or track number). The song's current album artwork is inserted into an `ImageView` widget at the top of the layout. A pen icon will be superimposed over the bottom right corner of the artwork to signal to the user that they can change the artwork.

Changes in a song's metadata will be handled by a dedicated fragment. Create a new Kotlin class by right-clicking the **songs** directory (**Project** > **app** > **java** > **name of the project** > **ui** > **songs**) then selecting **New** > **Kotlin Class/File**. Name the file `EditSongFragment` and select **Class** from the list of options. Once the `EditSongFragment.kt` file opens in the editor, modify its code so it reads as follows:

```
import androidx.fragment.app.Fragment

class EditSongFragment : Fragment() {

    private var _binding: FragmentEditSongBinding? = null
    private val binding get() = _binding!!
    private var song: Song? = null
    private var newArtwork: Bitmap? = null
    private lateinit var callingActivity: MainActivity

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        arguments?.let {
            val safeArgs = EditSongFragmentArgs.fromBundle(it)
            song = safeArgs.song
        }

        _binding = FragmentEditSongBinding.inflate(inflater, container, false)
        setHasOptionsMenu(true)
        callingActivity = activity as MainActivity

        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        var editable: Editable = SpannableStringBuilder(song!!.title)
        binding.editSongTitle.text = editable

        editable = SpannableStringBuilder(song!!.artist)
        binding.editSongArtist.text = editable

        editable = SpannableStringBuilder(song!!.track.toString().substring(0, 1))
        binding.editSongDisc.text = editable

        editable = SpannableStringBuilder(song!!.track.toString().substring(1, 4).toInt().toString())
        binding.editSongTrack.text = editable

        editable = SpannableStringBuilder(song!!.year)
        binding.editSongYear.text = editable

        // Retrieve the song's album artwork
        callingActivity.insertArtwork(song!!.albumID, binding.editSongArtwork)

        // TODO: Define edit song artwork action here
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

In the above code, the `onCreateView` method retrieves the arguments which were supplied when the `EditSongFragment` was opened. Referring back to the `mobile_navigation` navigation graph, we can see the `EditSongFragment` fragment has an argument called `song`, and so we use that name to retrieve the user's selected `Song` object from the fragment's `Safe Args` class. The `Song` object is stored in a variable called `song` so it can be used elsewhere in the fragment.

The relevant details from the Song object are loaded into the fragment\_edit\_song layout's EditText widgets via the layout's binding class. To load text into an EditText widget, the text must first be converted to an Editable object, which is a string that can be edited by the user. Next, the album artwork is inserted into the ImageView widget using the MainActivity class's insertArtwork method. If the user clicks the artwork ImageView or the pen icon ImageView that appears in the bottom right corner of the artwork, then a window will appear allowing the user to search their device for a new image to use as the album artwork. To put this feature into effect, replace the TODO comment in the onCreateView method with the following code:

```
binding.editSongArtwork.setOnClickListener {
    registerResult.launch(Intent(Intent.ACTION_PICK, MediaStore.Images.Media.INTERNAL_CONTENT_URI))
}

binding.editSongArtworkIcon.setOnClickListener {
    registerResult.launch(Intent(Intent.ACTION_PICK, MediaStore.Images.Media.INTERNAL_CONTENT_URI))
}
```

The above code assigns both ImageView widgets an onClick listener that will launch an intent with an action of ACTION\_PICK. The ACTION\_PICK action means that the intent expects the user to select a data item using the device's document provider. The type of data item is defined in the second parameter of the intent, which in this case is the URI associated with an image file on the user's device. To respond to the user's selection, add the following code below the list of variables at the top of the class:

```
private val registerResult = registerForActivityResult(ActivityResultContracts.StartActivityForResult()) { result ->
    if (result.resultCode == AppCompatActivity.RESULT_OK) {
        try {
            val selectedImageUri = result.data?.data ?: return@registerForActivityResult
            val source = ImageDecoder.createSource(requireActivity().contentResolver, selectedImageUri)
            newArtwork = ImageDecoder.decodeBitmap(source)

            Glide.with(this)
                .load(selectedImageUri)
                .centerCrop()
                .into(binding.editSongArtwork)
        } catch (ignore: FileNotFoundException) {
        } catch (ignore: IOException) {
        }
    }
}
```

The registerResult variable defined above launches an intent using a method called registerForActivityResult. If the intent is executed successfully and the user selects an image, then the result of the request will be RESULT\_OK. In which case, the URI of the selected image is retrieved from the intent result data. Next, the raw image data associated with the URI is extracted using the ImageDecoder class's createSource method. For storage purposes, the raw image data is converted to a Bitmap representation of the image. Finally, the image rendering framework Glide replaces the artwork that was loaded into the artwork ImageView with the user's selected image.

The user can save their changes to the song's metadata by pressing a menu item in the app toolbar.



## Edit music info

SAVE

To display the save menu item, add the following code below the onCreateView method:

```
override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    menu.findItem(R.id.search).isVisible = false
    menu.findItem(R.id.save).isVisible = true

    super.onCreateOptionsMenu(menu, inflater)
}
```

The onCreateOptionsMenu method hides the search icon menu item and reveals the save menu item. Both menu items were defined earlier in the main.xml menu resource file. To make the save menu item operational and persist the user's changes to the song's metadata, add the following code below the onCreateOptionsMenu method:

```

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    return when (item.itemId) {
        R.id.save -> {
            val newTitle = binding.editSongTitle.text.toString()
            val newArtist = binding.editSongArtist.text.toString()
            val newDisc = binding.editSongDisc.text.toString()
            val newTrack = binding.editSongTrack.text.toString()
            val newYear = binding.editSongYear.text.toString()

            // Check no fields are blank
            for (string in listOf(newTitle, newArtist, newDisc, newTrack, newYear)) {
                if (string.isBlank()) {
                    Toast.makeText(activity, getString(R.string.check_fields_not_empty), Toast.LENGTH_SHORT).show()
                    return true
                }
            }

            val completeTrack = when (newTrack.length) {
                3 -> newDisc + newTrack
                2 -> newDisc + "0" + newTrack
                else -> newDisc + "00" + newTrack
            }.toInt()

            // Save the new album artwork if the artwork has been changed
            if (newArtwork != null){
                val file = callingActivity.getArtworkFile(song?.albumID!!)
                callingActivity.saveImage(newArtwork!!, file)
            }

            // Check if any metadata fields require updating
            if (newTitle != song!!.title || newArtist != song!!.artist || completeTrack != song!!.track || newYear !=
            song!!.year) {
                song!!.title = newTitle
                song!!.artist = newArtist
                song!!.track = completeTrack
                song!!.year = newYear

                callingActivity.updateSongInfo(song!!)
            }

            Toast.makeText(activity, getString(R.string.details_saved), Toast.LENGTH_SHORT).show()
            requireView().findNavController().popBackStack()
            true
        }

        else -> super.onOptionsItemSelected(item)
    }
}

```

Note you may need to add the following import statement to the top of the file:

```
import androidx.navigation.findNavController
```

The `onOptionsItemSelected` method responds to clicks of the save menu item by retrieving the contents of all the `EditText` widgets in the `fragment_edit_photo` layout. If any of the strings of text are empty, then a toast notification will advise the user that they need to provide the missing information. Next, the method proceeds to save the new album artwork image (if necessary) and update the `Song` object with the metadata values provided by the user. The updated `Song` object is sent to the database via a `MainActivity` method called `updateSongInfo`. Finally, the user is transported back to the fragment that was open before the `EditSongFragment` fragment using the `NavController` class's `popBackStack` method. The `popBackStack` method restores the previous destination in the user's navigation history for a given navigation graph.

## Saving changes to a song's metadata

There are several methods we must add to the `MainActivity` class to save changes to a song's metadata. The first

method is called `getArtworkFile` and will generate a `File` object that details where the artwork image file for a given album will be stored. To define the `getArtworkFile` method, open the `MainActivity.kt` file (**Project > app > java > name of the project**) and add the following code below the `showSongPopup` method:

```
fun getArtworkFile(filename: String): File {
    val cw = ContextWrapper(application)
    val directory = cw.getDir("albumArt", Context.MODE_PRIVATE)
    return File(directory, "$filename.jpg")
}
```

The `getArtworkFile` method contains an argument that accepts a string detailing the album ID of the album the artwork is associated with. The album ID will contribute to the filename. All artwork images are stored in an internal directory called `albumArt`. The `albumArt` directory is accessed using the `getDir` method, which will also create the directory if it does not already exist. The operating mode for the `albumArt` directory is set to `MODE_PRIVATE`, which means the directory and its contents will only be accessible to this application. A `File` object containing the details of the directory and the image's filename is returned by the method once the above processing is complete.

The `File` object returned by the `getArtworkFile` method can be passed to a method called `saveImage`, which uses the `File` object to save images in the app's storage. To define the `saveImage` method, add the following code below the `getArtworkFile` method:

```
fun saveImage(bitmap: Bitmap, path: File) {
    try {
        FileOutputStream(path).apply {
            // Use the compress method on the BitMap object to write image to the OutputStream
            bitmap.compress(Bitmap.CompressFormat.JPEG, 100, this)
            close()
        }
    } catch (ignore: Exception) { }
}
```

The `saveImage` method creates an instance of the `FileOutputStream` class and uses it to write the image bitmap to the `albumArt` directory defined in the `File` object. The image writing is initiated using the `Bitmap` class's `compress` method, which also specifies the format of the output image (JPEG in this instance) and the image quality (100 equals maximum quality, 0 equals minimum quality). Once the image file has been written, the `FileOutputStream` object is closed to prevent memory leaks.

Moving on, let's turn our attention to a method called `updateSongInfo`, which will send updated `Song` objects to the `Room` database. To do this, the `updateSongInfo` method will need to interact with the `MusicViewModel` view model, so add the following variable to the list of variables at the top of the `MainActivity` class:

```
private lateinit var musicViewModel: MusicViewModel
```

Initialise the variable by adding the following code to the `onCreate` method:

```
musicViewModel = ViewModelProvider(this)[MusicViewModel::class.java]
```

Once the `musicViewModel` variable has been initialised, define the `updateSongInfo` method by adding the following code below the `saveImage` method:

```
fun updateSongInfo(song: Song) = lifecycleScope.launch(Dispatchers.Default) {
    musicViewModel.updateMusicInfo(song)

    // See if the play queue needs to be updated
    if (playQueue.isNotEmpty()) {
        val newQueue = playQueue

        fun findIndex(): Int {
            return newQueue.indexOfFirst {
                it.second.songID == song.songID
            }
        }

        // HashMap key = queue index, value = queue item ID
        val queueIndexQueueIDMap = HashMap<Int, Int>()
```

```

do {
    val index = findIndex()
    if (index != -1) {
        queueIndexQueueIDMap[index] = playQueue[index].first
        newQueue.removeAt(index)
    }
} while (index != -1)

// Add the affected queue items (with updated metadata) back to the play queue
for ((index, queueID) in queueIndexQueueIDMap) {
    val queueItem = Pair(queueID, song)
    newQueue.add(index, queueItem)
}
playbackViewModel.currentPlayQueue.postValue(newQueue)
}
}

```

The `updateSongInfo` method sends the updated `Song` object to the `MusicViewModel` view model, which in turn will update the entry for that song in the Room database. We also need to update areas of the app that might be using the song, such as the play queue. For this purpose, an internal method called `findIndex` will find the index of the first occurrence of the song in the play queue. A `do/while` block is used to repeatedly run the `findIndex` method until the returned index equals `-1`, which means no further instances of the song could be found. The queue index and queue item ID of each occurrence of the song are added to a Hashmap called `queueIndexQueueIDMap`. The Hashmap contains a collection of key/value pairs. In this case, the key will equal the index of the item within the play queue and the value will equal the ID that was assigned to the `Pair` object when it was added to the play queue. Often, the ID and index will be the same; however, if the play queue has been shuffled or items have been added or removed then the values could be different. After the index and ID values have been added to the Hashmap, the queue item is removed from the play queue because it contains an outdated `Song` object.

Once all instances of the song in the play queue have been mapped to the Hashmap and removed from the play queue, replacement `Pair` objects are generated using the up to date `Song` object and the queue item IDs stored in the Hashmap. The updated `Pair` objects are then inserted into the play queue at the indices specified in the Hashmap. In this way, we update every instance of the song in the play queue while preserving the original order of the queue items.

## Setting up the PlayQueue fragment and layout

In this section, we will design the play queue fragment. The fragment will require a layout. To create a new layout file, right-click the **layout** directory then select **New > Layout Resource File**. Name the file `fragment_play_queue` then press OK. Once the `fragment_play_queue.xml` layout opens in the editor, switch to Code view and modify the file so it reads as follows:

```

<?xml version="1.0" encoding="utf-8"?>
<androidx.recyclerview.widget.RecyclerView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:contentDescription="@null" />

```

The `fragment_play_queue` layout is straightforward and simply contains a `RecyclerView` widget that will display the contents of the play queue. To make the play queue fragment operational, create a new Kotlin class by right-clicking the **playQueue** directory (**Project > app > java > name of the project > ui**) then selecting **New > Kotlin Class/File**. Name the file `PlayQueueFragment` and select `Class` from the list of options. Once the `PlayQueueFragment.kt` file opens in the editor, modify its code so it reads as follows:

```

import androidx.fragment.app.Fragment

class PlayQueueFragment : Fragment() {
    private var _binding: FragmentPlayQueueBinding? = null
    private val binding get() = _binding!!
    private lateinit var callingActivity: MainActivity

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,

```

```

savedInstanceState: Bundle?
): View {
    _binding = FragmentPlayQueueBinding.inflate(inflater, container, false)
    callingActivity = activity as MainActivity

    return binding.root
}

@SuppressLint("NotifyDataSetChanged")
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    binding.root.layoutManager = LinearLayoutManager(activity)
    binding.root.itemAnimator = DefaultItemAnimator()

    // TODO: Initialise PlayQueueAdapter here
}

override fun onDestroyView() {
    super.onDestroyView()
    _binding = null
}
}

```

In the above code, the `onCreateView` method initialises the `fragment_play_queue` layout's binding class so the fragment can interact with the layout's components. The only widget in the layout is a `RecyclerView`, and so the above code assigns the `RecyclerView` a linear layout manager to stack the contents of the `RecyclerView` vertically. Also, an item animator is applied, which will provide a set of standard animations when `RecyclerView` items are added, removed or updated.

## Displaying the play queue

In this section, we'll design an adapter that will handle the user's play queue. It will display the details of every song in the play queue and allow the user to rearrange the songs if they wish. To facilitate this, we need to create a layout that will display the details for a given play queue item. Create a new layout resource file in the usual way, by right-clicking the **layout** directory (**Project** > **app** > **res**) then selecting **New** > **Layout Resource File**. Name the layout `queue_item` then press OK. Once the layout opens in the editor, switch it to Code view and edit the file so it reads as follows:

```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout 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="wrap_content"
    android:background="?attr/selectableItemBackground"
    android:padding="12dp" >

    <ImageView
        android:id="@+id/handle"
        android:layout_width="40dp"
        android:layout_height="match_parent"
        android:src="@drawable/ic_drag_handle"
        android:contentDescription="@string/handle_view_desc"
        android:layout_alignParentStart="true"
        android:layout_centerVertical="true"
        app:tint="@color/material_on_surface_emphasis_medium" />

    <LinearLayout
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:layout_toEndOf="@id/handle"
        android:layout_toStartOf="@id/menu"
        android:layout_marginHorizontal="8dp"

```

```
android:layout_centerVertical="true" >
```

```
<TextView  
    android:id="@+id/title"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:singleLine="true"  
    android:textSize="16sp" />
```

```
<TextView  
    android:id="@+id/artist"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:singleLine="true"  
    android:textSize="14sp"/>
```

```
</LinearLayout>
```

```
<ImageButton  
    android:id="@+id/menu"  
    android:layout_width="25dp"  
    android:layout_height="25dp"  
    android:src="@drawable/ic_more"  
    android:contentDescription="@string/options_menu"  
    android:layout_alignParentEnd="true"  
    android:layout_centerVertical="true"  
    style="@style/Widget.AppCompat.ActionButton.Overflow" />
```

```
</RelativeLayout>
```

The root element of the `queue_item` layout is a `RelativeLayout` widget. On the left-hand side of the layout, there is an `ImageView` widget containing an icon of a handle that will allow the user to drag and reorder items in the play queue. Meanwhile, on the right-hand side of the layout, there is an `ImageButton` widget that will open an options menu when clicked. The remainder of the layout is occupied by a `LinearLayout` widget containing two `TextView` widgets. The `TextView` widgets will contain the song's title and artist name, respectively. Both widgets have a `singleLine` attribute set to `true` which will restrict each widget's contents to a single line of text and help ensure each play queue item occupies the same amount of space.

Moving on, we'll now create an adapter that will populate the `RecyclerView` and handle user interactions. Right-click the **playQueue** directory then select **New > Kotlin Class/File**. Name the file `PlayQueueAdapter` and select `Class` from the list of options. Once the **PlayQueueAdapter.kt** file opens in the editor, edit its code as follows:

```
class PlayQueueAdapter(private val fragment: PlayQueueFragment, private val activity: MainActivity):  
    RecyclerView.Adapter<PlayQueueAdapter.PlayQueueViewHolder>() {  
    var currentlyPlayingQueueID = -1  
    var playQueue = mutableListOf<Pair<Int, Song>>()  
  
    inner class PlayQueueViewHolder(itemView: View) :  
        RecyclerView.ViewHolder(itemView),  
        View.OnClickListener {  
  
        internal var mTitle = itemView.findViewById<View>(R.id.title) as TextView  
        internal var mArtist = itemView.findViewById<View>(R.id.artist) as TextView  
        internal var mHandle = itemView.findViewById<ImageView>(R.id.handle)  
        internal var mMenu = itemView.findViewById<ImageButton>(R.id.menu)  
  
        init {  
            itemView.isClickable = true  
            itemView.setOnClickListener(this)  
        }  
  
        override fun onClick(view: View) {  
            // TODO: Skip to a different position in the play queue  
        }  
    }  
}
```

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): PlayQueueViewHolder {
```

```

return PlayQueueViewHolder(LayoutInflater.from(parent.context).inflate(R.layout.queue_item, parent, false))
}

@SuppressLint("ClickableViewAccessibility")
override fun onBindViewHolder(holder: PlayQueueViewHolder, position: Int) {
    val currentSong = playQueue[position].second

    holder.mTitle.text = currentSong.title
    holder.mArtist.text = currentSong.artist

    val emphasisColour = if (playQueue[position].first == currentlyPlayingQueueID) {
        ContextCompat.getColor(activity, R.color.design_default_color_secondary)
    } else ContextCompat.getColor(activity, R.color.material_on_surface_emphasis_medium)

    holder.mTitle.setTextColor(emphasisColour)
    holder.mArtist.setTextColor(emphasisColour)

    holder.mHandle.setOnTouchListener { _, event ->
        // TODO: Handle the drag action here
        return@setOnTouchListener true
    }

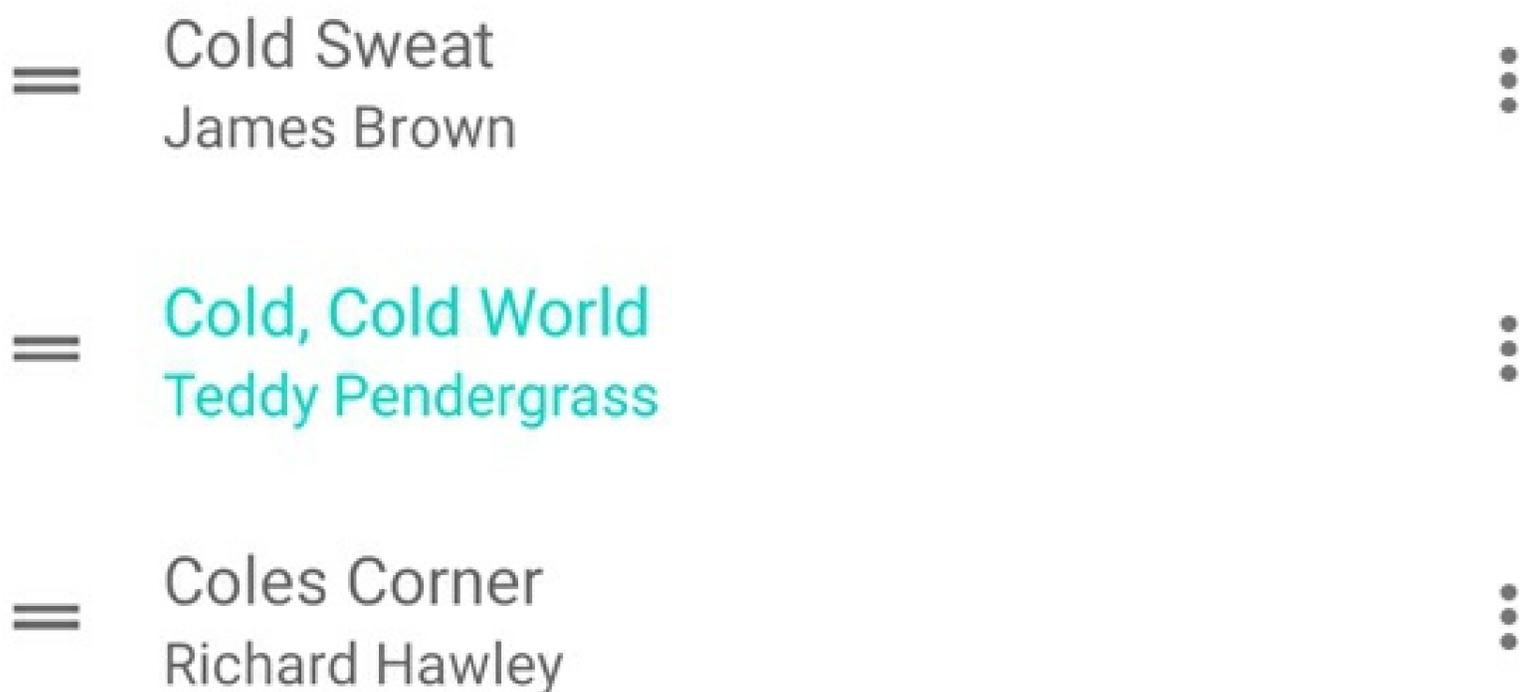
    holder.mMenu.setOnClickListener {
        // TODO: Open the options menu here
    }
}

override fun getItemCount() = playQueue.size
}

```

The PlayQueueAdapter class contains two variables in its primary constructor called fragment and activity. The variables will store instances of the PlayQueueFragment class and the MainActivity class, respectively. In the body of the adapter, a variable called currentlyPlayingQueueID will store the ID value of the Pair object associated with the currently playing song. The value of this variable will help distinguish the currently playing song from the rest of the play queue. Also, there is a variable called playQueue, which will store the complete list of Pair objects that form the play queue.

Next, an inner class called PlayQueueViewHolder is established. This inner class will initialise the components of the **queue\_item.xml** layout and help handle user interactions. The adapter knows to use the **queue\_item.xml** layout because this is the layout that is inflated by the onCreateViewHolder method. The adapter also contains a method called onBindViewHolder, which is responsible for populating the data at each position in the RecyclerView. In this case, the onBindViewHolder retrieves the corresponding Song object for the given position in the RecyclerView and play queue. It then uses the Song object's information to populate the song's title and artist name TextView widgets. The onBindViewHolder method also checks whether the ID of the queue item being displayed matches the ID of the currently playing queue item. If there is a match, then the text colour is set to the secondary colour of the active theme to highlight the currently playing queue item. Otherwise, the regular medium emphasis onSurface colour is used, as shown below.



To ensure the play queue is up to date we must register an observer on the PlaybackViewModel view model's currentPlayQueue variable. To implement the observer, open the **PlayQueueFragment.kt** file (**Project > app > java > name of the project > ui > playQueue**) and add the following variables to the top of the class:

```
private val playbackViewModel: PlaybackViewModel by activityViewModels()
private lateinit var playQueueAdapter: PlayQueueAdapter
```

Note you may also need to add the following import statement to the top of the file:

```
import androidx.fragment.app.activityViewModels
```

The above variables will provide access to the PlaybackViewModel class and the PlayQueueAdapter class, respectively. To initialise the playQueueAdapter variable and apply it to the RecyclerView, replace the TODO comment in the onViewCreated method with the following code:

```
playQueueAdapter = PlayQueueAdapter(this, callingActivity)
binding.root.adapter = playQueueAdapter
```

Moving on, add the following code to the bottom of the onViewCreated method to register an observer to the PlaybackViewModel's currentPlayQueue variable:

```
playbackViewModel.currentPlayQueue.observe(viewLifecycleOwner, { queue ->
    queue?.let {
        if (playQueueAdapter.playQueue.size > it.size) {
            // Song(s) removed from the play queue
            val difference = playQueueAdapter.playQueue - it.toSet()
            for (item in difference) {
                val index = playQueueAdapter.playQueue.indexOfFirst { queueItem ->
                    queueItem.first == item.first
                }
                if (index != -1) {
                    playQueueAdapter.playQueue.removeAt(index)
                    playQueueAdapter.notifyItemRemoved(index)
                }
            }
        } else {
            // Adapter loaded from scratch or play queue shuffled
            playQueueAdapter.playQueue = it.toMutableList()
            playQueueAdapter.notifyDataSetChanged()
        }
    }
})
```

The observer defined above will notify the fragment whenever the underlying data in the PlaybackViewModel view model's currentPlayQueue variable changes. In this way, the fragment can load the play queue and respond to

changes, such as when the play queue is shuffled or items are removed. An if expression is used to determine the appropriate response to observer notifications. If the new play queue is smaller than the play queue loaded into the adapter, then this means one or more queue items have been removed. In which case, the code subtracts the new list of Pair objects from the existing list. Any remaining Pair objects are no longer in the play queue and so are removed from the adapter. Also, the adapter's notifyItemRemoved method is used to remove the queue item from the RecyclerView.

In contrast, if the observer reports a play queue that is the same size or larger than the play queue in the adapter, then this likely means the fragment is loading for the first time and the adapter is currently empty or the play queue has been shuffled. In either case, the entire play queue must be reloaded from scratch. Also, the adapter's notifyDataSetChanged method is used to refresh the entire RecyclerView. You may be wondering what happens in the event items are added to the play queue. While the app does allow the user to add items to the play queue, it is not possible to perform this action from the play queue fragment. So if the user adds songs to the play queue elsewhere in the app, when they return to the play queue fragment the adapter will need to reload from scratch anyway.

In addition to observing the play queue, the play queue fragment also must monitor the Pair object ID of the currently playing queue item. To register the second observer, add the following code below the play queue observer:

```
playbackViewModel.currentlyPlayingQueueID.observe(viewLifecycleOwner, { position ->
    position?.let { playQueueAdapter.currentlyPlayingSongChanged(it) }
})
```

Whenever the currently playing song queue ID changes, the new queue ID will be sent to an adapter method called currentlyPlayingSongChanged. To define the currentlyPlayingSongChanged method, add the following code to the **PlayQueueAdapter.kt** file below the getItemCount method:

```
fun currentlyPlayingSongChanged(newQueueID: Int) {
    val oldCurrentlyPlayingIndex = playQueue.indexOfFirst {
        it.first == currentlyPlayingQueueID
    }

    currentlyPlayingQueueID = newQueueID
    if (oldCurrentlyPlayingIndex != -1) notifyItemChanged(oldCurrentlyPlayingIndex)

    val newCurrentlyPlayingIndex = playQueue.indexOfFirst {
        it.first == currentlyPlayingQueueID
    }
    notifyItemChanged(newCurrentlyPlayingIndex)
}
```

The currentlyPlayingSongChanged method finds the indices of the previous (if applicable) and new currently playing queue items. Next, it uses the adapter's notifyItemChanged method to refresh both items. As covered in the onBindViewHolder method, the text for the currently playing queue item will appear a different colour to distinguish the item from the rest of the play queue.

The final thing we will configure in this section is for the RecyclerView to automatically scroll to the currently playing queue item when the play queue fragment opens. To implement this feature, return to the PlayQueueFragment class and add the following code below the onCreateView method:

```
override fun onResume() {
    super.onResume()

    // This code finds the position in the recycler view list of the currently playing song, and scrolls to it
    val currentlyPlayingQueueIndex = playQueueAdapter.playQueue.indexOfFirst {queueItem ->
        queueItem.first == playQueueAdapter.currentlyPlayingQueueID
    }

    if (currentlyPlayingQueueIndex != -1) (binding.root.layoutManager as
    LinearLayoutManager).scrollToPositionWithOffset(currentlyPlayingQueueIndex, 0)
}
```

The above code references the onResume stage of the fragment lifecycle. The onResume stage will run whenever the fragment becomes visible to the user, either when it is launched for the first time or if the user returns to the app after leaving the fragment open. In this case, we instruct the onResume method to find the index of the currently

playing song in the play queue. The RecyclerView's layout manager's `scrollToPositionWithOffset` method is then used to scroll to the currently playing queue item's position. If the play queue is empty or the currently playing queue index cannot be found, then the value of the `currentlyPlayingQueueIndex` variable will equal -1 and no scroll event will occur.

## Reordering items in the play queue

The user will be able to drag and reorder play queue items. To enable this functionality, open the **PlayQueueFragment.kt** file (**Project > app > java > name of the project > ui > playQueue**) and add the following variable to the top of the class:

```
private val itemTouchHelper by lazy {
    val simpleItemTouchCallback =
        object : ItemTouchHelper.SimpleCallback(UP or DOWN, 0) {
            override fun onSelectedChanged(viewHolder: RecyclerView.ViewHolder?, actionState: Int) {
                super.onSelectedChanged(viewHolder, actionState)

                if (actionState == ACTION_STATE_DRAG) viewHolder?.itemView?.alpha = 0.5f
            }

            override fun clearView(recyclerView: RecyclerView, viewHolder: RecyclerView.ViewHolder) {
                super.clearView(recyclerView, viewHolder)

                viewHolder.itemView.alpha = 1.0f
                playbackViewModel.currentPlayQueue.value = playQueueAdapter.playQueue
            }

            override fun onMove(recyclerView: RecyclerView, viewHolder: RecyclerView.ViewHolder, target:
RecyclerView.ViewHolder): Boolean {
                val from = viewHolder.layoutPosition
                val to = target.layoutPosition
                if (from != to) {
                    val song = playQueueAdapter.playQueue[from]
                    playQueueAdapter.playQueue.removeAt(from)
                    playQueueAdapter.playQueue.add(to, song)
                    playQueueAdapter.notifyItemMoved(from, to)
                }

                return true
            }

            override fun onSwiped(viewHolder: RecyclerView.ViewHolder, direction: Int) {}
        }
    ItemTouchHelper(simpleItemTouchCallback)
}
```

Note you may need to add the following import statement to the top of the file:

```
import androidx.recyclerview.widget.ItemTouchHelper.*
```

The above variable defines an `ItemTouchHelper.SimpleCallback` object that will handle user gesture interactions with items in the RecyclerView. In the `SimpleCallback` object's primary constructor, you can define the drag directions and swipe directions that the item touch helper should respond to. In this case, the accepted drag directions are UP and DOWN, which will allow the user to move play queue items up and down. If you wanted the user to be able to drag items left and right as well then you could also add the drag directions LEFT and RIGHT. You can also specify swipe directions. In this case, the swipe directions parameter is set to 0, which disables the swipe feature.

The `SimpleCallback` object contains several callback methods that respond to user interactions. The first method is called `onSelectedChanged` and defines what happens when a RecyclerView item is selected. In this case, the method makes the selected item 50% transparent by altering the item's alpha property. Once the item is released, the `clearView` method restores the selected item to full opacity. The `clearView` method also sends the reordered play queue to the `PlaybackViewModel` class. Next, the `onMove` method tracks the movement of the RecyclerView items in real-time. It determines the position the item moved from and to and updates the play queue accordingly. After

each movement, the adapter's `notifyItemMoved` method is called to update the `RecyclerView` and show the user the reordered play queue.

To attach the item touch helper to the `RecyclerView`, add the following line of code to the bottom of the `onViewCreated` method:

```
itemTouchHelper.attachToRecyclerView(binding.root)
```

Next, to direct the item touch helper to respond to item drags, add the following method below the `onResume` method:

```
fun startDragging(viewHolder: RecyclerView.ViewHolder) = itemTouchHelper.startDrag(viewHolder)
```

To run the `startDragging` method whenever the user touches the handle `ImageView` widget for a given `RecyclerView` item, return to the `PlayQueueAdapter.kt` file (**Project > app > java > name of the project > ui > playQueue**). Replace the `TODO` comment in the `onTouch` listener that is applied to handle `ImageView` in the `onBindViewHolder` method with the following code:

```
if (event.actionMasked == MotionEvent.ACTION_DOWN) fragment.startDragging(holder)
```

The above code responds to `ACTION_DOWN` touch events. The `ACTION_DOWN` event occurs when the screen is first touched. In other words, as soon as the user presses the handle `ImageView`, the `PlayQueueFragment` class's `startDragging` method will allow the user to drag the song to another position in the play queue.

## Interacting with the play queue

There are several features of the play queue that we have not yet implemented. First, when the user presses the menu `ImageButton` widget for a play queue item, a popup menu should open and allow the user to remove the song from the play queue. To incorporate this feature, create a new menu resource file by right-clicking the `menu` directory (**Project > app > res**) then selecting **New > Menu Resource File**. Name the file `queue_item_menu` then click **OK**. Once the `queue_item_menu.xml` menu resource file opens in the editor, switch the file to `Code` view and add the following item inside the menu element:

```
<item android:id="@+id/remove_item"
    android:title="@string/remove_from_queue" />
```

The above code defines a menu item with an ID of `remove_item` that will display the text "Remove from play queue". To make the popup menu operational, return to the `PlayQueueFragment.kt` file (**Project > app > java > name of the project > ui > playQueue**) and add the following code below the `startDragging` method:

```
fun showPopup(view: View, index: Int) {
    PopupMenu(requireContext(), view).apply {
        inflate(R.menu.queue_item_menu)
        setOnMenuItemClickListener {
            if (it.itemId == R.id.remove_item) callingActivity.removeQueueItem(index)
            true
        }
    }
    show()
}
```

Note you may need to add the following import statement to the top of the file:

```
import android.widget.PopupMenu
```

The above code defines a method called `showPopup` menu that accepts two arguments: the `View` that the popup menu should launch from (the menu `ImageButton` widget in this case) and the index of the selected play queue item. Next, the method uses the `PopupMenu` class to inflate the `queue_item_menu` resource file. The `queue_item_menu` menu resource contains an item with an ID of `remove_item`. To define what action should occur when this item is pressed, the method implements an `onMenuItemClick` listener. If the selected menu item has the ID `remove_item` then a `MainActivity` class method called `removeQueueItem` will remove the selected item from the play queue.

Moving on, let's define a couple more methods that help the user interact with the play queue. First, when the user clicks a play queue item, playback should skip to the user's selected item. The code that implements this feature will be located in the `MainActivity` class. Open the `MainActivity.kt` file (**Project > app > java > name of the project**) and add the following code below the `updateSongInfo` method:

```

fun skipToQueueItem(position: Int) {
    currentlyPlayingQueueID = playQueue[position].first
    lifecycleScope.launch {
        updateCurrentlyPlaying()
        play()
    }
}

```

The skipToQueueItem method sets the currently playing queue ID to the queue item ID associated with the user's selection. Next, the updateCurrentlyPlaying and play methods load the song into the media browser service and commence playback.

The next method will handle requests to remove items from the play queue. Add the following code below the skipToQueueItem method:

```

fun removeQueueItem(index: Int) {
    if (playQueue.isNotEmpty() && index != -1) {
        // Check if the currently playing song is being removed from the play queue
        val currentlyPlayingSongRemoved = playQueue[index].first == currentlyPlayingQueueID

        playQueue.removeAt(index)

        if (currentlyPlayingSongRemoved) {
            currentlyPlayingQueueID = when {
                playQueue.isEmpty() -> {
                    val mediaController = MediaControllerCompat.getMediaController(this)
                    mediaController.transportControls.stop()
                    return
                }
                playQueue.size == index -> playQueue[0].first
                else -> playQueue[index].first
            }

            lifecycleScope.launch {
                updateCurrentlyPlaying()
                if (pbState == STATE_PLAYING) play()
            }
        }

        playbackViewModel.currentPlayQueue.value = playQueue
    }
}

```

The removeQueueItem method contains an argument specifying the index of the queue item to be removed. Once the corresponding item is removed from the play queue, the method checks if the removed item is the currently playing queue item. If this is the case, then the currently playing song must change to a song that is still in the play queue. A when block is used to determine the appropriate song to play. If the removal of the currently playing queue item means that the play queue is now empty, then the media browser service's onStop method will end the playback session. Alternatively, if there are further songs in the play queue, then the next available song will become the currently playing queue item. If the removed song was the last item in the play queue then playback will resume from the first song in the play queue. In any case, the amended play queue is sent to the PlaybackViewModel view model and any areas that are observing the view model's currentPlayQueue variable will be notified.

The next method will handle requests to add new songs to the play queue. Add the following code below the removeQueueItem method:

```

private fun playNext(song: Song) {
    val sortedQueue = playQueue.sortedByDescending {
        it.first
    }

    val highestQueueID = if (sortedQueue.isNotEmpty()) sortedQueue[0].first
    else -1

    val queueItem = Pair(highestQueueID + 1, song)
}

```

```
val index = playQueue.indexOfFirst {
    it.first == currentlyPlayingQueueID
}
```

```
playQueue.add(index + 1, queueItem)
```

```
playbackViewModel.currentPlayQueue.value = playQueue
Toast.makeText(this, getString(R.string.added_to_queue, song.title), Toast.LENGTH_SHORT).show()
}
```

The playNext method adds a Song object that is supplied to the method as an argument to the play queue. To add the song to the play queue, we must create a Pair object for it. The Pair object must contain a unique queue item ID. To determine the queue item ID, the method first finds the highest queue item ID in the play queue and adds 1 to the number when generating the new Pair object. If the play queue was empty at the time the song is being added to the play queue then the new queue item ID will be 0.

Next, the method first finds the index of the currently playing queue item and adds the newly created Pair object to the next available position. This means the newly added song will appear next in the playback queue. If the play queue is empty, then the Pair object will be added to the beginning of the play queue list. Finally, the updated play queue is sent to the PlaybackViewModel view model and a toast notification informs the user that the song has been successfully added to the play queue.

Let's now integrate these new methods with the play queue fragment RecyclerView adapter. First, open the **PlayQueueAdapter.kt** file (**Project > app > java > name of the project > ui > playQueue**) and replace the TODO comment in the PlayQueueViewHolder inner class's onClick method with the following code:

```
activity.skipToQueueItem(layoutPosition)
```

Now, whenever the user selects an item in the play queue, the index of that item in the RecyclerView will be sent to MainActivity's skipToQueueItem method. This will prompt MainActivity to skip to the user's selected song and commence playback.

Next, locate the adapter's onBindViewHolder method and replace the TODO comment in the menu button's onClick listener with the following code:

```
fragment.showPopup(it, position)
```

The above code runs the PlayQueueFragment class's showPopup method, which opens a popup options menu. The index of the selected item in the RecyclerView is supplied as an argument when the method is invoked so the play queue item can be removed if the user selects the relevant menu option.

The last task in this section is to configure the MainActivity class to monitor the play queue and currently playing queue ID because this data will be required for other processes. To implement the observers, add the following code to the bottom of MainActivity class's onCreate method:

```
playbackViewModel.currentPlayQueue.observe(this) { queue ->
    queue?.let {
        playQueue = queue.toMutableList()
    }
}
```

```
playbackViewModel.currentlyPlayingQueueID.observe(this) {
    currentlyPlayingQueueID = it
}
```

The above code registers observers on the PlaybackViewModel view model's currentPlayQueue and currentlyPlayingQueueID variables. Whenever either variable is updated, the value of the new variable is used to update MainActivity's playQueue or currentlyPlayingQueueID variables as appropriate.

## Setting up the Search fragment and layout

The music app will allow the user to search for songs in their music library. This functionality will be achieved by querying the Room database for entries that match the user's search term and displaying the output in a dedicated fragment. The search results fragment will require a layout. Create a new layout file in the usual way: right-click the **layout** directory then select **New > Layout Resource File**. Name the file **fragment\_search** then press OK. Once the **fragment\_search.xml** layout opens in the editor, switch to Code view and modify the file so it reads as follows:

```

<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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:paddingVertical="8dp">

    <TextView
        android:id="@+id/noResults"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/no_results"
        android:textSize="14sp"
        android:visibility="gone"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent" />

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recyclerView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:contentDescription="@string/search_results"/>
</androidx.constraintlayout.widget.ConstraintLayout>

```

The root element of the `fragment_search` layout is a `ConstraintLayout` widget. The `ConstraintLayout` coordinates a `RecyclerView` widget, which will display the list of search results, and a `TextView` widget, which will advise that no search results were found. The `TextView` widget is hidden by default because its `visibility` attribute is set to `gone`; however, the widget will become visible whenever the query returns no results.

Moving on, let's build the fragment which will handle the search results. Create a new Kotlin class by right-clicking the `search ui` directory (**Project** > **app** > **java** > **name of the project** > **ui**) then selecting **New** > **Kotlin Class/File**. Name the file `SearchFragment` and select `Class` from the list of options. Once the `SearchFragment.kt` file opens in the editor, modify its code so it reads as follows:

```

import androidx.fragment.app.Fragment
import android.widget.SearchView

class SearchFragment : Fragment() {

    private var _binding: FragmentSearchBinding? = null
    private val binding get() = _binding!!
    private var musicDatabase: MusicDatabase? = null
    private var searchView: SearchView? = null
    private lateinit var callingActivity: MainActivity

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        _binding = FragmentSearchBinding.inflate(inflater, container, false)
        setHasOptionsMenu(true)
        callingActivity = activity as MainActivity
        musicDatabase = MusicDatabase.getDatabase(requireContext())

        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        // TODO: Initialise the adapter and apply it to the RecyclerView

        binding.recyclerView.layoutManager = LinearLayoutManager(activity)

```

```
binding.recyclerView.itemAnimator = DefaultItemAnimator()
}
```

```
override fun onDestroyView() {
    super.onDestroyView()
    _binding = null
}
}
```

The SearchFragment class's onCreateView method initialises the fragment\_search layout's binding class so the fragment can interact with the layout's components. An instance of the MusicDatabase class is also established so the search fragment can query the Room database and retrieve search results. When the search fragment opens, it should expand the search icon in the app toolbar so the user can type their query. To enable this, add the following code below the onCreateView method:

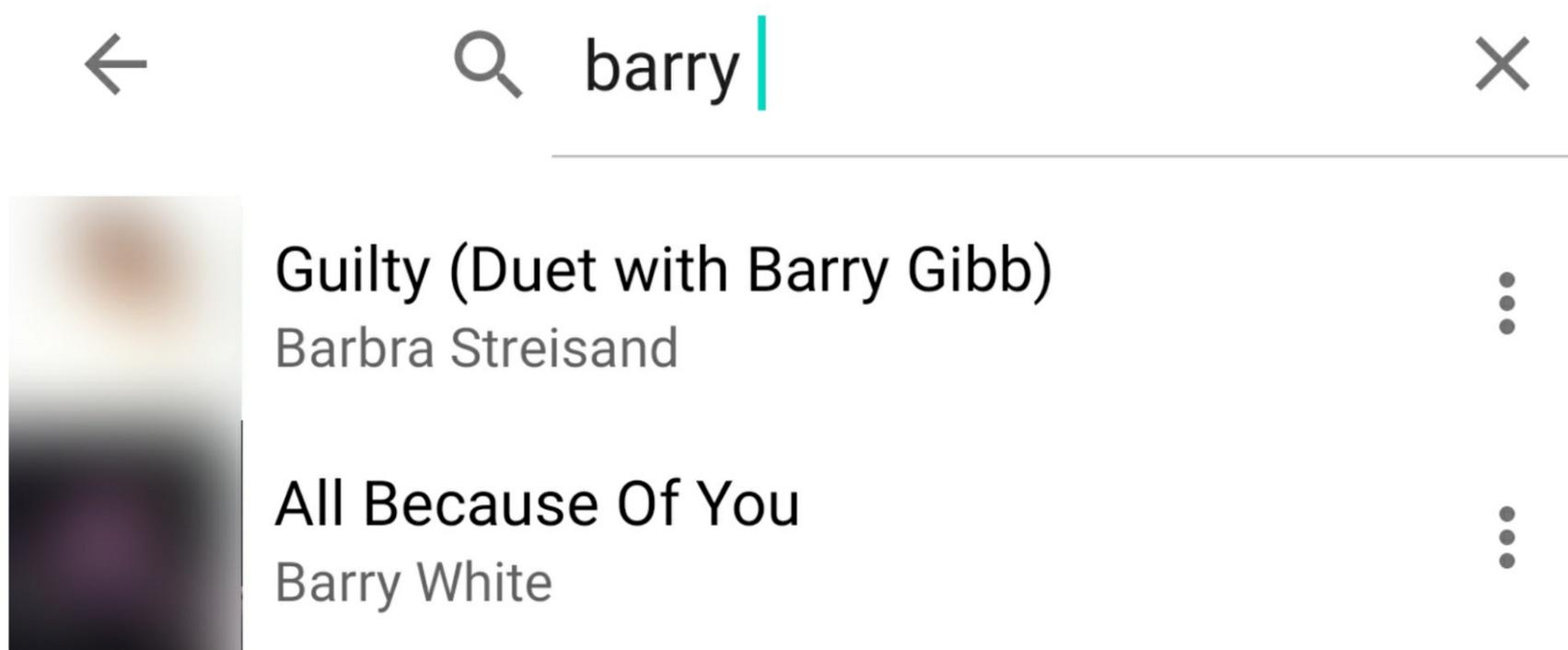
```
override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    val searchItem = menu.findItem(R.id.search)
    searchView = searchItem.actionView as SearchView

    val onQueryListener = object : SearchView.OnQueryTextListener {
        override fun onQueryTextChange(newText: String): Boolean {
            search("%$newText%")
            return true
        }
    }
    override fun onQueryTextSubmit(query: String): Boolean = true
}

searchView?.apply {
    isIconifiedByDefault = false
    queryHint = getString(R.string.search_hint)
    setOnQueryTextListener(onQueryListener)
}

super.onCreateOptionsMenu(menu, inflater)
}
```

The onCreateOptionsMenu method coordinates the menu items in the app toolbar. In the above code, the onCreateOptionsMenu method assigns the search icon menu item to a variable called searchView. The search view's isIconifiedByDefault attribute is set to false, which expands the search box ready for the user to enter their query. The search box will feature a query hint of "Search music" to indicate that the user can query their music library. Finally, an onQueryTextChange listener is attached to the search box to detect when the user is typing. Each time the query text changes, even just by a letter, a method called search will query the Room database.



*Album artwork blurred for copyright reasons.*

Percentage symbols are appended before and after the user's query to indicate that we are looking for results that

contain the user's search term at any position in the raw data. This is because the search query will ultimately be executed by the following SQL query in the MusicDao class (**Project > app > java > name of the project**; see the `findBySearchSongs` method):

```
SELECT * FROM music_table WHERE song_title LIKE :search OR song_artist LIKE :search OR song_album LIKE :search
```

The above query searches the database for entries with a song title, artist or album that include the user's search term. If the search term was sent without percentage symbols, then the SQL query would only return results that are an exact match. For example, if the user is looking for a song called "Somewhere Over The Rainbow" then they would have to type the full song name to return any results. Enclosing the search term in percentage symbols directs the database to return results that contain the term anywhere within the raw data. For example, if the user typed "over", then the search term used in the database query would be "%over%". The resultant query would return the song "Somewhere Over The Rainbow" even though it is just a partial match.

## Displaying the search results

In this section, we'll design an adapter that will load the search results into the `fragment_search` layout's RecyclerView widget and handle user interactions. Right-click the **search** directory then select **New > Kotlin Class/File**. Name the file `SearchAdapter` and select `Class` from the list of options. Once the `SearchAdapter.kt` file opens in the editor, edit its code so it reads as follows:

```
class SearchAdapter(private val activity: MainActivity):
RecyclerView.Adapter<SearchAdapter.SongsViewHolder>() {
    var songs = mutableListOf<Song>()

    inner class SongsViewHolder(itemView: View) :
        RecyclerView.ViewHolder(itemView),
        View.OnClickListener {

        internal var mArtwork = itemView.findViewById<View>(R.id.artwork) as ImageView
        internal var mTitle = itemView.findViewById<View>(R.id.title) as TextView
        internal var mArtist = itemView.findViewById<View>(R.id.artist) as TextView
        internal var mMenu = itemView.findViewById<ImageButton>(R.id.menu)

        init {
            itemView.isClickable = true
            itemView.setOnClickListener(this)
            itemView.setOnLongClickListener {
                activity.showSongPopup(it, songs[layoutPosition])
                return@setOnLongClickListener true
            }
        }

        override fun onClick(view: View) {
            activity.playNewSongs(listOf(songs[layoutPosition]), 0, false)
        }

        override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): SongsViewHolder {
            return SongsViewHolder(LayoutInflater.from(parent.context).inflate(R.layout.song_preview, parent, false))
        }

        override fun onBindViewHolder(holder: SongsViewHolder, position: Int) {
            val current = songs[position]

            activity.insertArtwork(current.albumID, holder.mArtwork)

            holder.mTitle.text = current.title
            holder.mArtist.text = current.artist
            holder.mMenu.setOnClickListener {
                activity.showSongPopup(it, current)
            }
        }
    }
}
```

```
override fun getItemCount() = songs.size
```

The above code begins by declaring a variable called `activity` in the `SearchAdapter` class's primary constructor. The `activity` variable will hold a reference to the `MainActivity` class and allow the adapter to use the `MainActivity` class's public methods and data. In the body of the adapter, a variable called `songs` will store the list of `Song` objects that were returned by the user's search query. Next, an inner class called `SongsViewHolder` is established. This inner class will initialise the components of the `song_preview.xml` layout that was created earlier for the songs fragment and adapter. The adapter knows to use the `song_preview.xml` layout because this is the layout that is inflated by the `onCreateViewHolder` method. If the user clicks an item in the `RecyclerView`, then a list containing the selected `Song` object is sent to `MainActivity`'s `playNewSongs` method to initiate playback. Meanwhile, if the user long clicks an item (presses it for several seconds) then `MainActivity`'s `showSongPopup` method will load a popup menu and allow the user to perform actions such as adding the song to the play queue.

The adapter also contains a method called `onBindViewHolder`, which populates the data at each position in the `RecyclerView`. In this case, the `onBindViewHolder` method uses the information from the `Song` object associated with a given position in the `RecyclerView` to load album artwork into the artwork `ImageView` widget and load the song's title and artist name into the two `TextView` widgets. The artwork image is loaded by a `MainActivity` method called `insertArtwork` that we defined earlier. The last method in the adapter is called `getItemCount` and it is used to determine how many items are loaded into the `RecyclerView`. In this case, the number of items will equal the size of the list of `Song` objects that were returned in the search results.

Moving on, let's now define the code that runs the user's search query and loads the results into the `SearchAdapter` class. Return to the `SearchFragment.kt` file (**Project > app > java > name of the project > ui > search**) and add the following variable to the list of variables at the top of the class:

```
private lateinit var searchAdapter: SearchAdapter
```

The `searchAdapter` variable will provide access to the `SearchAdapter` class. To initialise the variable and apply it to the `RecyclerView`, replace the `TODO` comment in the `onViewCreated` method with the following code:

```
searchAdapter = SearchAdapter(callingActivity)
binding.recyclerView.adapter = searchAdapter
```

Once the adapter has been initialised and attached to the `RecyclerView` widget, we can define the method that executes the user's search query and loads the results. To do this, add the following code below the `onCreateOptionsMenu` method:

```
private fun search(query: String) = lifecycleScope.launch(Dispatchers.IO) {
    val songs = musicDatabase!!.musicDao().findBySearchSongs(query).take(10)
```

```
    lifecycleScope.launch(Dispatchers.Main) {
        val adapterSongs = searchAdapter.songs
```

```
        when {
```

```
            songs.isEmpty() -> {
```

```
                val previousSongCount = searchAdapter.songs.size
```

```
                searchAdapter.songs = mutableListOf()
```

```
                searchAdapter.notifyItemRangeRemoved(0, previousSongCount)
```

```
                binding.noResults.visibility = View.VISIBLE
```

```
            }
```

```
            adapterSongs.isEmpty() -> {
```

```
                binding.noResults.visibility = View.GONE
```

```
                searchAdapter.songs = songs.toMutableList()
```

```
                searchAdapter.notifyItemRangeInserted(0, songs.size)
```

```
            }
```

```
            else -> {
```

```
                binding.noResults.visibility = View.GONE
```

```
                val removeItems = adapterSongs - songs.toSet()
```

```
                val addItem = songs - adapterSongs.toSet()
```

```
                for (s in removeItems) {
```

```
                    val index = searchAdapter.songs.indexOfFirst {
```

```
                        it.songID == s.songID
```

```
                    }
```



```
return super.onCreateOptionsMenu(menu)
```

The above code inflates the contents of the **main.xml** menu resource file and assigns an `onSearchClick` listener to the `SearchView` item. When the user clicks the search icon, the `NavController` class will navigate to the search fragment, which can be found under the ID `nav_search` in the **mobile\_navigation.xml** navigation graph.

There are a couple of further measures we need to consider about the search fragment. First, when the user navigates to the search fragment the `SearchView` will be expanded by setting its `isIconified` property to `false`. This is necessary to expand the search box but when the user leaves the search fragment it is important to set the `SearchView`'s `isIconified` property back to `true`. One way the user can leave the search fragment is by pressing the back button at the bottom of the device.



To check whether the `SearchView` is expanded when the back button is pressed and reset the `SearchView`'s `isIconified` property back to `true` if necessary, add the following code below the `onCreateOptionsMenu` method:

```
override fun onBackPressed() {  
    // TODO: Handle closure of the currently playing fragment  
  
    if (!searchView.isIconified) {  
        searchView.isIconified = true  
        searchView.onActionViewCollapsed()  
    }  
}
```

The above code collapses and iconifies the `SearchView` whenever the back button at the bottom of the device window is pressed; however, there is an additional back button we must consider. The app toolbar will also contain a back button in the top left corner that is referred to as the home as up button. If the user presses this button and the user is in the search fragment then we will need to collapse and iconify the `SearchView` as described above. To implement this, locate the `onSupportNavigateUp` method and modify its code so it reads as follows:

```
override fun onSupportNavigateUp(): Boolean {  
    val navController = findNavController(R.id.nav_host_fragment)  
    if (!searchView.isIconified) {  
        searchView.isIconified = true  
        searchView.onActionViewCollapsed()  
    }  
    return navController.navigateUp(appBarConfiguration) || super.onSupportNavigateUp()  
}
```

Now, if the user presses a back button while the `SearchView` is expanded then it will be collapsed and iconified as the user exits the search fragment.

In addition to collapsing the `SearchView` when the user exits the search fragment, it is also worth considering that the device's keyboard may still be open from the user typing their query. To address this, we must implement a method that closes the keyboard when it is no longer required. Add the following code below the `playNext` method:

```
fun hideKeyboard(activity: Activity) {  
    val inputManager = activity.getSystemService(Context.INPUT_METHOD_SERVICE) as InputMethodManager  
    val currentFocusedView = activity.currentFocus  
    if (currentFocusedView != null) inputManager.hideSoftInputFromWindow(  
        currentFocusedView.windowToken,  
        InputMethodManager.HIDE_NOT_ALWAYS  
    )  
}
```

The above code uses the `hideSoftInputFromWindow` method to remove the soft input keyboard from whatever app component is currently in focus. We will need to call the `hideKeyboard` method when the `SearchFragment` fragment enters the `onStop` stage of its lifecycle, which occurs when the fragment is no longer active. To implement this, open the **SearchFragment.kt** file (**Project > app > java > name of the project > ui > search**) and add the

following code below the onCreateOptionsMenu method:

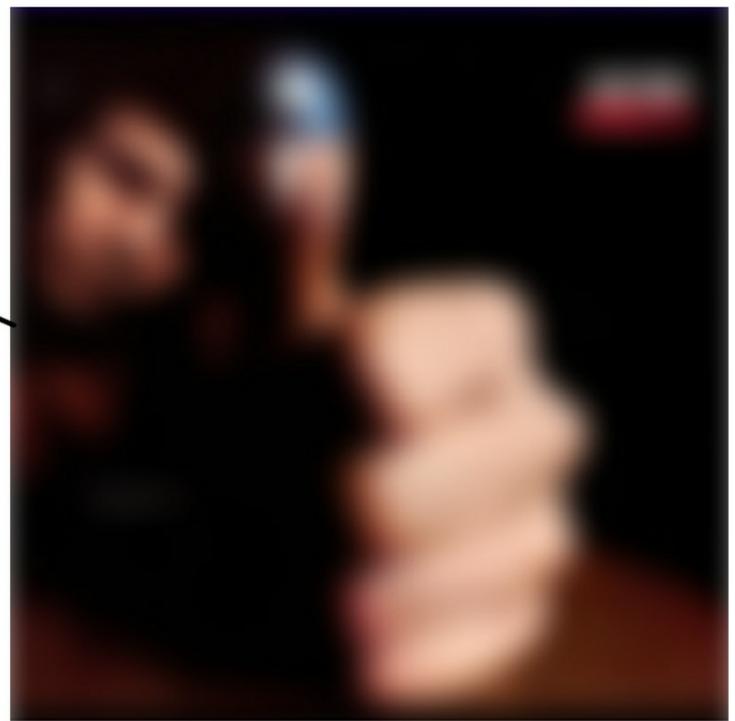
```
override fun onStop() {  
    super.onStop()  
    callingActivity.hideKeyboard(callingActivity)  
}
```

The device's keyboard will now automatically be hidden from view whenever the user navigates away from the search fragment.

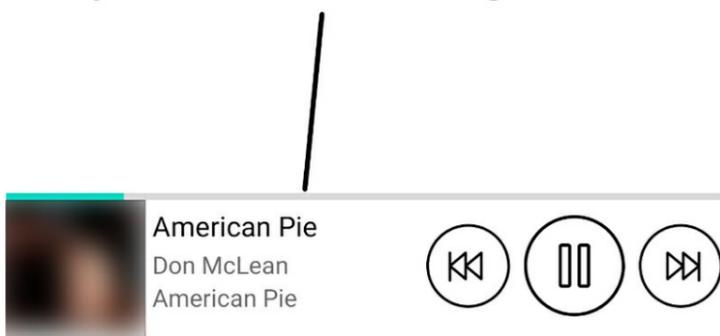
## The playback controls navigation graph

The songs, play queue and search fragments we have created so far are all components of the main **mobile\_navigation.xml** navigation graph. The user can navigate from one destination to another but no two destinations can be active at the same time. In this section, we will design a second navigation graph that will be active on the screen simultaneously with the **mobile\_navigation** navigation graph. The second navigation graph will contain the playback controls at the bottom of the screen and enable the user to navigate to a fragment where they can view the details of the currently playing song.

Currently Playing Fragment



Playback Controls Fragment



*Album artwork blurred for copyright reasons.*



To create a new navigation graph, right-click the **navigation** folder (**Project > app > res > navigation**) then select **New > Navigation Resource File**. Name the file **controls\_navigation** then press OK. Once the **controls\_navigation.xml** file opens in the editor, switch to Code view and modify the file so it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>  
<navigation 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"  
    app:startDestination="@+id/nav_controls">  
  
<fragment  
    android:id="@+id/nav_controls"
```

```
android:name="com.example.music.ui.controls.ControlsFragment"  
android:label=""  
tools:layout="@layout/fragment_controls" />
```

```
<fragment  
  android:id="@+id/nav_currently_playing"  
  android:name="com.example.music.ui.currentlyPlaying.CurrentlyPlayingFragment"  
  android:label=""  
  tools:layout="@layout/fragment_currently_playing" />  
</navigation>
```

The above navigation graph contains two destinations: a fragment called ControlsFragment that will contain the playback controls, and a fragment called CurrentlyPlayingFragment that will occupy the full screen and display the details of the currently playing song. In the upcoming sections, we will design the fragments and their associated layouts.

## Setting up the PlaybackControls fragment and layout

The playback controls will allow the user to pause and resume playback, skip tracks and more. The fragment containing the buttons will also display information about the currently playing song and enable the user to navigate to a dedicated currently playing song fragment that will occupy the full screen. The playback controls fragment will require a layout. To create a new layout file, right-click the **layout** directory then select **New > Layout Resource File**. Name the file fragment\_controls then press OK. Once the **fragment\_controls.xml** layout opens in the editor, switch to Code view and modify the file so it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>  
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"  
  android:layout_height="wrap_content"  
  android:layout_width="match_parent"  
  android:background="?attr/selectableItemBackground"  
  android:layout_gravity="bottom">
```

```
<ProgressBar  
  android:id="@+id/songProgressBar"  
  android:layout_width="match_parent"  
  android:layout_height="4dp"  
  style="?android:attr/progressBarStyleHorizontal" />
```

```
<ImageView  
  android:id="@+id/artwork"  
  android:layout_width="80dp"  
  android:layout_height="80dp"  
  android:contentDescription="@string/set_album_artwork"  
  android:transitionName="@string/transition_image"  
  android:layout_below="@id/songProgressBar"  
  android:layout_alignParentStart="true" />
```

```
<LinearLayout  
  android:layout_width="0dp"  
  android:layout_height="wrap_content"  
  android:padding="4dp"  
  android:orientation="vertical"  
  android:layout_alignTop="@id/artwork"  
  android:layout_alignBottom="@id/artwork"  
  android:layout_toEndOf="@id/artwork"  
  android:layout_toStartOf="@id/btnBackward" >
```

```
<TextView  
  android:id="@+id/title"  
  android:layout_width="wrap_content"  
  android:layout_height="wrap_content"  
  android:transitionName="@string/transition_title"  
  android:layout_marginBottom="2dp"
```

```
    android:singleLine="true"  
    android:textSize="16sp"  
    android:textColor="?attr/colorOnSurface" />
```

```
<TextView  
    android:id="@+id/artist"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:transitionName="@string/transition_subtitle"  
    android:singleLine="true"  
    android:textSize="14sp"  
    android:textColor="@color/material_on_surface_emphasis_medium" />
```

```
<TextView  
    android:id="@+id/album"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:transitionName="@string/transition_subtitle2"  
    android:singleLine="true"  
    android:textSize="14sp"  
    android:textColor="@color/material_on_surface_emphasis_medium" />
```

```
</LinearLayout>
```

```
<ImageButton  
    android:id="@+id/btnBackward"  
    android:layout_width="48dp"  
    android:layout_height="48dp"  
    android:src="@drawable/ic_back"  
    android:transitionName="@string/transition_back"  
    android:contentDescription="@string/skip_back"  
    android:layout_centerVertical="true"  
    android:layout_toStartOf="@id/btnPlay"  
    style="@style/Widget.Custom.Button" />
```

```
<ImageButton  
    android:id="@+id/btnPlay"  
    android:layout_width="58dp"  
    android:layout_height="58dp"  
    android:layout_marginHorizontal="8dp"  
    android:src="@drawable/ic_play"  
    android:transitionName="@string/transition_play"  
    android:contentDescription="@string/play_or_pause_current_track"  
    android:layout_centerVertical="true"  
    android:layout_toStartOf="@id/btnForward"  
    style="@style/Widget.Custom.Button" />
```

```
<ImageButton  
    android:id="@+id/btnForward"  
    android:layout_width="48dp"  
    android:layout_height="48dp"  
    android:src="@drawable/ic_next"  
    android:transitionName="@string/transition_forward"  
    android:contentDescription="@string/skip_ahead"  
    android:layout_centerVertical="true"  
    android:layout_alignParentEnd="true"  
    style="@style/Widget.Custom.Button" />
```

```
</RelativeLayout>
```

The root element of the `fragment_controls` layout is a `RelativeLayout` widget, which will coordinate the majority of the layout's widgets. The first widget is a `ProgressBar`, which will show the progress of playback through the currently playing song. The layout also contains an `ImageView` widget, which will display the artwork of the currently playing song; several `TextView` widgets that will display the song's title, artist name and album name, respectively; and multiple `ImageButton` widgets that will comprise the playback controls. The three `TextView`

widgets will be stacked vertically inside a `LinearLayout` widget that is positioned centrally between the artwork `ImageView` widget and the playback control buttons.

You may notice several of the widgets include a `transitionName` attribute. The `transitionName` attributes will allow us to apply a transition animation when the user navigates between the playback controls and currently playing fragments. In brief, both fragments will share corresponding widgets with matching transition names. For example, both fragment layouts will contain a `TextView` widget displaying the title of the song and a `transitionName` attribute set to `transition_title`. The contents of the complementary widgets will merge when the user navigates from one fragment to another and this will help create a neat transition animation.

To make the playback controls fragment operational, create a new Kotlin class by right-clicking the **controls ui** directory (**Project > app > java > name of the project > ui**) then selecting **New > Kotlin Class/File**. Name the file `ControlsFragment` and select `Class` from the list of options. Once the `ControlsFragment.kt` file opens in the editor, modify its code so it reads as follows:

```
import androidx.fragment.app.Fragment
import androidx.fragment.app.activityViewModels

class ControlsFragment : Fragment() {

    private var _binding: FragmentControlsBinding? = null
    private val binding get() = _binding!!
    private var fastForwarding = false
    private var fastRewinding = false
    private val playbackViewModel: PlaybackViewModel by activityViewModels()
    private lateinit var callingActivity: MainActivity

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        _binding = FragmentControlsBinding.inflate(inflater, container, false)
        callingActivity = activity as MainActivity
        return binding.root
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }

    override fun onResume() {
        super.onResume()

        binding.songProgressBar.max = playbackViewModel.currentPlaybackDuration.value ?: 0
        binding.songProgressBar.progress = playbackViewModel.currentPlaybackPosition.value ?: 0
    }
}
```

The `ControlsFragment` class contains several variables including a binding variable that provides access to the `fragment_controls` layout via its binding class, variables that provide access to the `PlaybackViewModel` and `MainActivity` classes and their data, and two variables called `fastForwarding` and `fastRewinding` that contain boolean values currently set to `false`. The `fastForwarding` and `fastRewinding` variables will record whether the user is actively fast forwarding or rewinding the current song.

Next, the `onCreateView` method initialises the binding and `callingActivity` variables. The `onDestroyView` method later sets the binding variable back to `null` when the fragment is in the process of shutting down and the `fragment_controls` layout is no longer accessible. Finally, the `onResume` method, which runs whenever the user launches or returns to the fragment, loads the current playback position and duration of the currently playing song into the `fragment_controls` layout's `ProgressBar` widget. The `ProgressBar` will use this data to update its progress indicator.

Moving on, we'll now write the code that populates the `fragment_controls` layout's widgets with the details of the currently playing song. Add the following code below the `onCreateView` method:

```

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    playbackViewModel.currentlyPlayingSong.observe(viewLifecycleOwner, {
        if (it != null) {
            binding.title.text = it.title
            binding.artist.text = it.artist
            binding.album.text = it.album
            callingActivity.insertArtwork(it.albumID, binding.artwork)
            binding.root.setOnClickListener {
                val extras = FragmentNavigatorExtras(
                    binding.artwork to binding.artwork.transitionName,
                    binding.title to binding.title.transitionName,
                    binding.album to binding.album.transitionName,
                    binding.artist to binding.artist.transitionName,
                    binding.btnPlay to binding.btnPlay.transitionName,
                    binding.btnBackward to binding.btnBackward.transitionName,
                    binding.btnForward to binding.btnForward.transitionName
                )
                findNavController().navigate(R.id.nav_currently_playing, null, null, extras)
            }
        } else {
            binding.title.text = null
            binding.artist.text = null
            binding.album.text = null
            Glide.with(callingActivity)
                .clear(binding.artwork)
            binding.root.setOnClickListener(null)
        }
    })
}

```

```

playbackViewModel.isPlaying.observe(viewLifecycleOwner, { isPlaying ->
    isPlaying?.let {
        if (it) binding.btnPlay.setImageResource(R.drawable.ic_pause)
        else binding.btnPlay.setImageResource(R.drawable.ic_play)
    }
})

```

```

playbackViewModel.currentPlaybackPosition.observe(viewLifecycleOwner, { position ->
    position?.let {
        binding.songProgressBar.progress = position
    }
})

```

```

// keep track of currently playing song duration
playbackViewModel.currentPlaybackDuration.observe(viewLifecycleOwner, { duration ->
    duration?.let {
        binding.songProgressBar.max = it
    }
})

```

```

// TODO: Respond to the playback controls here
}

```

Note you may need to add the following import statements to the top of the file:

```

import androidx.navigation.fragment.FragmentNavigatorExtras
import androidx.navigation.fragment.findNavController

```

The `onViewCreated` method registers an observer on the `PlaybackViewModel` view model's `currentlyPlayingSong` variable, which contains the `Song` object associated with the currently playing song. The information from this `Song` object is then used to populate the artwork `ImageView` widget and song details `TextView` widgets with the metadata for that song. In addition, we set an `onClick` listener to the `RelativeLayout` root element. Whenever the user clicks any part of the layout (except the playback control buttons) they will trigger a navigation event towards the

CurrentlyPlayingFragment fragment as defined in the **controls\_navigation.xml** navigation graph. The navigation event contains an extras bundle detailing all the shared elements between the user's current location and their navigational destination. In this instance, the shared elements include every widget from the fragment\_controls layout that contains a transitionName attribute because they will be used for the transition animation.

The currentlyPlayingSong observer also contains an else block that runs if the currently playing song is null/empty. The else block clears all the currently loaded song metadata. There are also some other observers in the fragment. For example, there is an observer on the view model's isPlaying variable, which contains a boolean value indicating whether or not a song is currently playing. The observer alternates the image in the play/pause ImageButton widget between a play symbol and pause symbol as appropriate. The next two observers monitor the view model's currentPlaybackPosition and currentPlaybackDuration variables, respectively. The playback position observer updates the ProgressBar widget's progress indicator during playback. Meanwhile, the playback duration variable sets the maximum value for the ProgressBar widget to the length of the song, which helps calibrate the playback position progress updates.

## Responding to the playback controls

Continuing with the ControlsFragment class, we will now write the code that responds to user interactions with the playback controls buttons. First, locate the onCreateView method and replace the TODO comment with the following code:

```
// play/pause btn actions
binding.btnPlay.setOnClickListener {
    callingActivity.playPauseControl()
}
```

```
binding.btnBackward.setOnClickListener{
    if (fastRewinding) fastRewinding = false
    else callingActivity.skipBack()
}
```

```
binding.btnBackward.setOnLongClickListener {
    fastRewinding = true
    lifecycleScope.launch {
        do {
            callingActivity.fastRewind()
            delay(500)
        } while (fastRewinding)
    }
    return@setOnLongClickListener false
}
```

```
binding.btnForward.setOnClickListener{
    if (fastForwarding) fastForwarding = false
    else callingActivity.skipForward()
}
```

```
binding.btnForward.setOnLongClickListener {
    fastForwarding = true
    lifecycleScope.launch {
        do {
            callingActivity.fastForward()
            delay(500)
        } while (fastForwarding)
    }
    return@setOnLongClickListener false
}
```

Note you may need to add the following import statements to the top of the file:

```
import androidx.lifecycle.lifecycleScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch
```

The above code registers `onClick` and `onLongClick` listeners to the `ImageButton` widgets that comprise the playback controls. The play button will pause and unpause the currently playing song when clicked. Meanwhile, the skip back and skip forward buttons will skip through the songs in the play queue. The skip back and skip forward buttons also contain `onLongClick` listeners that will fast forward or rewind playback. The fast forward and fast rewind actions are wrapped in a `do/while` block which means the actions will repeat for as long as the `fastForwarding` (or `fastRewinding`) variable is true. Furthermore, there will be a 500 ms interval between repetitions, as specified using the `delay` method. The `delay` method blocks the current worker thread until the time has elapsed. For this reason, it is essential to wrap the task that is being delayed in a coroutine scope so you can delegate the task to an alternative worker thread.

We'll discuss the mechanics of the `fastForward` and `fastRewind` methods shortly. In brief, both methods will alter the playback position by 5000 ms. In other words, for every 500 ms that the skip forward button is pressed, the playback progress will jump forward by 5000 ms. This equates to a 10x increase in playback speed. The same rule applies when the skip backward button is pressed, except playback will move in reverse.

An important consideration with the fast forward and rewind buttons is that playback must return to normal once the user releases the button. By default, the `do/while` loop would continue regardless and continue altering the playback position. For this reason, the `onLongClick` listeners return a value of `false`, which signals that the `onLongClick` listener alone cannot completely handle the click event. This means once the user releases the button and the `onLongClick` event ends, the regular `onClick` event will run. The `onClick` listeners use the boolean values stored in the `fastForwarding` and `fastRewinding` variables to determine whether the user is actively fast forwarding or rewinding the current song. If this is the case, then the `onClick` listeners set the `fastForwarding` and `fastRewinding` variables back to `false`, which terminates the `do/while` loop that was powering the fast forward or rewind feature, thereby returning playback to its regular speed.

The playback control actions will be relayed to the media browser service via dedicated methods in the `MainActivity` class. To respond to the play/pause button, open the `MainActivity.kt` file (**Project > app > java > name of the project**) and add the following method below the `hideKeyboard` method:

```
fun playPauseControl() {
    when (pbState) {
        STATE_PAUSED -> play()
        STATE_PLAYING -> mediaController.transportControls.pause()
    } else -> {
        // Play the first song in the library if the play queue is currently empty
        if (playQueue.isNullOrEmpty()) playNewSongs(completeLibrary, 0, false)
    } else {
        // It's possible a queue has been built without ever pressing play
        lifecycleScope.launch {
            updateCurrentlyPlaying()
            play()
        }
    }
}
```

The `playPauseControl` method uses a `when` block to respond to the different playback states. If a song is currently paused or playing, then the method toggles the playback state as appropriate. On the other hand, if no song is currently paused or playing, then the method initiates playback of the play queue or the entire music library if the play queue is empty.

Next, add the following methods below the `playPauseControl` method to handle requests to skip back or skip forward in the play queue:

```
fun skipBack() = mediaController.transportControls.skipToPrevious()
fun skipForward() = mediaController.transportControls.skipToNext()
```

The `skipBack` and `skipForward` methods use the relevant media browser service methods to change the song. If the user presses the skip back button less than five seconds into the currently playing song, then the current song will restart rather than skipping back a track. For more information on these methods, refer back to the 'The media browser service - part 1' section.

Next, add the following methods below the skipForward method to handle requests to fast forward or rewind the currently playing song.

```
fun fastRewind() {  
    val pos = currentPlaybackPosition - 5000  
    if (pos < 0) skipBack()  
    else mediaController.transportControls.seekTo(pos.toLong())  
}
```

```
fun fastForward() {  
    val pos = currentPlaybackPosition + 5000  
    if (pos > currentPlaybackDuration) skipForward()  
    else mediaController.transportControls.seekTo(pos.toLong())  
}
```

The fastRewind and fastForward methods skip the playback position back 5000 ms and forward 5000 ms, respectively. If either method detects that the user has reached the start or the end of the current song, then it will run the relevant skip back or skip forward method. For the methods to function correctly, they require information about the current playback position and duration of the currently playing song. Both bits of data can be retrieved from the PlaybackViewModel view model. To handle this, add the following variables to the top of the MainActivity class:

```
private var currentPlaybackPosition = 0  
private var currentPlaybackDuration = 0
```

Next, add the following code to the bottom of the onCreate method to register observers on the relevant PlaybackViewModel variables and update the value of the currentPlaybackPosition and currentPlaybackDuration variables accordingly:

```
playbackViewModel.currentPlaybackPosition.observe(this) { position ->  
    position?.let {  
        currentPlaybackPosition = it  
    }  
}
```

```
playbackViewModel.currentPlaybackDuration.observe(this) { duration ->  
    duration?.let {  
        currentPlaybackDuration = it  
    }  
}
```

## Setting up the Currently Playing fragment and layout

In this section, we'll design a fragment that will display information about the currently playing song as well as feature several playback control buttons. The currently playing fragment will require a layout, so right-click the **layout** directory (**Project** > **app** > **res**) and create a new layout called fragment\_currently\_playing. Once the **fragment\_currently\_playing.xml** file opens in the editor add the following code:

```
<?xml version="1.0" encoding="utf-8"?>  
<androidx.constraintlayout.widget.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:paddingBottom="24dp"  
    android:background="?attr/colorSurface">  
  
    <ImageView  
        android:id="@+id/artwork"  
        android:layout_width="match_parent"  
        android:layout_height="0dp"  
        android:clickable="true"  
        android:transitionName="@string/transition_image"  
        android:contentDescription="@string/album_artwork"  
        app:layout_constraintTop_toTopOf="parent"
```

```
app:layout_constraintDimensionRatio="1:1" />
```

```
<ImageButton  
    android:id="@+id/currentClose"  
    android:layout_width="30dp"  
    android:layout_height="30dp"  
    android:layout_marginStart="14dp"  
    android:layout_marginTop="36dp"  
    android:src="@drawable/ic_down"  
    android:translationZ="200dp"  
    android:contentDescription="@string/close_currently_playing"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toTopOf="parent"  
    style="@style/Widget.Custom.Button" />
```

```
<ImageButton  
    android:id="@+id/currentButtonRepeat"  
    android:layout_width="30dp"  
    android:layout_height="30dp"  
    android:layout_marginStart="8dp"  
    android:src="@drawable/ic_repeat"  
    android:contentDescription="@string/repeat_current_playlist"  
    app:layout_constraintTop_toBottomOf="@id/artwork"  
    app:layout_constraintBottom_toTopOf="@id/currentSeekBar"  
    app:layout_constraintStart_toStartOf="parent"  
    style="@style/Widget.Custom.Button" />
```

```
<ImageButton  
    android:id="@+id/currentButtonShuffle"  
    android:layout_width="30dp"  
    android:layout_height="30dp"  
    android:layout_marginEnd="8dp"  
    android:src="@drawable/ic_shuffle"  
    android:contentDescription="@string/shuffle_play_queue"  
    app:layout_constraintTop_toBottomOf="@id/artwork"  
    app:layout_constraintBottom_toTopOf="@id/currentSeekBar"  
    app:layout_constraintEnd_toEndOf="parent"  
    style="@style/Widget.Custom.Button" />
```

```
<TextView  
    android:id="@+id/title"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:transitionName="@string/transition_title"  
    android:singleLine="true"  
    android:textSize="20sp"  
    android:textColor="?attr/colorOnSurface"  
    app:layout_constraintEnd_toStartOf="@id/currentButtonShuffle"  
    app:layout_constraintStart_toEndOf="@id/currentButtonRepeat"  
    app:layout_constraintBottom_toTopOf="@id/artist" />
```

```
<TextView  
    android:id="@+id/artist"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:transitionName="@string/transition_subtitle"  
    android:layout_margin="5dp"  
    android:singleLine="true"  
    android:textSize="18sp"  
    android:textColor="@color/material_on_surface_emphasis_medium"  
    app:layout_constraintEnd_toStartOf="@id/currentButtonShuffle"  
    app:layout_constraintStart_toEndOf="@id/currentButtonRepeat"
```

```
app:layout_constraintTop_toBottomOf="@id/artwork"  
app:layout_constraintBottom_toTopOf="@id/currentSeekBar" />
```

```
<TextView  
    android:id="@+id/album"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:transitionName="@string/transition_subtitle2"  
    android:singleLine="true"  
    android:textSize="18sp"  
    android:textColor="@color/material_on_surface_emphasis_medium"  
    android:layout_centerHorizontal="true"  
    app:layout_constraintEnd_toStartOf="@id/currentButtonShuffle"  
    app:layout_constraintStart_toEndOf="@id/currentButtonRepeat"  
    app:layout_constraintTop_toBottomOf="@id/artist" />
```

```
<SeekBar  
    android:id="@+id/currentSeekBar"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:layout_marginVertical="26dp"  
    app:layout_constraintBottom_toTopOf="@id/btnPlay" />
```

```
<TextView  
    android:id="@+id/currentPosition"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginStart="10dp"  
    android:textSize="14sp"  
    android:textColor="@color/material_on_surface_emphasis_medium"  
    app:layout_constraintTop_toBottomOf="@id/currentSeekBar"  
    app:layout_constraintStart_toStartOf="parent" />
```

```
<TextView  
    android:id="@+id/currentMax"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginEnd="10dp"  
    android:textSize="14sp"  
    android:textColor="@color/material_on_surface_emphasis_medium"  
    app:layout_constraintTop_toBottomOf="@id/currentSeekBar"  
    app:layout_constraintEnd_toEndOf="parent" />
```

```
<ImageButton  
    android:id="@+id/btnBackward"  
    android:layout_width="55dp"  
    android:layout_height="55dp"  
    android:layout_marginHorizontal="12dp"  
    android:src="@drawable/ic_back"  
    android:transitionName="@string/transition_back"  
    android:contentDescription="@string/skip_back"  
    app:layout_constraintTop_toTopOf="@id/btnPlay"  
    app:layout_constraintBottom_toBottomOf="@id/btnPlay"  
    app:layout_constraintEnd_toStartOf="@id/btnPlay"  
    style="@style/Widget.CustomButton" />
```

```
<ImageButton  
    android:id="@+id/btnPlay"  
    android:layout_width="70dp"  
    android:layout_height="70dp"  
    android:layout_marginBottom="12dp"  
    android:src="@drawable/ic_play"  
    android:transitionName="@string/transition_play"
```

```

    android:contentDescription="@string/play_or_pause_current_track"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintBottom_toBottomOf="parent"
    style="@style/Widget.Custom.Button" />

```

```

<ImageButton
    android:id="@+id/btnForward"
    android:layout_width="55dp"
    android:layout_height="55dp"
    android:layout_marginHorizontal="12dp"
    android:src="@drawable/ic_next"
    android:transitionName="@string/transition_forward"
    android:contentDescription="@string/skip_ahead"
    app:layout_constraintTop_toTopOf="@id/btnPlay"
    app:layout_constraintBottom_toBottomOf="@id/btnPlay"
    app:layout_constraintStart_toEndOf="@id/btnPlay"
    style="@style/Widget.Custom.Button" />
</androidx.constraintlayout.widget.ConstraintLayout>

```

At the top of the `fragment_currently_playing` layout, an `ImageView` widget will display the currently playing song's album artwork. The `ImageView` will occupy the maximum possible width. Its height attribute is set to `0dp` which means the height will be the maximum possible value afforded by its constraints. In this case, the `ImageView` has a `constraintDimensionRatio` of `1:1`, which means the height of the `ImageView` will equal its width. In this way, we ensure the artwork appears as a perfect square, regardless of the screen size. In the top left corner of `ImageView`, there will be a small down arrow `ImageButton` widget. This `ImageButton` will allow the user to close the currently playing fragment when clicked. To ensure the `ImageButton` is not obscured by the artwork `ImageView`, the `ImageButton` features a `translationZ` attribute set to a value high enough to elevate it above the `ImageView`.

Below the artwork `ImageView`, several `TextView` widgets will display information about the currently playing song (e.g. its title, artist and album). There is also an `ImageButton` widget on either side of the `TextView` widgets that will allow the user to toggle the repeat mode and toggle the shuffle mode, respectively. The next widget is a `SeekBar`. `SeekBar` widgets behave similar to `ProgressBar` widgets but also contain a draggable thumb. In this case, the `SeekBar` widget thumb will allow the user to manually change the playback position. Lastly, the layout defines several `ImageButton` widgets that will facilitate the play/pause, skip back and skip forward playback controls. Throughout the `fragment_currently_playing` layout, you may notice multiple widgets feature `transitionName` attributes, which link them with a corresponding widget in the `fragment_controls` layout. These transition names will facilitate a transition animation where the contents of each widget merge into their counterpart as the user navigates from one fragment to another.

To make the currently playing fragment operational, create a new Kotlin class by right-clicking the **currentlyPlaying** directory (**Project** > **app** > **java** > **name of the project** > **ui**) then selecting **New** > **Kotlin Class/File**. Name the file `CurrentlyPlayingFragment` and select `Class` from the list of options. Once the `CurrentlyPlayingFragment.kt` file opens in the editor, modify its code as follows:

```

import androidx.fragment.app.Fragment
import androidx.fragment.app.activityViewModels
import android.transition.TransitionInflater
import androidx.preference.PreferenceManager

class CurrentlyPlayingFragment : Fragment() {

    private val playbackViewModel: PlaybackViewModel by activityViewModels()
    private var currentlyPlayingSong: Song? = null
    private var _binding: FragmentCurrentlyPlayingBinding? = null
    private val binding get() = _binding!!
    private var fastForwarding = false
    private var fastRewinding = false
    private lateinit var callingActivity: MainActivity
    private lateinit var sharedPreferences: SharedPreferences

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        sharedElementEnterTransition = TransitionInflater.from(context).inflateTransition(android.R.transition.move)
    }
}

```

```

        sharedElementReturnTransition = TransitionInflater.from(context).inflateTransition(android.R.transition.move)
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        _binding = FragmentCurrentlyPlayingBinding.inflate(inflater, container, false)
        callingActivity = activity as MainActivity
        sharedPreferences = PreferenceManager.getDefaultSharedPreferences(requireActivity())
        return binding.root
    }
}

```

In the above code, the `onCreate` method helps process the animation that occurs as the `ControlsFragment` transitions into the `CurrentlyPlayingFragment` and vice versa. In both instances, the animation is set to `move`, which means every widget in the corresponding fragment layouts with a `transitionName` attribute will move to the location of their counterpart widget, thereby creating a transition animation. The next method in the fragment is called `onCreateView`, which initialises the `fragment_currently_playing` layout's binding class, the `callingActivity` variable that will grant access to the `MainActivity` class and its data, and the `sharedPreferences` variable which will allow the fragment to read and write data to a shared preferences file.

Moving on, let's now define the `onViewCreated` method, which will handle user interactions with the fragment's components. Add the following code below the `onCreateView` method:

```

@SuppressLint("ClickableViewAccessibility")
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    // The currently playing fragment will overlay the active fragment from the
    // mobile_navigation navigation graph. We need to intercept touch events
    // that would otherwise reach the underlying fragment
    binding.root.setOnTouchListener { _, _ ->
        return@setOnTouchListener true
    }

    // get information about the currently playing song
    playbackViewModel.currentlyPlayingSong.observe(viewLifecycleOwner, { song ->
        song?.let {
            currentlyPlayingSong = it
            binding.title.text = it.title
            binding.album.text = it.album
            binding.artist.text = it.artist
            callingActivity.insertArtwork(it.albumID, binding.artwork)
        }
    })

    // check whether a song is currently playing
    playbackViewModel.isPlaying.observe(viewLifecycleOwner, { playing ->
        playing?.let {
            if (it) binding.btnPlay.setImageResource(R.drawable.ic_pause)
            else binding.btnPlay.setImageResource(R.drawable.ic_play)
        }
    })

    // keep track of currently playing song duration
    playbackViewModel.currentPlaybackDuration.observe(viewLifecycleOwner, { duration ->
        duration?.let {
            binding.currentSeekBar.max = it
            binding.currentMax.text = SimpleDateFormat("mm:ss", Locale.UK).format(it)
        }
    })
}

```

```

// keep track of currently playing song position
playbackViewModel.currentPlaybackPosition.observe(viewLifecycleOwner, { position ->
    position?.let {
        binding.currentSeekBar.progress = position
        binding.currentPosition.text = SimpleDateFormat("mm:ss", Locale.UK).format(it)
    }
})

// toggle play/pause
binding.btnPlay.setOnClickListener{ callingActivity.playPauseControl() }

// restart or play previous song when backward button is clicked
binding.btnBackward.setOnClickListener{
    if (fastRewinding) fastRewinding = false
    else callingActivity.skipBack()
}

binding.btnBackward.setOnLongClickListener {
    fastRewinding = true
    lifecycleScope.launch {
        do {
            callingActivity.fastRewind()
            delay(500)
        } while (fastRewinding)
    }
    return@setOnLongClickListener false
}

// skip to next song when forward button is pressed
binding.btnForward.setOnClickListener{
    if (fastForwarding) fastForwarding = false
    else callingActivity.skipForward()
}

binding.btnForward.setOnLongClickListener {
    fastForwarding = true
    lifecycleScope.launch {
        do {
            callingActivity.fastForward()
            delay(500)
        } while (fastForwarding)
    }
    return@setOnLongClickListener false
}
}
}

```

Note you may need to add the following import statements to the top of the file:

```

import java.text.SimpleDateFormat
import androidx.lifecycle.lifecycleScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch

```

The `onViewCreated` method runs once the fragment layout's view hierarchy has been established. In the above code, the method registers observers on several `PlaybackViewModel` view model variables, starting with the `currentlyPlayingSong` variable which contains the `Song` object associated with the currently playing song. Data from the `Song` object is used to populate the `fragment_currently_playing` layout's `TextView` and `ImageView` widgets. The next observer monitors the view model's `isPlaying` variable. If the `isPlaying` variable is set to true, then the `btnPlay` `ImageButton` will display a pause icon, while if the variable is set to false then the `ImageButton` will display a play icon instead. Observers are also registered on the `currentPlaybackDuration` and `currentPlaybackPosition` variables, and their values are loaded into the `SeekBar` widget so the user can see the playback progress through the current song. Two `TextView` widgets will also display the current playback position and currently playing song duration. For the playback position and duration values to be displayed correctly, the `SimpleDateFormat` class converts the time values to `mm:ss` (minute : seconds) format. For example, if a song is

three minutes and twenty seconds long then the duration will be displayed as '03:20'.

The code then proceeds to assign an onClick listener to the btnPlay ImageButton. The onClick listener will run the MainActivity class's playPauseControl method to pause/resume playback as appropriate. Next, the actions for the skip forward and skip back buttons are defined. If either button is clicked, then the MainActivity class's skipForward or skipBack will change the currently playing song. Meanwhile, if either button is long pressed, then the MainActivity class's fastForward or fastRewind method will run every 500 milliseconds until the button is released.

There are still some additional components to the fragment\_currently\_playing layout that we must configure. To address this, add the following code to the bottom of the onCreateView method:

```
val shuffleSetting = sharedPreferences.getBoolean("shuffle", false)
if (shuffleSetting) binding.currentButtonShuffle.setColorFilter(ContextCompat.getColor(callingActivity,
R.color.design_default_color_secondary))

binding.currentButtonShuffle.setOnClickListener{
    if (callingActivity.shuffleCurrentPlayQueue())
        binding.currentButtonShuffle.setColorFilter(ContextCompat.getColor(callingActivity,
R.color.design_default_color_secondary))
    else binding.currentButtonShuffle.setColorFilter(ContextCompat.getColor(requireActivity(),
R.color.material_on_surface_emphasis_medium))
}

var repeatSetting = sharedPreferences.getInt("repeat", REPEAT_MODE_NONE)
when (repeatSetting) {
    REPEAT_MODE_ALL -> binding.currentButtonRepeat.setColorFilter(ContextCompat.getColor(callingActivity,
R.color.design_default_color_secondary))
    REPEAT_MODE_ONE -> {
        binding.currentButtonRepeat.setColorFilter(ContextCompat.getColor(callingActivity,
R.color.design_default_color_secondary))
        binding.currentButtonRepeat.setImageDrawable(ContextCompat.getDrawable(requireActivity(),
R.drawable.ic_repeat_one))
    }
}

binding.currentButtonRepeat.setOnClickListener{
    when (repeatSetting) {
        REPEAT_MODE_NONE -> {
            repeatSetting = REPEAT_MODE_ALL
            binding.currentButtonRepeat.setColorFilter(ContextCompat.getColor(callingActivity,
R.color.design_default_color_secondary))
            callingActivity.setRepeatMode(repeatSetting)
            Toast.makeText(requireActivity(), "Repeat play queue", Toast.LENGTH_SHORT).show()
        }
        REPEAT_MODE_ALL -> {
            repeatSetting = REPEAT_MODE_ONE
            binding.currentButtonRepeat.setImageDrawable(ContextCompat.getDrawable(requireActivity(),
R.drawable.ic_repeat_one))
            callingActivity.setRepeatMode(repeatSetting)
            Toast.makeText(requireActivity(), "Repeat current song", Toast.LENGTH_SHORT).show()
        }
        REPEAT_MODE_ONE -> {
            repeatSetting = REPEAT_MODE_NONE
            binding.currentButtonRepeat.setImageDrawable(ContextCompat.getDrawable(requireActivity(),
R.drawable.ic_repeat))
            binding.currentButtonRepeat.setColorFilter(ContextCompat.getColor(requireActivity(),
R.color.material_on_surface_emphasis_medium))
            callingActivity.setRepeatMode(repeatSetting)
            Toast.makeText(requireActivity(), "Repeat mode off", Toast.LENGTH_SHORT).show()
        }
    }
}
```

```
binding.currentClose.setOnClickListener {
    findNavController().popBackStack()
}
```

```
binding.artwork.setOnClickListener {
    showPopup(binding.currentClose)
}
```

```
binding.currentSeekBar.setOnSeekBarChangeListener(object : SeekBar.OnSeekBarChangeListener {
    override fun onStopTrackingTouch(seekBar: SeekBar) {}
    override fun onStartTrackingTouch(seekBar: SeekBar) {}
    override fun onProgressChanged(seekBar: SeekBar?, progress: Int, fromUser: Boolean) {
        if (fromUser) callingActivity.seekTo(progress)
    }
})
```

Note you may need to add the following import statements to the top of the file:

```
import androidx.navigation.fragment.findNavController
import android.support.v4.media.session.PlaybackStateCompat.*
```

The above code begins by retrieving the value stored under the shuffle key in the shared preferences file to determine whether or not the current play queue is shuffled. If the play queue is shuffled, then the secondary colour associated with the active theme is applied to the shuffle ImageButton to indicate that shuffle mode is currently active.



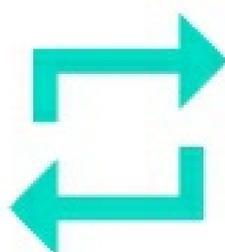
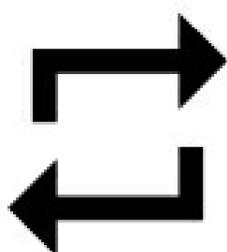
Whenever the user clicks the shuffle button, the MainActivity class's shuffleCurrentPlayQueue method will shuffle or unshuffle the play queue as required. If the play queue is shuffled, then the shuffleCurrentPlayQueue method will return a value of true, which the onClick listener will use as a signal to apply the secondary colour filter to the shuffle button. On the other hand, if the play queue is unshuffled and the shuffleCurrentPlayQueue method returns a value of false, then the onClick listener will apply the regular on\_surface colour filter to the button.

A similar approach is taken with the repeat button. There are three different states the repeat button will cycle through when clicked: REPEAT\_MODE\_ALL, which will repeat the play queue; REPEAT\_MODE\_ONE, which will loop the current song; and REPEAT\_MODE\_NONE, which will deactivate the repeat mode. Different repeat icon drawable resources are used for each repeat mode state. The active theme's secondary colour is also applied to the two active repeat mode state icons, while the regular on\_surface colour filter is used when the repeat mode is inactive. Changes to the repeat mode will be handled by a MainActivity method called setRepeatMode.

NONE

ALL

ONE



In the top left corner of the layout, there is a down arrow button that will allow the user to exit the currently playing fragment. This is achieved via the NavController class's popBackStack method, which will close the currently playing fragment and restore the controls fragment at the bottom of the screen. The navigation transition animation will occur as the currently playing fragment is closed.

If the user clicks the artwork ImageView then a popup menu will invite the user to open the search fragment or view the play queue. We will enable this functionality in the next section. Finally, an OnSeekBarChange listener is applied to the SeekBar widget to detect when the user drags the thumb. The OnSeekBarChange listener contains several callback methods but the only one we will use is called onProgressChanged. The onProgressChanged method records the new position of the thumb during any drag events. The new position is sent to a MainActivity method called seekTo, which will update the current playback position accordingly.

## Handling user interactions with the CurrentlyPlaying fragment

There are several additional methods we must define to handle all of the user's interactions with the currently playing fragment. First, when the user clicks the artwork ImageView widget in the fragment\_currently\_playing layout, a popup menu will open and invite the user to search their music library and view the play queue. The popup menu will require a menu resource file. To create a menu resource file, locate and right-click the **menu** directory (**Project > app > res**), then select **New > Menu Resource File**. Name the file currently\_playing\_menu and once it opens in the editor modify the file's code so it reads as follows:

```
<?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/search"
        android:title="@string/search_hint"
        android:icon="@drawable/ic_search"
        app:actionViewClass="android.widget.SearchView" />

    <item android:id="@+id/queue"
        android:title="@string/play_queue"
        android:icon="@drawable/ic_queue" />
</menu>
```

Each item in the currently\_playing\_menu menu contains a title attribute and an icon attribute that will appear as shown below.

 Search music

 Play queue

To make the popup menu operational, return to the CurrentlyPlayingFragment class (**Project > app > java > name of the project > ui > currentlyPlaying**) and add the following code below the onCreateView method:

```
private fun showPopup(view: View) {
    PopupMenu(requireContext(), view).apply {
        inflate(R.menu.currently_playing_menu)
        setForceShowIcon(true)
        setOnDismissListener {
            callingActivity.hideSystemBars(true)
        }
        setOnMenuItemClickListener {
            when (it.itemId) {
                R.id.search -> {
                    findNavController().popBackStack()
                    callingActivity.findNavController(R.id.nav_host_fragment).navigate(R.id.nav_search)
                }
                R.id.queue -> {
                    findNavController().popBackStack()
                    callingActivity.findNavController(R.id.nav_host_fragment).navigate(R.id.nav_queue)
                }
            }
        }
    }
}
```

```

    }
    true
}
show()
}
}
}

```

Note you may need to add the following import statements to the top of the file:

```

import android.widget.PopupMenu
import androidx.navigation.findNavController

```

The showPopup method uses the Popup menu class to inflate the currently\_playing\_menu menu resource file. Each menu item also contains an icon image; however, by default, the icon will be hidden and only the item title will be displayed. To display both the icon and the title, we must use the Popup class's setForceShowIcon method. An onDismiss listener is also applied to the Popup menu, which will respond to the user dismissing the menu without pressing any of the menu items. In which case, a MainActivity class method called hideSystemBars will hide the system's toolbars and re-enter fullscreen immersive mode.

Finally, an onItemClick listener is applied to the popup menu to define the actions that are associated with the menu's items. If either the search or play queue items are selected, the NavController class's popBackStack method will close the currently playing fragment and restore the playback controls fragment at the bottom of the screen. The onItemClick listener also accesses the NavController that coordinates the destinations in the **mobile\_navigation.xml** navigation graph and navigates to either the search fragment or play queue fragment depending on which menu item was selected. Altogether, these two NavController events close the currently playing fragment and load the user's chosen destination.

To conclude the currently playing fragment, add the following methods below the onCreateView method:

```

override fun onResume() {
    super.onResume()
    callingActivity.hideSystemBars(true)
}

```

```

override fun onDestroyView() {
    super.onDestroyView()
    _binding = null
    callingActivity.hideSystemBars(false)
}

```

Similar to the steps that we had to follow for the popup menu's onDismiss listener, the onResume method defined above uses MainActivity's hideSystemBars method to re-enter fullscreen mode and create a fully immersive environment for the currently playing fragment. The onResume method ensures the fullscreen mode is activated whenever the user launches the currently playing fragment or returns to the app from elsewhere on the device. In contrast, the onDestroyView method uses the hideSystemBars method to exit fullscreen mode whenever the currently playing fragment is closed. This returns the app to its regular view and allows the user to continue to browse the app as normal.

Let's now define the hideSystemBars method itself. Open the **MainActivity.kt** file (**Project > app > java > name of the project**) and add the following code below the fastForward method:

```

fun hideSystemBars(hide: Boolean) {
    val windowInsetsController = ViewCompat.getWindowInsetsController(window.decorView) ?: return
    if (hide) {
        // Configure the behavior of the hidden system bars
        windowInsetsController.systemBarsBehavior =
            WindowInsetsControllerCompat.BEHAVIOR_SHOW_TRANSIENT_BARS_BY_SWIPE
        // Hide both the status bar and the navigation bar
        windowInsetsController.hide(WindowInsetsCompat.Type.systemBars())
        // Hide the toolbar to prevent the SearchView keyboard inadvertently popping up
        binding.toolbar.visibility = View.GONE
    } else {
        windowInsetsController.systemBarsBehavior =
            WindowInsetsControllerCompat.BEHAVIOR_SHOW_BARS_BY_SWIPE
    }
}

```

```

windowInsetsController.show(WindowInsetsCompat.Type.systemBars())
binding.toolbar.visibility = View.VISIBLE
}
}

```

The `hideSystemBars` method accepts a boolean value indicating whether the system bars should be hidden. If the system bars are hidden, then the application will enter fullscreen immersive mode. To achieve this, the `WindowInsetsControllerCompat` class's `hide` and `show` methods toggle the system bars visibility as required. The above code also determines how the system bars should respond to user interactions using the `WindowInsetsControllerCompat` class's `setSystemBarsBehavior` method. Available behaviours include:

- `BEHAVIOR_SHOW_BARS_BY_TOUCH` - The system bars will be revealed whenever the user touches the corresponding display.
- `BEHAVIOR_SHOW_BARS_BY_SWIPE` - The system bars will be revealed when the user swipes from the edge of the screen where the bar is hidden from.
- `BEHAVIOR_SHOW_TRANSIENT_BARS_BY_SWIPE` - The system bars will be temporarily revealed when the user swipes from the edge of the screen where the bar is hidden from. After a short timeout, the bar will be hidden again.

In the above code, we set the system bars to be revealed using swipe gestures; however, the reveal will be transient if the app is in fullscreen mode. Also, the above code hides the app's toolbar when fullscreen mode is active. Otherwise, menu items, navigational destination titles and other distractions may interfere with the currently playing fragment. If the `hideSystemBars` method is called with a value of `false` passed as the `hide` argument, then the system bars and app toolbar will become visible and behave normally again.

With regards to the transition between the playback controls fragment and the fullscreen currently playing fragment, there is just one more aspect that we must consider. Namely, how we will handle presses of the back button while the currently playing fragment is open. By default, the back button will run the `NavController` class's `popBackStack` command to return the user to their previous destination; however, this application uses two navigation graphs. For this reason, we must provide additional instructions for how the back button should respond when the currently playing fragment is open. Locate the `onBackPressed` method and replace the `TODO` comment with the following code:

```

val id = findNavController(R.id.nav_controls_fragment).currentDestination?.id ?: 0
if (id == R.id.nav_currently_playing) {
    findNavController(R.id.nav_controls_fragment).popBackStack()
    hideSystemBars(false)
} else super.onBackPressed()

```

The above code finds the ID of the current destination for the `NavController` associated with the `controls_navigation.xml` navigation graph. If the current destination ID belongs to the currently playing fragment, then the `NavController`'s `popBackStack` method and the `hideSystemBars` method will restore the playback controls fragment and exit fullscreen mode. Meanwhile, if the currently playing fragment is not open, then the `onBackPressed` method will continue as normal and return the user to their previous destination `mobile_navigation.xml` navigation graph's backstack.

With the transition to and from the currently playing fragment now taken care of, let's turn our attention to the other methods that are required to handle the user's interactions with the playback controls. First, add the following method below the `hideSystemBars` method to handle requests to shuffle and unshuffle the play queue:

```

// Returns true if the play queue has been shuffled, false if unshuffled
fun shuffleCurrentPlayQueue(): Boolean {
    val isShuffled = sharedPreferences.getBoolean("shuffle", false)
    if (playQueue.isNotEmpty()) {
        if (isShuffled) {
            playQueue.sortBy { it.first }
            Toast.makeText(applicationContext, "Play queue unshuffled", Toast.LENGTH_SHORT).show()
        } else {
            val currentItem = playQueue.find {
                it.first == currentlyPlayingQueueID
            }
            if (currentItem != null) {
                playQueue.remove(currentItem)
                playQueue.shuffle()
            }
        }
    }
}

```

```

        playQueue.add(0, currentItem)
        Toast.makeText(applicationContext, "Play queue shuffled", Toast.LENGTH_SHORT).show()
    }
}
playbackViewModel.currentPlayQueue.value = playQueue
}

```

```

sharedPreferences.edit().apply {
    putBoolean("shuffle", !isShuffled)
    apply()
}
return !isShuffled
}

```

The `shuffleCurrentPlayQueue` method retrieves the boolean value listed under the `shuffle` key in the shared preferences file. If the value associated with the `shuffle` key is true, then the method unshuffles the play queue by using Kotlin's `sortBy` method to sort the list of `Pair` objects stored in the `playQueue` variable. The `sortBy` method will organise the list of `Pair` objects based on the integer value they were assigned when added to the play queue, which will return the list to its original order. Meanwhile, if the value associated with the `shuffle` key is false, then the method will shuffle the list of `Pair` objects. Before the list is shuffled, however, the `Pair` object associated with the currently playing song is removed so it can be added to the beginning of the newly shuffled list. The result is a list of shuffled `Pair` objects with the currently playing song at the beginning. A toast notification informs the user that the play queue has been shuffled.

Once all the above processing is complete, the updated list of `Pair` objects is sent to the `PlaybackViewModel` view model's `currentPlayQueue` variable. The value associated with the `shuffle` key in the shared preferences file is also updated with a boolean value indicating whether or not the play queue is currently shuffled.

Moving on, add the following code below the `shuffleCurrentPlayQueue` method to define a method called `setRepeatMode` that will handle changes to the repeat mode:

```

fun setRepeatMode(repeatMode: Int): SharedPreferences.Editor
= sharedPreferences.edit().apply {
    putInt("repeat", repeatMode)
    apply()
}

```

The `setRepeatMode` method writes the user's repeat mode preference to the shared preferences file under the key called `repeat`. The user's repeat mode preference can be retrieved when required such as when the playback of a song finishes or the user opens the currently playing fragment and the fragment must determine which drawable icon to use for the repeat mode `ImageButton` widget. The numbers representing the different repeat modes can be found in the Android documentation:

[https://developer.android.com/reference/kotlin/android/support/v4/media/session/PlaybackStateCompat#repeat\\_mode](https://developer.android.com/reference/kotlin/android/support/v4/media/session/PlaybackStateCompat#repeat_mode)  
For example, `REPEAT_MODE_ONE` equals 1.

The last method is called `seekTo`, which is called whenever the user changes the position of the `SeekBar` thumb in the currently playing fragment:

```

fun seekTo(position: Int) = mediaController.transportControls.seekTo(position.toLong())

```

The `seekTo` method forwards the new position to the media browser service, which adjusts the playback position in the media stream accordingly.

## Building the music library

Every fragment required for the app to run is now properly configured. All that remains to do is define the methods that will maintain the music library. Let's begin by defining the methods that will scan the user's device for music. Open the `MainActivity.kt` file (**Project > app > java > name of the project**) add the following code below the `seekTo` method:

```

private fun musicQueryAsync(projection: Array<String>): Deferred<Cursor?> =
    lifecycleScope.async(Dispatchers.IO) {
        val selection = MediaStore.Audio.Media.IS_MUSIC
        val sortOrder = MediaStore.Audio.Media.TITLE + " ASC"
        return@async application.contentResolver.query(

```





and assign the song to disc 1. For instance, the when block would convert a track number of 1 to 1001. Also, if there is a problem interpreting the song's metadata (e.g. sometimes the track number can appear in an unexpected format like '11/23') then a number format exception may be thrown. In which case, a catch block intercepts the exception and sets the track property to 1001, which represents track 1 of disc 1. The user can rectify the track value via the edit song fragment.

Further metadata attributes retrieved from the Cursor interface columns include the song's title, artist name, album name, release year and album ID. An Elvis operator `?:` is included with each attribute to define the value that should be used if a certain field is missing. For example, if the artist column for a given song is null, then the value on the right-hand side of the Elvis operator "Unknown artist" will be used instead.

The Song object for each song in the user's music library will also contain a content URI, which contains the location of the audio file in the media store (e.g. `content://media/external/audio/media/`) and the song's ID. So the complete content URI will look like this `content://media/external/audio/media/271`. The app can later use this content URI to locate the associated audio file and play the song.

The next section of the `libraryRefresh` method focuses on retrieving the song's album artwork. First, the code uses the `getArtworkFile` method we defined earlier to generate a File object associated with the song's album. The File object will reference a JPEG image file in the app's internal storage that will be named based on the song's album ID. Next, an if expression checks whether the JPEG file referenced in the File object exists. If the file exists, then this means artwork for the album has already been saved and no further action is required. On the other hand, if a corresponding image file cannot be found, then the above code attempts to extract artwork from the audio file's metadata using the ContentResolver class's `loadThumbnail` method. The `loadThumbnail` method will attempt to extract a Bitmap representation of the audio file's album artwork. Also, the Bitmap will be resized to 640 x 640 pixels, as specified in the arguments that the above code supplies to the `loadThumbnail` method. The resultant Bitmap image will be saved as a JPEG file to the app's internal storage using the `saveImage` method. Finally, the completed metadata is used to build a Song object and inserted into the Room database via the MusicViewModel view model.

To monitor the music library, the MainActivity class will register an observer on the MusicViewModel view model's `allSongs` variable. To initialise the observer, add the following code to the bottom of the `onCreate` method:

```
musicViewModel.allSongs.observe(this) { songs ->
    songs.let { completeLibrary = it.toMutableList() }
}
```

The list of Song objects that comprise the user's music library will now be automatically transferred to the MainActivity class's `completeLibrary` variable whenever the underlying data in the Room database changes.

## Maintaining the music library

The user's music library will be refreshed each time the user returns to the app to determine whether any new audio files have been added or old files have been deleted. To handle this, add the following code to the **MainActivity.kt** file (**Project > app > java > name of the project**) below the `onCreate` method:

```
override fun onResume() {
    super.onResume()
    volumeControlStream = AudioManager.STREAM_MUSIC

    if (ContextCompat.checkSelfPermission(this, Manifest.permission.READ_EXTERNAL_STORAGE) ==
    PackageManager.PERMISSION_GRANTED) {
        libraryMaintenance()
    } else ActivityCompat.requestPermissions(this, arrayOf(Manifest.permission.READ_EXTERNAL_STORAGE), 1)
}
```

Note you may need to add the following import statement to the top of the file:

```
import android.Manifest
```

The `onResume` method refers to a stage of the Android activity lifecycle that runs when the activity is ready to handle user interactions. In the above code, we set the activity's `volumeControlStream` attribute to `STREAM_MUSIC`, which means the app will respond to the device's volume control buttons by altering the volume of the music being played. The remainder of the method checks whether the user has granted the app permission to access the device's files. If permission has been granted, then a method called `libraryMaintenance` will update the user's music library. Otherwise, the method will request permission from the user to access the

device's storage.

To handle the result of the permissions request, add the following code below the onResume method:

```
override fun onRequestPermissionsResult(requestCode: Int, permissions: Array<out String>, grantResults: IntArray) {  
    super.onRequestPermissionsResult(requestCode, permissions, grantResults)  
    if (ContextCompat.checkSelfPermission(this, Manifest.permission.READ_EXTERNAL_STORAGE) == PackageManager.PERMISSION_GRANTED) {  
        libraryMaintenance()  
    } else {  
        Toast.makeText(this, getString(R.string.permission_required), Toast.LENGTH_LONG).show()  
        finish()  
    }  
}
```

The above code checks whether the user has granted the app permission to access the device's storage. If permission has been granted, the libraryMaintenance method will update the user's music library. Meanwhile, if permission has not been granted, then a Toast notification will inform the user that storage permissions are required for the app to run and a method called finish will close the activity and the app.

To define the libraryMaintenance method, add the following code below the libraryRefresh method:

```
private fun libraryMaintenance() = lifecycleScope.launch(Dispatchers.Main) {  
    libraryRefresh()  
    if (completeLibrary.isNotEmpty()) {  
        val songsToDelete = checkLibrarySongsExistAsync().await()  
        if (songsToDelete.isNotEmpty()) deleteSongs(songsToDelete)  
    }  
}
```

This libraryMaintenance method runs the libraryRefresh method to scan the user's device for music and build the music library. If the music library is not empty, then a method called checkLibrarySongsExistAsync will return a list of all the Song objects that are associated with audio files that no longer exist. The list of Song objects is sent to a method called deleteSongs, which will remove the Song objects from the music library.

To define the checkLibrarySongsExistAsync method, add the following code below the libraryMaintenance method:

```
private fun checkLibrarySongsExistAsync(): Deferred<List<Song>> = lifecycleScope.async(Dispatchers.IO) {  
    var songsToBeDeleted = mutableListOf<Song>()  
    val projection = arrayOf(MediaStore.Audio.Media._ID)  
    val libraryCursor = musicQueryAsync(projection).await()  
    libraryCursor?.use { cursor ->  
        val idColumn = cursor.getColumnIndexOrThrow(MediaStore.Audio.Media._ID)  
        songsToBeDeleted = completeLibrary.toMutableList()  
  
        while (cursor.moveToNext()) {  
            val id = cursor.getLong(idColumn)  
  
            val indexOfSong = songsToBeDeleted.indexOfFirst { song: Song ->  
                song.songID == id  
            }  
            if (indexOfSong != -1) songsToBeDeleted.removeAt(indexOfSong)  
        }  
    }  
    return@async songsToBeDeleted  
}
```

Similar to the libraryRefresh method, the checkLibrarySongsExistAsync method builds a Cursor interface containing the details of every music audio file on the user's device. The difference this time is that the Cursor interface table will contain one column only, which is a column detailing the song's ID. Once the Cursor has been generated, a list variable called songsToBeDeleted is initialised. Initially, the list will contain every Song object in the music library. The method then iterates through the song IDs in the Cursor interface and removes the corresponding Song object from the list. Once every record in the Cursor has been processed, any remaining Song

objects in the list will be those that no longer exist on the device.

The list of Song objects that must be deleted is sent to a method called deleteSongs. The deleteSongs method will remove the Song objects from the Room database and play queue (if applicable). To define the deleteSongs method, add the following code below the checkLibrarySongsExistAsync method:

```
private suspend fun deleteSongs(songs: List<Song>) {
    for (s in songs) {
        // Remove all instances of the song from the play queue
        if (playQueue.isNotEmpty()) {
            do {
                val index = playQueue.indexOfFirst {
                    it.second.songID == s.songID
                }

                if (index != -1) removeQueueItem(index)
            } while (index != -1)
        }
        musicViewModel.deleteSong(s)
    }
    tidyArtwork(songs)
}
```

The deleteSongs method uses a for loop to iterate through the list of Song objects and determine whether any of the Song objects are present in the play queue. If a given song is found in the play queue, then a do/while block will find the index of each instance of the song in the play queue and remove them using the removeQueueItem method. Once all instances of the song have been removed from the play queue, the Song object is sent to the MusicViewModel view model's deleteSong method which will remove the Song object from the Room database. Finally, once all the Song objects have been processed, a method called tidyArtwork will delete any album artwork images files that are no longer required.

Let's now define the tidyArtwork method. Add the following code below the deleteSongs method:

```
private suspend fun tidyArtwork(songs: List<Song>) {
    val directory = ContextWrapper(application).getDir("albumArt", Context.MODE_PRIVATE)
    val musicDatabase = MusicDatabase.getDatabase(this)
    for (s in songs) {
        val artworkInUse = musicDatabase.musicDao().doesAlbumIDExist(s.albumID)
        if (artworkInUse.isNullOrEmpty()) {
            val path = File(directory, s.albumID + ".jpg")
            if (path.exists()) path.delete()
        }
    }
}
```

The tidyArtwork method iterates through the list of deleted Song objects and sends their album ID to a MusicDao class method called doesAlbumIDExist. If you refer to the **MusicDao.kt** class file (**Project > app > java > name of the project**) then you will see the doesAlbumIDExist method executes the following SQL query:

```
SELECT * from music_table WHERE song_album_id LIKE :albumID LIMIT 1
```

The above query searches the music\_table database table for any entries that have a song\_album\_id value matching the supplied album ID. The term 'LIMIT 1' is appended to the query which means a list containing a maximum of one Song object will be returned. The purpose of this query is to determine whether there are any Song objects in the Room database that still belong to the album ID of a deleted song. If there are no longer any entries for a given album ID, then that means any album artwork image file associated with that album ID can be deleted because it is no longer used. If the output list from the SQL query is empty for a given album ID, then the tidyArtwork method builds a File object leading to the JPEG file associated with that album ID in the artwork directory. If the File object is associated with a genuine file, then the file is deleted.

## Summary

Congratulations on completing the Music player app! In creating the app, you have covered the following skills and topics:

- Utilise and query the data in a Room SQLite database.
- Use a MediaBrowserService to coordinate the playback of audio files and display a notification to the user.
- Apply an ItemTouchHelper instance to a RecyclerView widget to respond to user gestures.
- Implement a navigation drawer that allows the user to navigate to different areas of the app.
- Utilise multiple NavController to coordinate different navigation graphs and navigation pathways.
- Save, delete and retrieve files from the app's internal files.
- Manually hide and reveal the soft input keyboard.
- Write coroutines that coordinate tasks behind the scenes.
- Display a popup menu to allow the user to interact with items in the play queue.
- Enter and exit fullscreen immersive mode.

# Releasing an application on the Google Play store

## Introduction

Once you have finished developing an application, you may like to share your hard work with the world. For this purpose, the Google Play store is the most popular distributor of Android applications and will help you reach the widest audience possible. In this section, we will cover how to bundle your application for distribution and publish it on the Google Play store.

To publish an application to the Google Play store, you will require a Google Play Developer account. Before you can create a Google Play Developer account, you must pay a one-time fee, which is currently \$25. For more information about the registration fee, then visit Google Play's Support page: <https://support.google.com/googleplay/android-developer/answer/6112435?hl=en-GB#zippy=%2Cstep-sign-up-for-a-google-play-developer-account%2Cstep-accept-the-developer-distribution-agreement%2Cstep-pay-registration-fee%2Cstep-complete-your-account-details>

## Preparing your application for publication

Before you publish an application, you should thoroughly test it to ensure it is easy to use, bug-free and does not crash. Ideally, you should test the application on at least one physical device (the more the better) as described at the beginning of the book in the section called 'Testing an application on a real device'. You can also test your application using Android Studio's virtual device emulator, which allows you to install your application on various virtual device types and models, including phones, tablets and wearable devices (see the previous section called 'Testing an application on a virtual device').

Once you have thoroughly tested the app and ensured its features work correctly, there are several other checks you should perform to ensure your project's code is ready for deployment. First, if you log any messages to the console or Logcat, then you should remove those calls from your code. For example, the following is a Log call that would need to be deleted:

```
Log.d("debug", "Log this message to the Logcat")
```

Next, open the Module-level **build.gradle** file (**Project > Gradle Scripts**) and locate the defaultConfig element. You should find two properties called `versionCode` and `versionName`.

```
android {  
    ...  
    defaultConfig {  
        versionCode 1  
        versionName "1.0"  
    }  
}
```

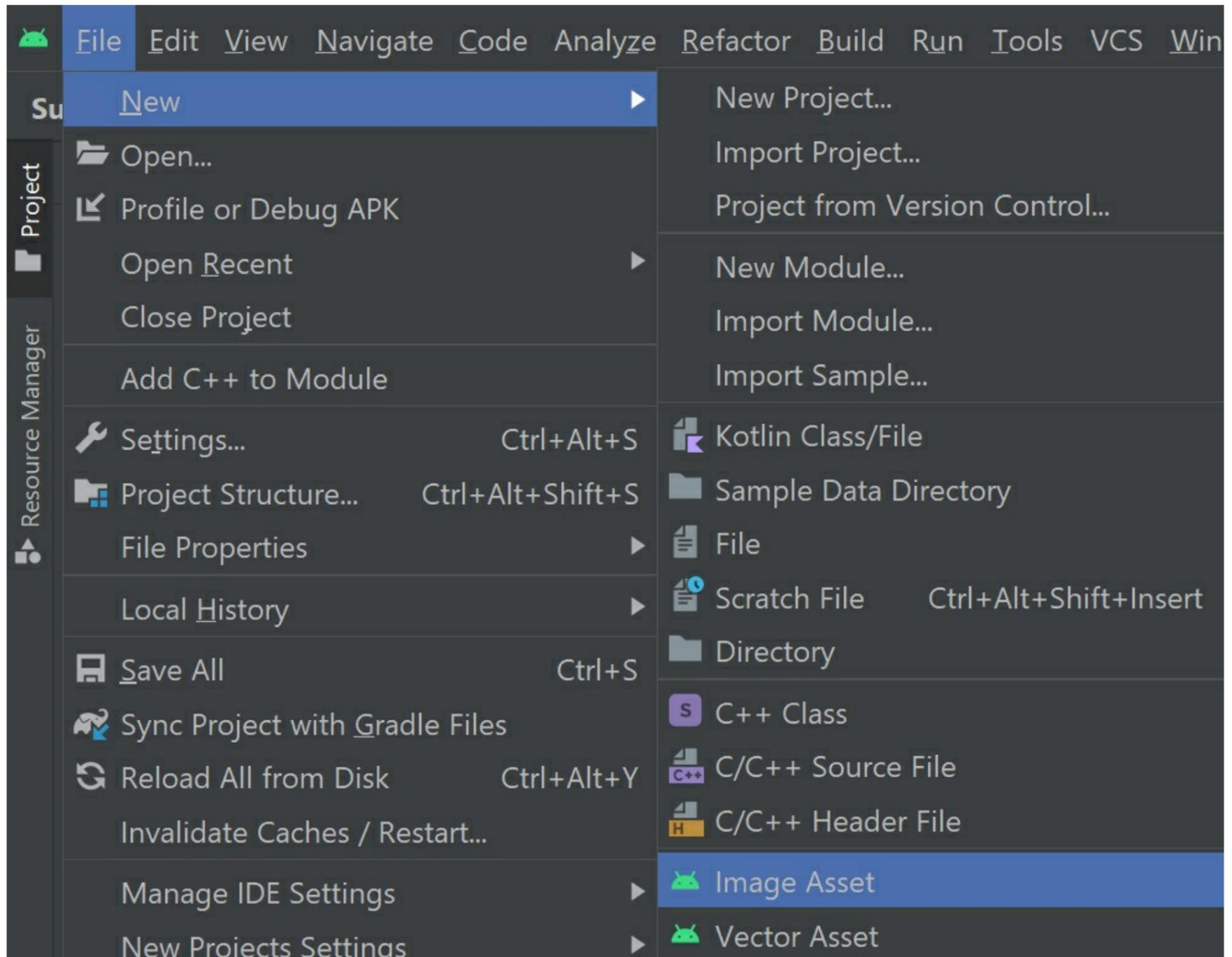
The first time you upload your application to Google, the `versionCode` value should be 1. For each subsequent application update, you will need to increase the `versionCode` value incrementally by 1 because Google does not permit two versions of your application to share the same version code. Devices use the `versionCode` value to protect against downgrades. In case you're curious, the maximum permissible `versionCode` value is 2100000000. That's a lot of updates!

The `versionName` property is for user and developer reference only. Unlike the `versionCode` property, the `versionName` property will be displayed in the Google Play store listing. By convention, the version name is reported in `<major>.<minor>.<point>` format. Typically, the first version of your application will have a version name like "1.0". The major digit should increase if you release a new version of the application that is significantly different to its predecessors (e.g. "2.0"). The minor digit should increase for regular updates such as performance improvements, bug fixes and new features (e.g. "1.1"). Finally, if the update is small, such as a minor bug fix, then you might like to include a point value (e.g. "1.0.1").

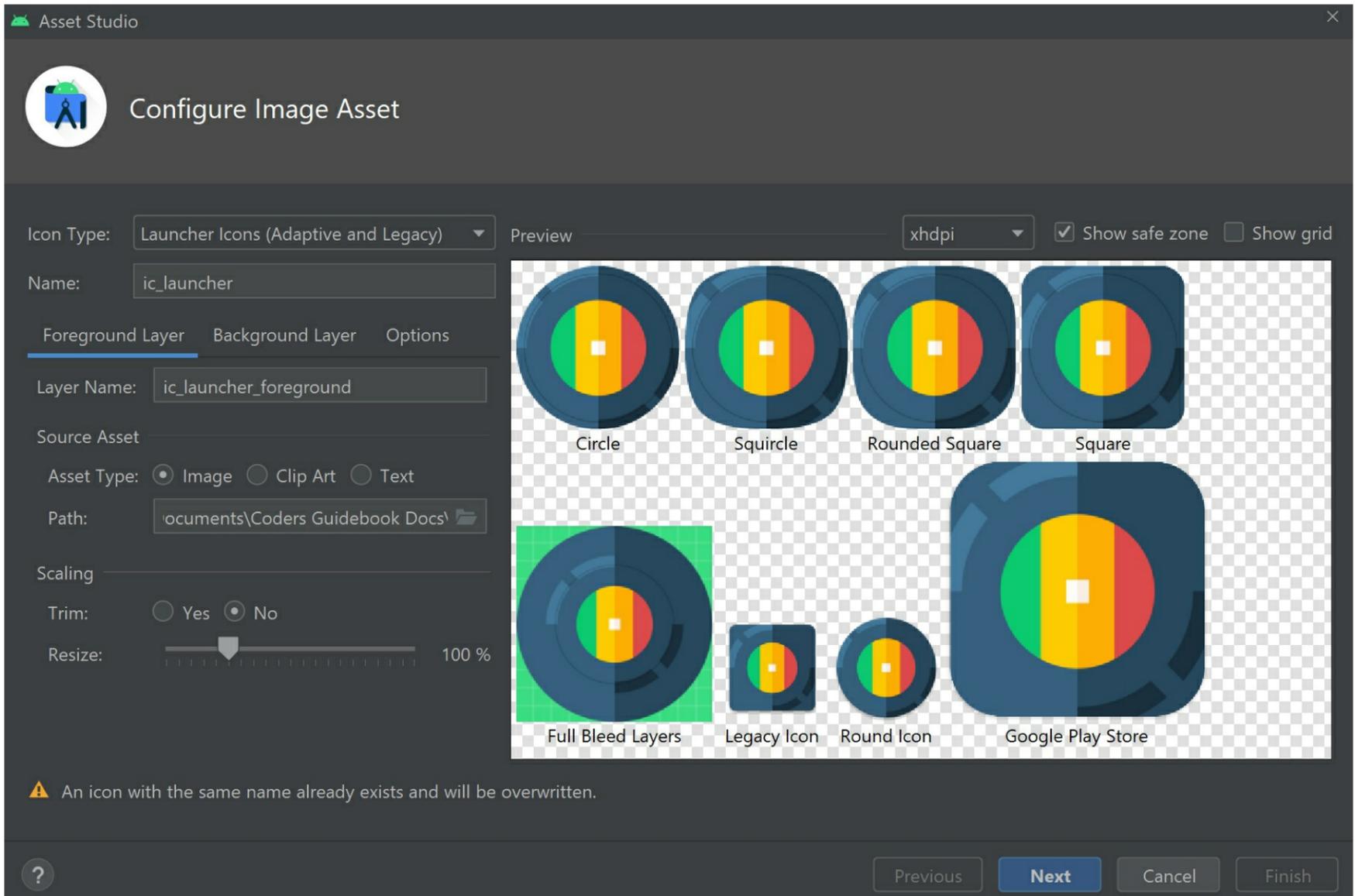
When you prepare an application for release, it is highly recommended that you create a launcher icon. The

launcher icon will identify your application on the user's device and Google Play store and is the icon that the user will click to open your application. For this purpose, you can create an icon or download one from an icon repository such as Freepik (<https://www.freepik.com/>). Be careful that you adhere to the relevant copyright guidelines when using third-party icons. For example, some icons will require you to pay a licence fee or attribute the author in your application and Google Play store listing.

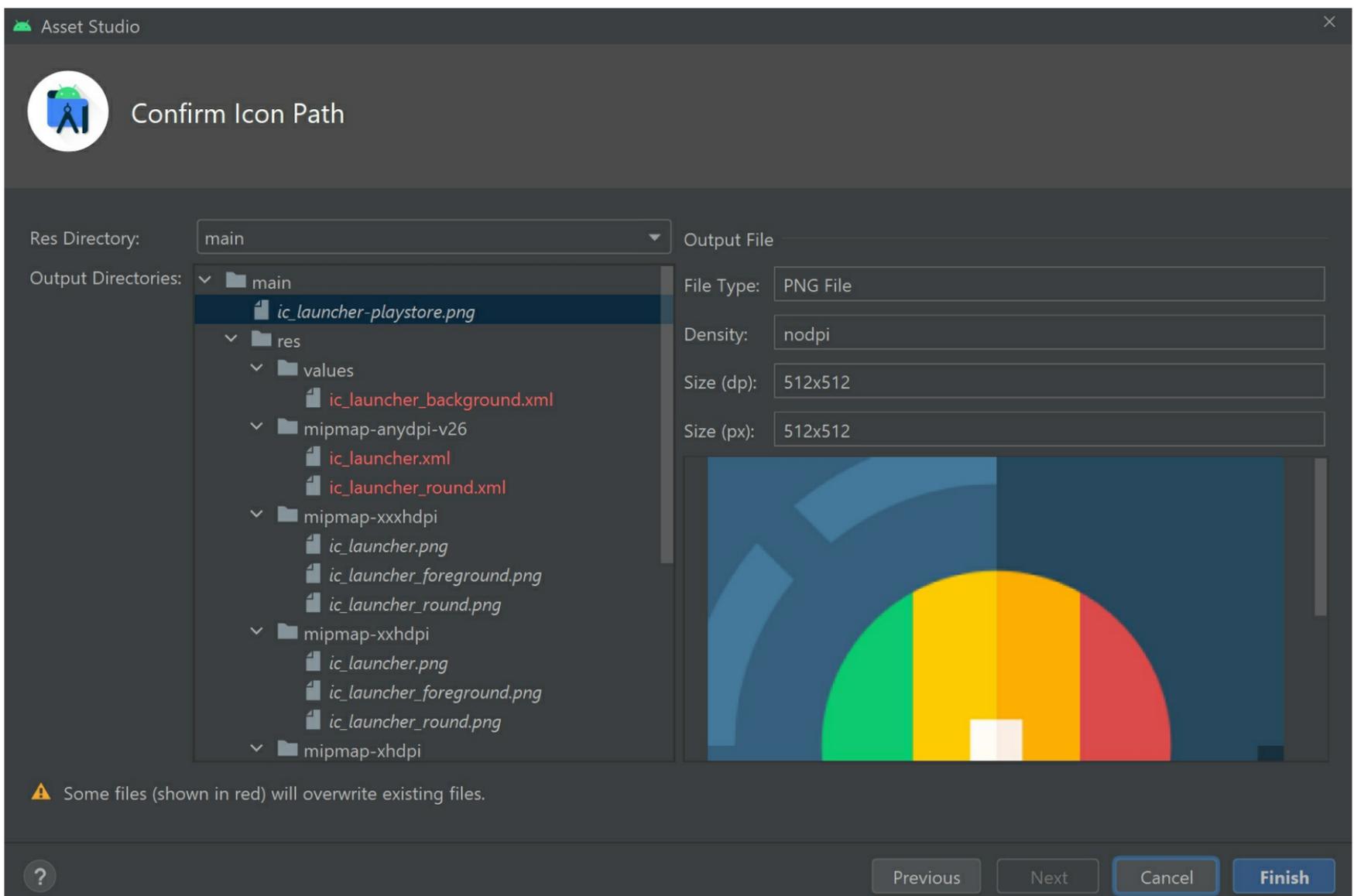
Once you've created or sourced an icon image, you can use Android Studio's Asset Studio to create the various resource files required to use the image as a launcher icon. For example, the Asset Studio tool will automatically create mipmap resources with the required shapes and pixel densities required for different devices and usage situations. To open the Asset Studio tool, click **File > New > Image Asset**



In the Asset Studio window, set the Icon Type to Launcher Icons (Adaptive & Legacy). Next, in the Foreground Layer section set the Source Image Asset Type to Image and locate the image file that you would like to use as the icon. In the preview window, you should see the various iterations of the launcher icon that the Asset Studio tool will generate. If you do not like the background colour that is shown in previews such as the Full Bleed Layers version, then you can change the colour in the Background Layer tab.



It is fine to leave the Foreground Layer Name as `ic_launcher_foreground` because we are looking to overwrite the default icon that was generated by Android Studio. Feel free to try out the other Image Asset configuration options and customise the icons to your liking. Once you are happy with the results, click Next. You should see a list of files that will be created. Some files may be highlighted in red. The highlighted icons were automatically generated by Android Studio, so it is fine to replace them.

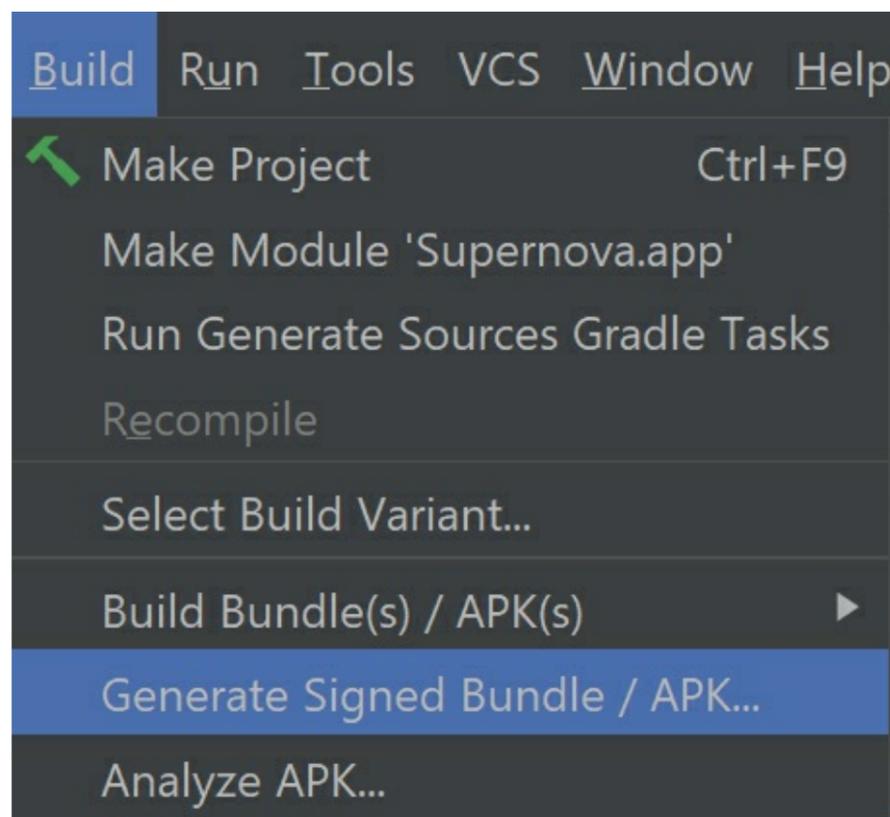


Click Finish to create the icon resources and add them to your project's files. From now on, whenever you install the application, your new launcher icon will be used rather than the default Android Studio icon. Note the above example assumes you stick to the regular naming convention of calling the launcher icon resource ic\_launcher. If you change the launcher icon name, then you will need to update the application manifest. To do this, open the **AndroidManifest.xml** file (**Project > app > Manifest**). Locate the application element and refer to the properties called icon and roundIcon. By default, the icon values will reference mipmap resources called ic\_launcher and ic\_launcher\_round, respectively; however, if you choose a different launcher icon name then you will need to update these values accordingly.

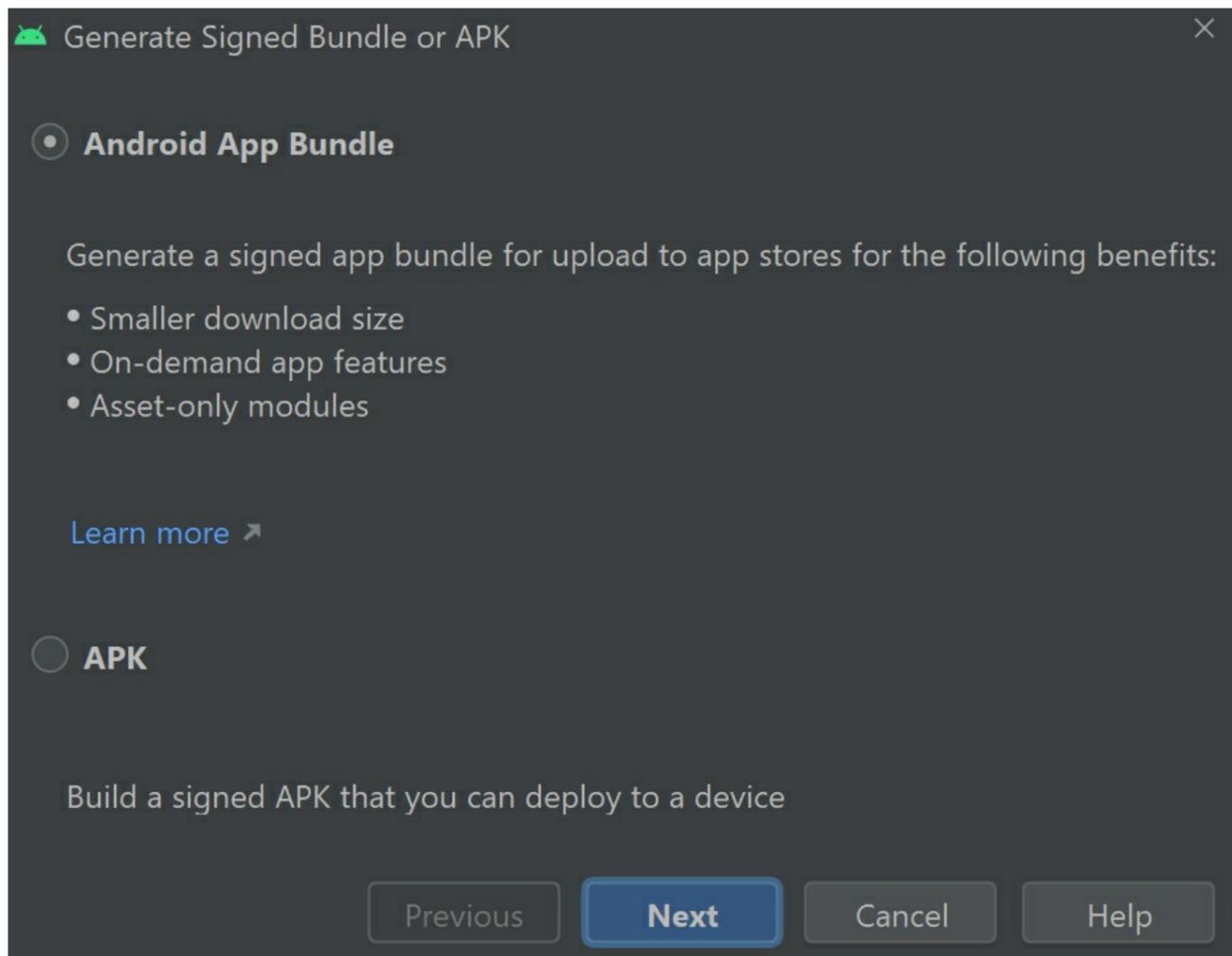
```
<application
  android:icon="@mipmap/ic_launcher"
  android:roundIcon="@mipmap/ic_launcher_round"
  ... >
```

## Creating an Android Application Bundle

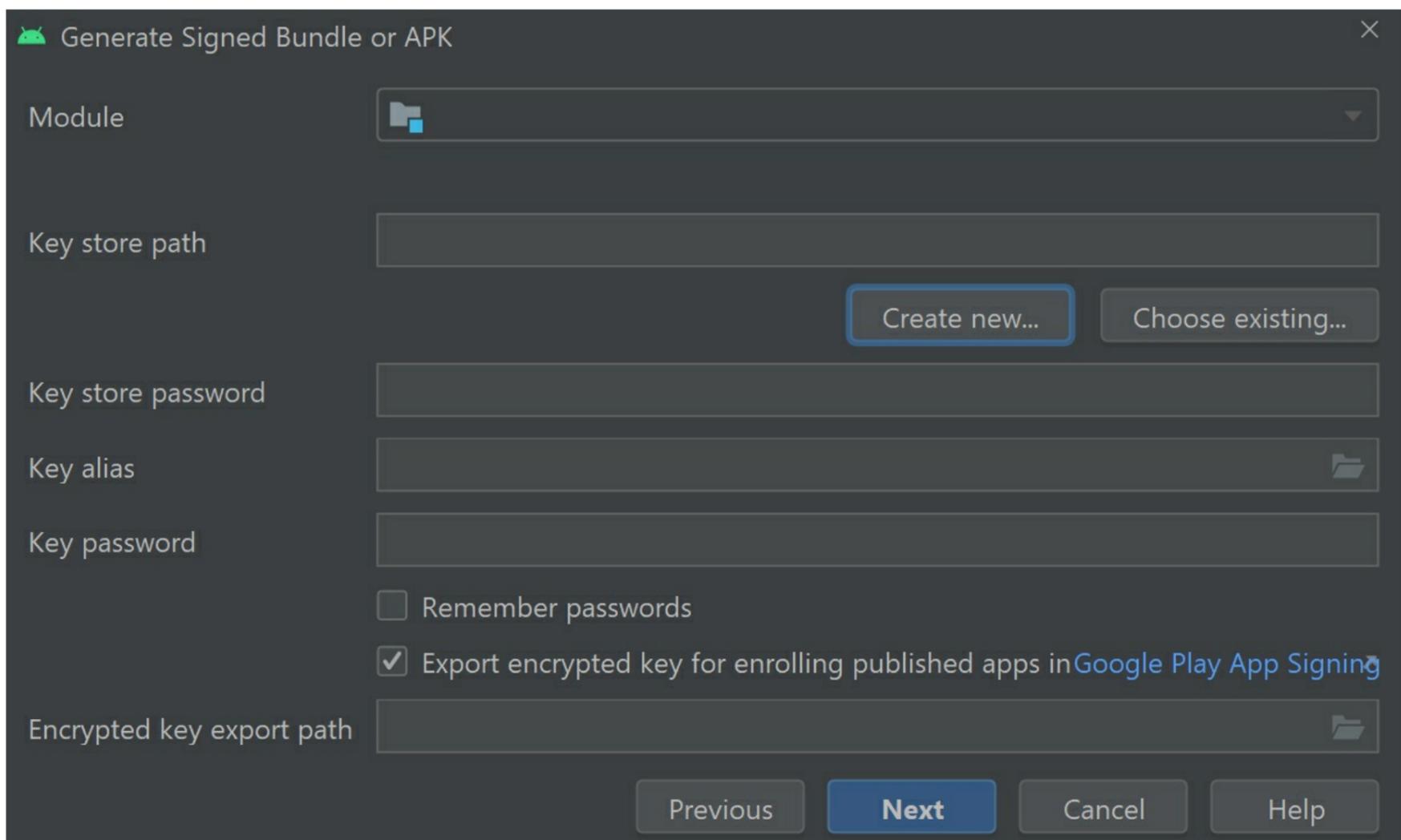
To distribute your application via the Google Play store, you will need to package your application in an Android App Bundle. For Google to accept an Android App Bundle, it must be accompanied by a certificate to prove that the application is authentic. To generate the necessary certificate, you must configure a key store using Android Studio. First, select **Build > Generate Signed Bundle / APK ...**



Next, select Android App Bundle and click Next.



In the Generate Signed Bundle or APK window, click the Create New... button under the Key store path field.



In the New Key Store window, you will be invited to define a name and password for the key store and create a key for your app.

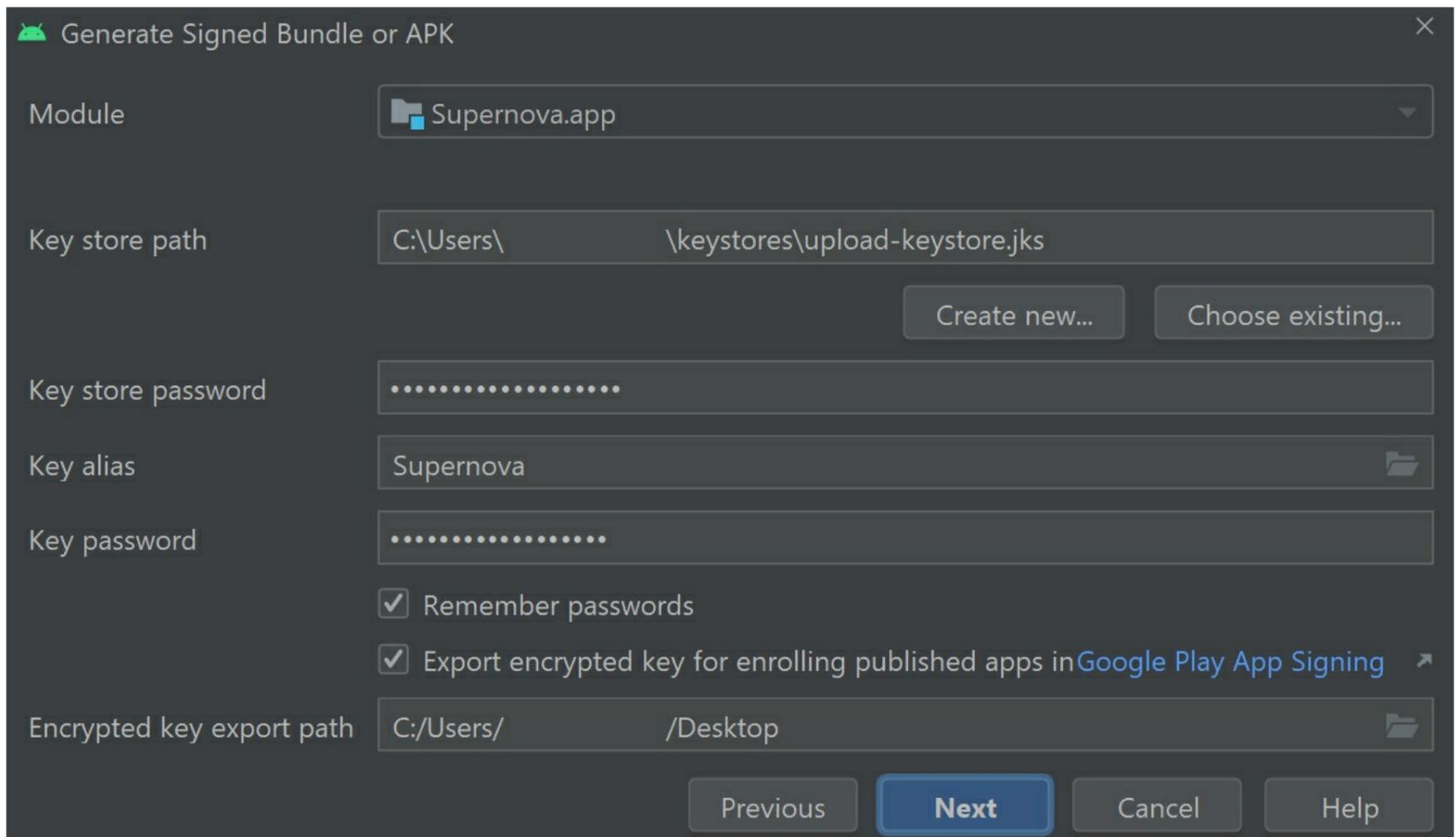
The image shows a 'New Key Store' dialog box with the following fields and values:

- Key store path:** C:\Users\... \keystores\upload-keystore.jks
- Password:** [masked]
- Confirm:** [masked]
- Key Alias:** my\_app\_key
- Key Password:** [masked]
- Key Confirm:** [masked]
- Validity (years):** 25
- Certificate:**
  - First and Last Name: [empty]
  - Organizational Unit: [empty]
  - Organization: [empty]
  - City or Locality: [empty]
  - State or Province: [empty]
  - Country Code (XX): [empty]

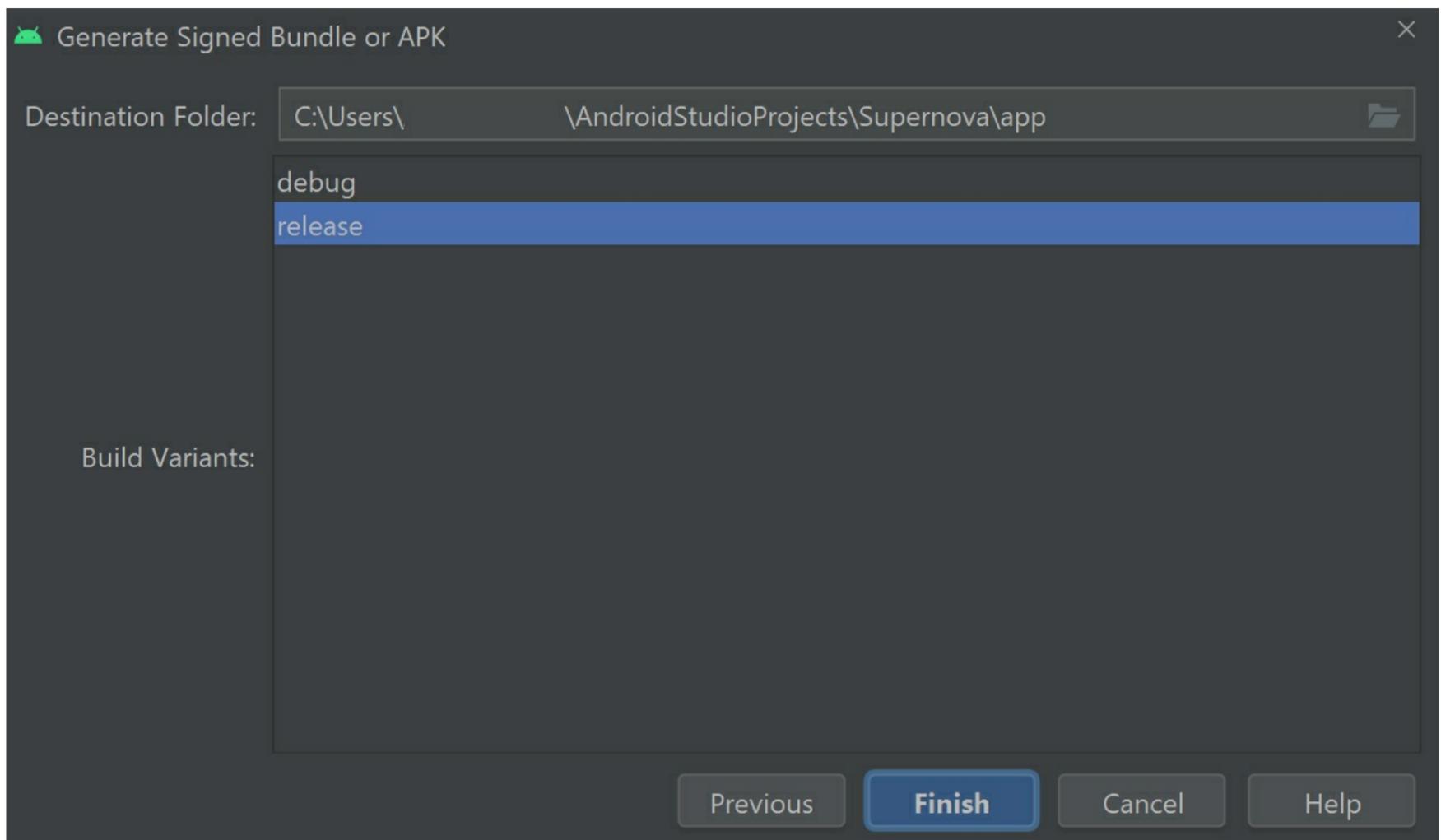
When completing the form in the New Key Store window, you should look to provide the following information:

- **Key store path** - The location on your computer where the .jks file for your key store will be stored. Typically, the key store file will be saved under the home directory for your user in a folder called key stores. You can use the same key store for multiple apps.
- **Key store password** - Input a secure password for your key store.
- **Key alias** - Enter an identifying name for your key. The key alias will be associated with this application only.
- **Key password** - Enter a password for the key. It is usually fine to enter a unique password; however, if you encounter a 'Key was created with errors' error then you will need to use the same password as the key store. This is a known issue that you can read about in the official Android documentation: <https://developer.android.com/studio/known-issues#ki-key-key-store-warning>
- **Key validity (years)** - Set how many years the key should be valid for. Typically, the validity is 25 years because the key should be valid for the lifetime of the application.
- **Certificate** - Enter the required details about yourself or your organisation. This information will not be displayed in the app but is included in the signing certificate that is packaged with the Android App Bundle.

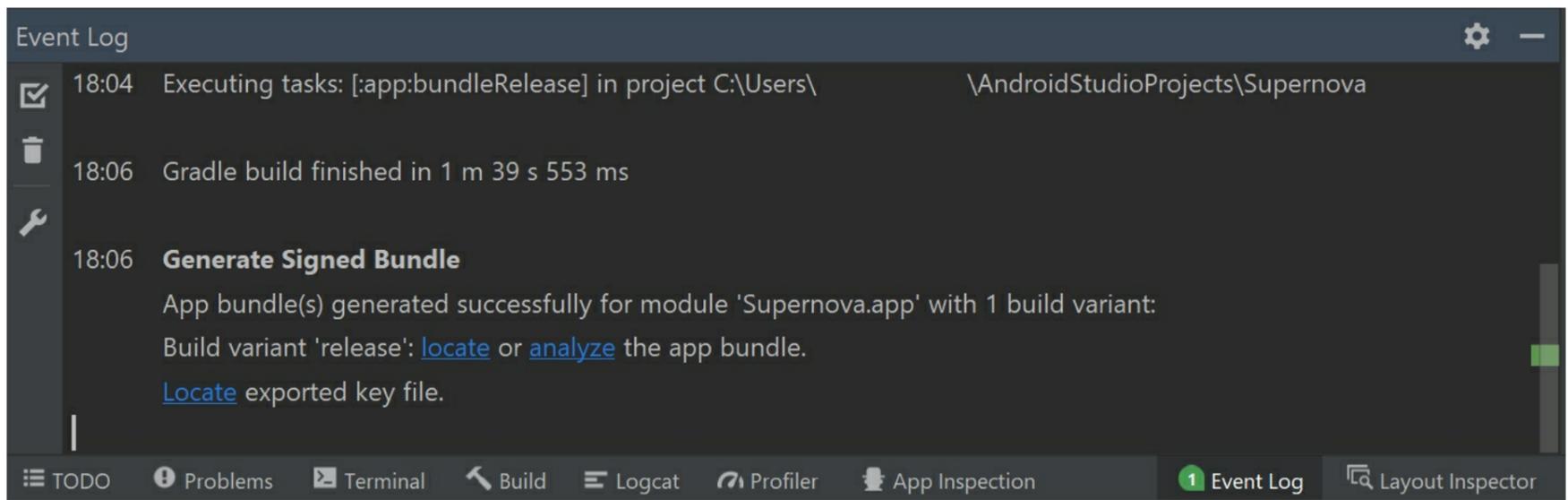
Once you have provided all the required details, click OK to create the key store file and register a key for your application. When you return to the Generate Signed Bundle or APK window, check that the Key store path, Key store password, Key alias and Key password fields are all correct given the key store and key you just created. If you ever need to create a new key for another app or select an existing key, then press the folder icon in the Key alias field.



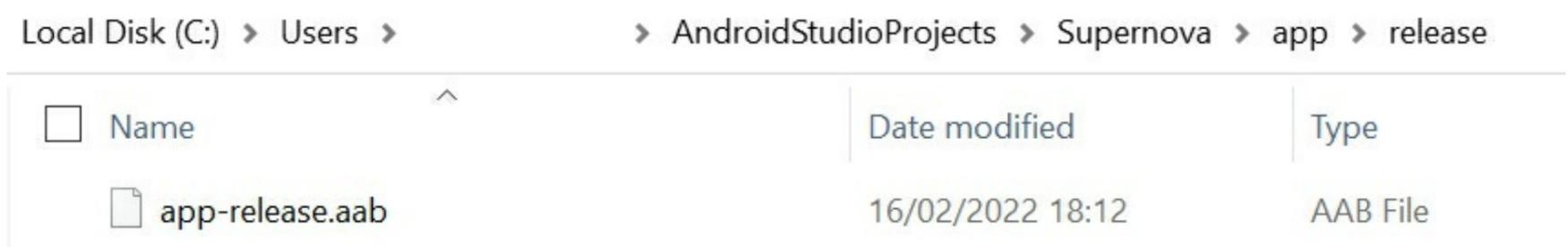
Once you've entered all the required details for your key store and application key, click Next. In the Generate Sign Bundle or APK window, select the Build Variant that you would like to create an Android App Bundle for. When you are publishing an application to the Google Play store, you will usually want to select the release build variant. Next, press Finish and Android Studio will compile your project code and create an Android App Bundle file.



Once the Android App Bundle has been created, a notification should advise where you can locate the bundle on your computer. If you miss the notification, then open the Event Log tab and find the Generate Signed Bundle message.

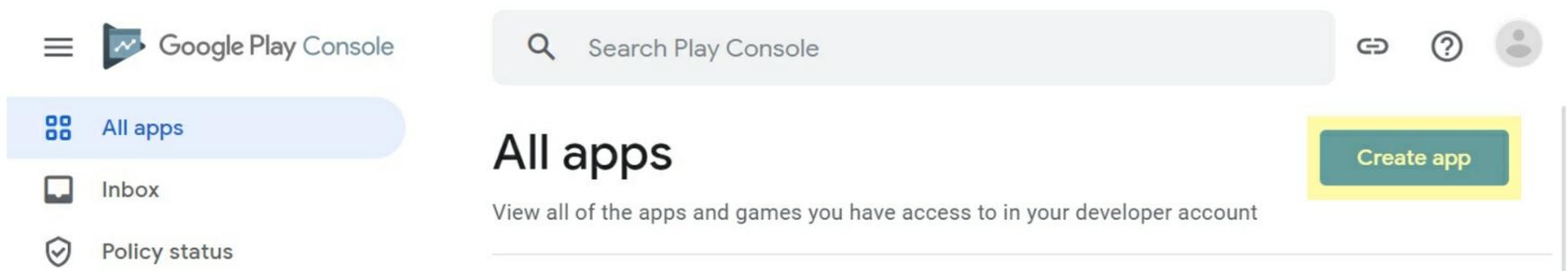


The Android App Bundle file will contain the .aab extension, as shown below:



## Creating a Google Play store listing

To publish an application on the Google Play store, you will require a Google Developer account. You can sign up for a Google Developer account here: <https://play.google.com/console/u/0/signup>. Note you may be required to pay a one-time registration fee of \$25. Once you have set up your Google Developer account, navigate to the Google Play Console dashboard (<https://play.google.com/apps/publish/>) and click the Create app button.



In the Create App window, complete the App details form and provide the necessary information including the application's name, default language, app or game status, and free or paid status. Note that a free application cannot be changed to a paid application once published.

# Create app

## App details

App name	<input type="text" value="My First App"/>
	<small>This is how your app will appear on Google Play</small> <span style="float: right;"><small>12 / 30</small></span>
Default language	<input type="text" value="English (United Kingdom) – en-GB"/>
App or game	<p>You can change this later in Store settings</p> <p><input checked="" type="radio"/> App</p> <p><input type="radio"/> Game</p>
Free or paid	<p>You can edit this later on the Paid app page</p> <p><input checked="" type="radio"/> Free</p> <p><input type="radio"/> Paid</p>
	<div style="border: 1px solid #ccc; padding: 5px;"><p> You can edit this until you publish your app. Once you've published, you can't change a free app to paid.</p></div>

Below the App details form, you will see several declarations that you will need to review before continuing. We are uploading the application in Android App Bundle format, which means the Play App Signing declaration must be checked along with the Developer Program Policies and US export laws declarations to continue. If you review all the declarations and are happy to proceed, then check all the relevant boxes and click the Create app button at the bottom of the window.

## Declarations

- |                            |                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Developer Program Policies | <input checked="" type="checkbox"/> Confirm app meets the Developer Program Policies<br>The application meets <a href="#">Developer Program Policies</a> . Please check out <a href="#">these tips on how to create policy compliant app descriptions</a> to avoid some common reasons for app suspension. If your app or store listing is <a href="#">eligible for advance notice</a> to the Google Play App Review team, <a href="#">contact us</a> prior to publishing. |
| Play App Signing           | <input checked="" type="checkbox"/> Accept the Play App Signing Terms of Service<br>To publish <a href="#">Android App Bundles</a> on Google Play you need to accept the <a href="#">Play App Signing Terms of Service</a> . You will be able to choose your app signing key when creating a release. <a href="#">Learn more</a>                                                                                                                                           |
| US export laws             | <input checked="" type="checkbox"/> Accept US export laws<br>I acknowledge that my software application may be subject to United States export laws, regardless of my location or nationality. I agree that I have complied with all such laws, including any requirements for software with encryption functions. I hereby certify that my application is authorized for export from the United States under these laws. <a href="#">Learn more</a>                       |

Google should then create a draft listing for your application. On the left-hand side of the console, you will find a menu of sections for customising the listing and publishing your application. We'll discuss the sections that are required to publish your application. First, locate the App content section in the Policy category.

# Policy



Policy status



App content

You should see a list of forms you need to fill in. We will discuss each form one by one. You must fill in each form accurately, so take your time.

## To do

### Privacy policy

Not started · Add a privacy policy to your store listing

Adding a privacy policy to your store listing helps provide transparency about how you treat sensitive user and device data

[Start](#)

### Ads

Not started · Let us know whether your app contains ads

You must let us know whether your app contains ads. The 'Contains ads' label is shown next to apps with ads on Google Play. Make sure this information is accurate, and is kept up to date.

[Start](#)

### App access

Not started · Provide instructions on how to access restricted parts of your app

If parts of your app are restricted based on login credentials, memberships, location, or other forms of authentication, provide instructions on how to access them

[Start](#)

## Privacy policy

Enter a web page URL for the privacy policy associated with your application. Your privacy policy should be transparent and describe how your application will handle and store user data. Including a privacy policy is mandatory if your app will be used by persons aged under 13 years old.

## Privacy Policy

Add a privacy policy to your store listing to help provide transparency about how you treat sensitive user and device data. [Learn more](#)

You must add a privacy policy if your target audience includes children under 13. Check the [User Data policy](#) to avoid common violations.

Enter a URL, for example <https://example.com/privacy>

## Ads

If your application will contain advertisements, then you must specify this because Google will display a “Contains ads” label in your store listing. You can find out more information about whether you need to declare advertisements by reading Google’s Ads policy: <https://support.google.com/googleplay/android-developer/answer/9857753>

## Ads

---

Let us know whether your app contains ads. This includes ads delivered by third party ad networks. Make sure this information is accurate and is kept up to date. [Learn more](#)

Ads

Does your app contain ads? Check the [Ads policy](#) to make sure your app is compliant.

- Yes, my app contains ads  
The 'Contains ads' label will be shown next to your app on Google Play. [Learn more](#)
- No, my app does not contain ads

## App access

If access to any area of your app is restricted or requires user authentication, then you must provide Google with valid credentials and instructions for accessing the restricted content. For example, if access to your application is based on the user’s membership tier, then you must provide Google with the login credentials for an account that has access to all tiers. Google will not share this information but may use it to review your application and confirm compliance with their policies and terms and conditions.

## App access

---

If parts of your app are restricted based on login credentials, memberships, location, or other forms of authentication, provide instructions on how to access them. Make sure this information is kept up to date.

Google may use this information to review your app. It won't be shared, or used for any other reason. [Learn more](#)

All functionality is available without special access

All or some functionality is restricted

[+ Add new instructions](#)

You can add up to 5 instructions

Allow Google to use these credentials for performance and app compatibility testing  
Tests are used to improve app compatibility with different Android versions and devices

## Content ratings

The store listing for your application will feature an international age rating coalition (IARC) rating to indicate what age group your application is suitable for. To generate a content age rating for your application, click the Start questionnaire button and provide the required details. For example, you may be asked to provide contact details for yourself/your organisation, information about the category and purpose of your application, and details of any explicit or sensitive content.

# Content ratings

## Receive ratings for your app from official rating authorities

Complete the content rating questionnaire to receive official content ratings for your app. Ratings are displayed on Google Play to help users identify whether your app is suitable for them.



[Start questionnaire](#)

[Learn more](#)

### Target audience and content

Google will ask several questions to determine who the target demographic of your application is. For example, you will be asked to specify what the target age range of your application is. Note you cannot target your application to persons under the age of 13 unless you have received the requisite content rating from the IARC. If your application will be used by children then make sure you read Google's family policy to ensure your application's content and advertisements (if applicable) are suitable: <https://support.google.com/googleplay/android-developer/answer/9893335>

#### Target audience and content

1  Target age — 2  App details — 3  Ads — 4  Store presence — 5  Summary

### Target age

Target age group

What are the target age groups of your app?

Based on your response we'll highlight any actions that you may need to take, and the policies you may need to comply with.

Make sure you review the [Developer Policy Center](#) before publishing your app. Apps that don't comply with these policies may be removed from Google Play. [Learn more](#)

 You can't select age groups below 13 because your app's ESRB rating is 'Teen' or higher.

- 5 and under
- 6-8
- 9-12
- 13-15
- 16-17
- 18 and over

### News apps

You must declare whether or not your application will broadcast news or news articles. If your application will display news, then Google may ask for more information regarding how you source and report the news.

# News apps

Let us know whether your app is a news app. This helps us make sure you comply with the Google Play News policy. [Learn more](#)

---

News apps

Is your app a news app?



No



Yes

I confirm my app complies with the [Google Play News policy](#)

## COVID-19 contact tracing and status apps

If your application will be used for COVID-19-related purposes such as contact tracing, infection status, or vaccination status, then you must report that here. Often, such applications require endorsement from a government authority or recognised healthcare organisation, so Google may ask for evidence of this if necessary.

# COVID-19 contact tracing and status apps

---

To help us understand whether your app is a COVID-19 contact tracing or status app, select all of the statements below that apply to your app.



My app is a publicly available COVID-19 contact tracing app

For example, an app that tracks or monitors infected or exposed individuals for the purpose of COVID-19 response or mitigation



My app is a publicly available COVID-19 status app

For example, an app that verifies an individual's current infection status, vaccination status, or history of infection for the purpose of determining the individual's eligibility for travel or entry into public spaces. [Learn more](#)



My app is not a publicly available COVID-19 contact tracing or status app

## Data safety

You must disclose how your application will gather, process and store user data. Google will ask you to complete a questionnaire detailing your approach to data handling and protection. When completing the questionnaire, make sure you read all the accompanying resources provided by Google and provide comprehensive and accurate answers. For example, you will be asked to describe whether your application handles location data, the user's personal information, financial data and more.

## Location

Show ▾

 0 data types selected

## Personal info

Show ▾

 0 data types selected

## Financial info

Hide ▲

Credit card, debit card, or bank account number 

Purchase history 

Credit info 

Other financial info 

## Health and fitness

Show ▾

 0 data types selected

Once you have completed all the forms listed in the App content section, navigate to the Main store listing section under the Store presence category.

**Grow**



**Store presence**

**Main store listing**

**Custom store listings**

In the Main store listing section, you will get the opportunity to provide information that will be used in the Google Play store listing for your application. This is your chance to describe all the wonderful things about your app and show the user why they should download it. For example, you will be invited to provide a short and full description of the application. The short description will provide a brief overview of the application, which the user can expand to reveal the full description if they wish to learn more. You will also get the opportunity to upload the app icon, a feature graph that will be displayed at the top of the listing, and a link to a YouTube video promoting your app that will be embedded in the listing.

Feature graphic \*



Drop a PNG or JPEG file here to upload

[Upload](#)

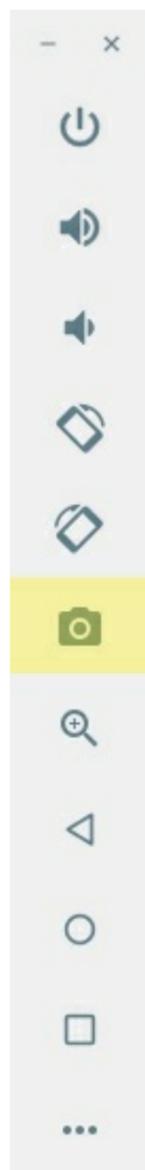
Your feature graphic must be a PNG or JPEG, up to 1MB, and 1,024 px by 500 px

Video

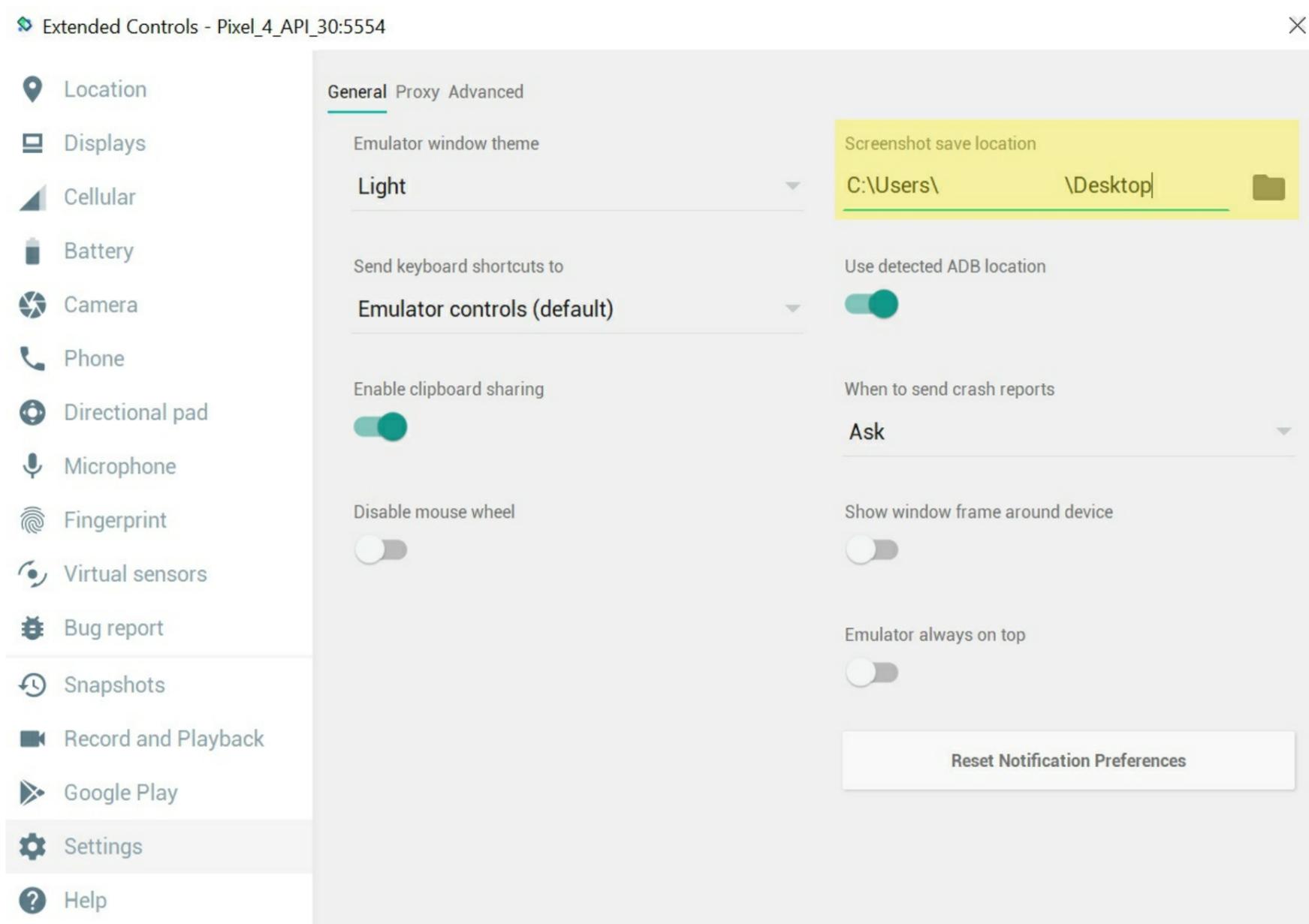
<https://www.youtube.com/watch?v=>

Add a video by entering a YouTube URL. This video must be public or unlisted, ads must be turned off, it must not be age restricted, and it should be landscape.

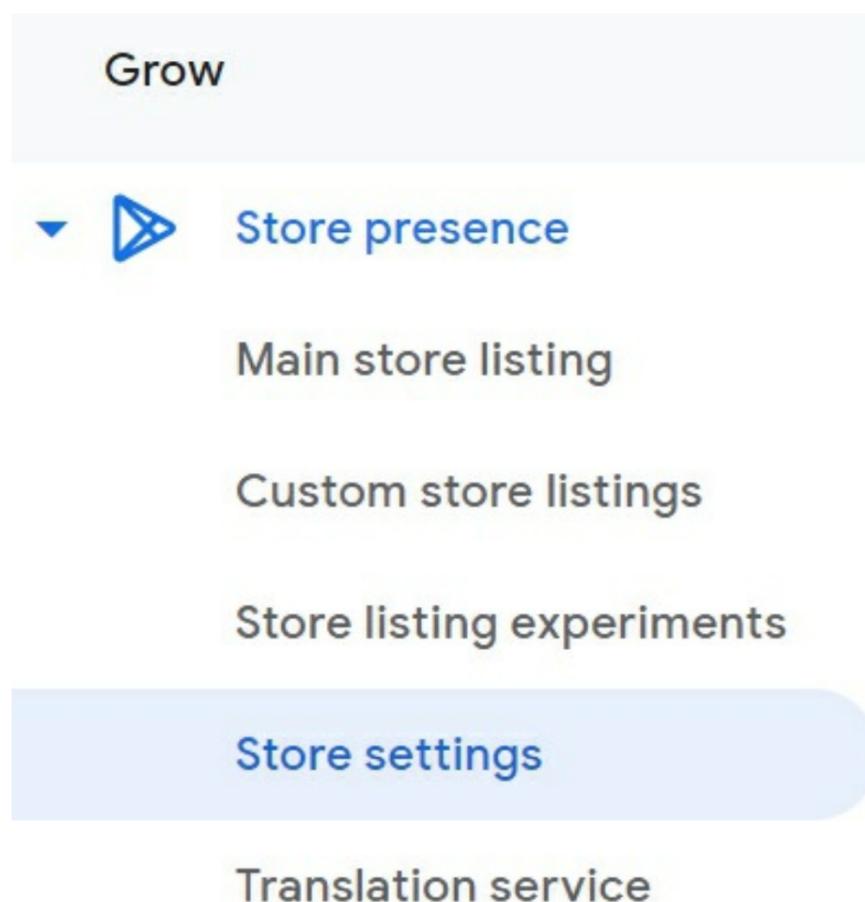
At the bottom of the page, you will also get the opportunity to upload multiple images of your application being used on mobile devices and tablets. You can use images captured on real devices or Android Studio's virtual emulator. If you are using the virtual emulator, then you can capture screenshots using the camera icon in the action bar.



You can view and change the location that screenshots are saved to by opening the virtual emulator's settings.



There is one more mandatory form you need to complete before you can publish your app. Navigate to the Store settings section, which can be found under the Store presence category.



In the Store settings window, you will be invited to define a few details to help drive traffic to your application. First, in the App category subsection, specify whether your application is an app or game, select the most appropriate category, and choose up to five tags. These details will help your application appear in Play store search results. Next, in the Store listing contact details subsection, you can provide contact details such as your organisation's email address and website. These details will be displayed in the Play store listing, so bear in mind that you might receive spam from businesses that discover your listing (I'm speaking from experience!). Finally, at the bottom of the window, you should see a checkbox that allows you to opt-in to external marketing. If you check

this option, then Google may advertise your app outside of the Play store.

### Store settings

---

App or game *	App ▼
Category *	Photography ▼
Tags	Camera, Fashion, Health & fitness, Lifestyle, Photo editor
	<a href="#">Manage tags</a>

---

### Store listing contact details

This information is shown to users on Google Play

Email address *	hello@mybusinessemail.com
Phone number	0123456789
Website	https://www.mywebsite.com

Congratulations, you have now provided all the mandatory information required to list your application in the Play store. The Google Play console provides various other tools for customising your store listing and promoting and monetizing your app. Feel free to explore the Google Play console further and see whether any of the additional tools apply to your listing. Once you are ready to publish your application and upload your project code, proceed to the next section.

## Publish your Android App Bundle

Once you have completed all the tasks in the previous section and configured the Play Store listing for your application, you can upload your Android App Bundle and publish your application. While it can be tempting to release your application to the public right away, it is often beneficial to stagger the release and allow time for testers to try your app and provide feedback. In this way, when you do eventually release your app fully, you can be confident that it is as error-free and robust as possible.

In the Release category of the Google Play console, you should see a production channel and several testing channels.

## Release



Releases overview



Production



Testing

Open testing

Closed testing

Internal testing

Pre-registration



Pre-launch report

The purpose of each channel is as follows:

- **Internal testing** - You can specify up to 100 testers by email who will receive invitations to install your app. Internal testing is often the first level of testing and typically you would invite persons within your organisation or friends and family to test your application at this stage. If you are releasing a paid app, then users who are invited to install the app during the internal testing stage can do so for free. This is not true for the closed and open testing stages.
- **Closed testing** - Similar to the internal testing stage, you must invite users to test your app in the closed testing stage; however, your selection criteria can be broader. For example, you can invite pre-registered users and Google Groups to test your app. The closed testing channel also provides a dedicated channel for testers to provide feedback and other analytics such as a pre-launch report to monitor issues. You can also communicate with testers privately through the Google Play console to gather feedback.
- **Open testing** - Your app will be available on the Google Play store but users will be aware that it is still in the testing stage. You can monitor usage analytics and identify issues via the pre-launch report. User feedback will not impact your Play store rating because the application is still in the testing phase. Also, you can restrict the availability of your application to certain countries.
- **Production** - Your app will be live on the Google Play store for the general public to install. You can restrict your application to certain countries and regions. User reviews will impact your Play Store rating.

To create a testing or production track, navigate to the relevant section and press Create track. Alternatively, you can select an existing track and use that instead.

### Closed testing

Create track

Test pre-release versions of your app with your own groups of testers. [Show more](#)

Depending on your chosen track, you may need to complete additional configuration steps. First, for the internal testing tracks, you will see a Testers tab as shown below:

## Testers

Up to 100 testers can join your internal tests. You can choose more than 100 testers, but only the first 100 to join will be successful.

Testers

<input checked="" type="checkbox"/>	List name	Users	
<input checked="" type="checkbox"/>	Testers	1	<a href="#">→</a>

[Create email list](#)

Feedback URL or email address

hello@mybusinessemail.com

Let testers know how to provide you with feedback

25 / 512

### How testers join your test

Join on the web

Testers can join your test on the web

[↪ Copy link](#)

In the Testers subsection, you can create lists containing the email addresses of people you would like to invite to test your application. You can enter each tester's email address manually or upload a comma-separated CSV file containing the email addresses. If you need to add or remove an email address from a list, then click the blue arrow next to the list. Below the list of testers, you will see a field where you can provide a URL or email address for testers to send feedback to. Finally, at the bottom of the window, you will see a button to generate a link that you can send to invited testers so they can download your application.

For closed and open testing tracks, in addition to inviting and managing tester cohorts via the Testing tab, you will also see a Countries/regions tab. In the Countries/regions tab, you can specify the locations that you would like your application to be available in. These location specifications are based on the country that the user's Google account is registered in, not the user's current location. You must specify at least one location to release your application on the closed and open testing tracks.

## Countries / regions ?

Sync countries / regions with production ?

[Sync countries / regions](#)

Country / region	Status
United Kingdom	<span>✔</span> Available
United States Includes 7 locations	<span>✔</span> Available

Available (2) ▼

[Remove countries / regions](#)

[Add countries / regions](#)

To release your application to a testing or production track, navigate to the relevant section and click the Create new release button.

# Internal testing

Create new release

Create and manage internal testing releases to make your app available to up to 100 internal testers. [Learn more](#)

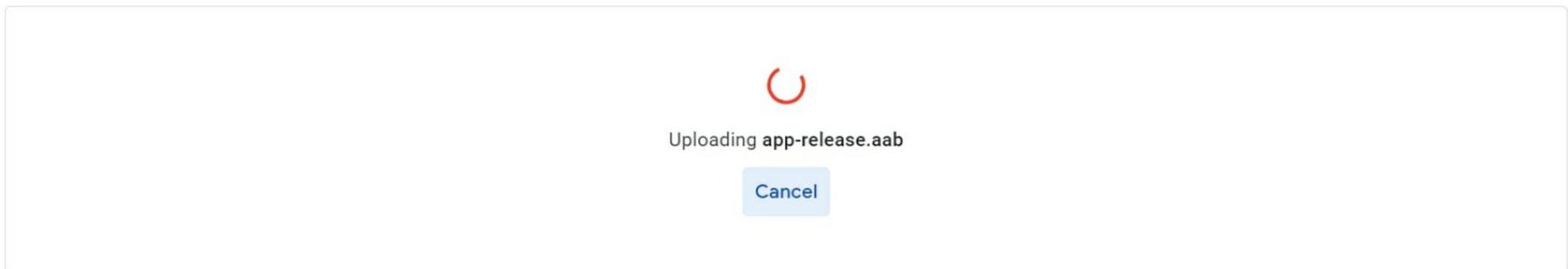
First, in the App bundles section, upload the Android App Bundle that you generated in the Creating an Android Application Bundle section. This file is likely called **app-release.aab**.

## App integrity

✔ Releases signed by Google Play

Google will generate and protect an app signing key for your releases

## App bundles



Note if you are uploading an updated version of the application and you forget to increase the versionCode value in **build.gradle** file, then this is when the Google Play console will let you know. Remember that each new Android App Bundle you upload must have a higher versionCode value than its predecessor. If the bundle is accepted, then its details will be displayed as shown below.

File type	Version	API levels	Target SDK	Screen layouts	ABIs	Required features
App bundle	13 (1.8)	27+	31	4	All	1 <span>⋮</span> <a href="#">→</a>

Next, scroll to the bottom of the window. In the Release details section, you will be able to specify a release name and release notes. The Google Play console will likely auto-populate the release name based on the releaseCode and releaseName value from the application's **build.gradle** file. For the release notes, you should describe the new features and improvements that you have implemented since the last release. Of course, a brief list of updates will be fine; however, feel free to get creative! If you need inspiration then you might like to check out this Medium article where Freddie Harrison discusses the benefits of writing engaging release notes: <https://medium.com/@freddiewrites/writing-great-app-store-release-notes-3f4cf291e9aa>

## Release details

Release name \*

13 (1.8)

8 / 50

This is so you can identify this release, and isn't shown to users on Google Play. We've suggested a name based on the first app bundle or APK in this release, but you can edit it.

Release notes

[Copy from a previous release](#)

<en-GB>  
Performance improvements and bug fixes  
</en-GB>

Release notes provided for 1 language

Let users know what's in your release. Enter release notes for each language within the language tags.

Android will then review your release, which typically takes less than an hour but could be longer if someone needs to manually review your application. Once the review is complete, the Google Play console should show that the release is live. Also, Google will generate a report describing any critical or recommended improvements you could make to your application. To view the report, navigate to the Pre-launch report Overview section. As you read through the list of warnings, yellow warnings can sometimes be ignored if you do not feel like they are relevant; however, if you see any red warnings then you should address them because releasing an application containing critical warnings may harm your ranking in the Google Play store.

The screenshot shows the 'Pre-launch report overview' page in the Google Play console. The sidebar on the left includes sections for 'Releases overview', 'Production', 'Testing', 'Pre-launch report', 'Reach and devices', 'App bundle explorer', and 'Setup'. The 'Pre-launch report' section is expanded to show 'Overview', 'Details', and 'Settings'. The main content area is titled 'Pre-launch report overview' and shows 'App version: 12.aab'. Under the 'Stability' section, there are 8 unique issues (8 warnings), all of which are 'Non-SDK API' warnings. A table lists these warnings with their types and details, including code snippets like 'Landroid/view/View;.>computeFitSystemWindows(Landroid/graphics/Rect;Landroid/graphics/Rect;)Z'. Below the table, there are navigation controls for 'Show rows: 5' and '1 - 5 of 8'. The 'Performance' section below shows 'No issues found'.

You should follow the above steps for creating a new release whenever you need to roll out an update to your application to either a testing or production track.

The Google Play console allows you to promote and demote releases between testing tracks and production. For example, if you are satisfied that the application has passed all the internal testing checks, then you can promote the release to a closed testing track. To do this, locate the release, click Promote release and select the testing track (or production track) that you would like the release to move to. The release will then be available to the regions and testers specified in that test track, or the general public in the case of the production track.

The screenshot shows the 'Releases' page in the Google Play console. The sidebar on the left includes 'Production' and 'Testing' sections. The main content area shows a release for 'Supernova (1.1)' which is 'Available to internal testers'. Below the release information, there are 'Show summary' and 'Promote release' buttons. The 'Promote release' button is clicked, and a dropdown menu is open showing options for 'Closed testing', 'Open testing', and 'Production'. The 'Closed testing' option is selected, and a sub-menu is open showing 'Alpha' and 'Supernova' tracks.

And that's it! Simply progress through the testing tracks and gather as much tester feedback as possible. Once you're confident that your app is in top condition, then release it to a production track. If you ever need to update a release, then increase the versionCode and versionName values in the app's **build.gradle** file, create a new Android App Bundle and upload it as a new release to the relevant testing or production track.

## Summary

Congratulations! You now have all the knowledge you need to create an Android application, release your application on the Google Play store and share your hard work with the world. In completing this section, you have learned the following skills:

- How to create and sign an Android App Bundle for distribution.
- Generate a launcher icon for your application.
- Create a listing in the Google Play Store.
- Upload your application to the Google Play store and navigate through the various testing and production tracks.
- Use the Google Play console's pre-launch report feature to monitor issues and ensure your application is robust and error-free.